IB002
ALGORITMY A DATOVÉ STRUKTURY I

Ivana Černá

Jaro 2019

Fakulta informatiky, Masarykova univerzita

INFORMACE O PŘEDMĚTU

vyučující předmětu

- Ivana Černá *(přednášky)*
- Vojtěch Řehák *(cvičení)*
- Nikola Beneš, Tomáš Brázdil, Jan Obdržálek, Petr Novotný, Jaromír Plhák, Tomáš Effenberger, Dominik Velan, Jan Horáček, Jan Koniarik, Mária Michalíková, Juraj Pančík, Viktória Vozárová, Tatiana Zbončáková *(vedení cvičení)*
- Jan Horáček, Adam Matoušek, Tatiana Zbončáková *(konzultace)*

interaktivní osnova předmětu - kompletní informace o výuce

- organizace výuky přednášky, cvičení, domácí úkoly, konzultace
- studijní materiály slajdy z přednášky, sbírka příkladů, zadání úkolů, rozcestníky, video přednáška ...
- hodnocení předmětu odpovědníky, domácí úkoly, speciální domácí úkol, implementační a znalostní část zkoušky
- diskusní fórum předmětu v IS

LITERATURA

T. Cormen, Ch. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*. Third Edition. MIT Press, 2009

obrázky použité v prezentaci jsou částečně převzaty z uvedené monografie

další odkazy v Interaktivní osnově

SYLABUS

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

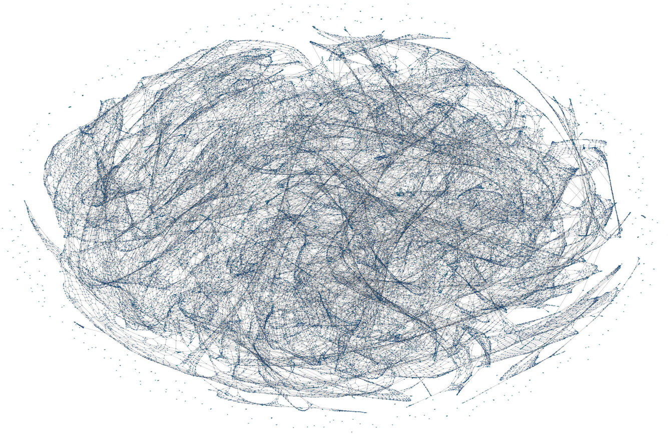
algoritmy

řazení rozdělováním, slučováním, haldou, v lineárním čase

prohledávání grafu, souvislost grafu, cesty v grafu

MOTIVACE

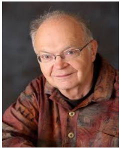
řešit jinak neřešitelné problémy...



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

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



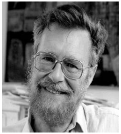
Algorithms: a common language for nature, human, and computer. *— Avi Wigderson*

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.

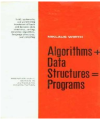
— Linus Torvalds (creator of Linux)

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



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

pro zábavu a zisk...

Část I

Návrh a analýza algoritmů

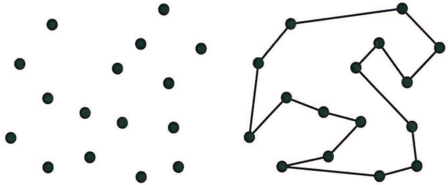
SLOŽITOST A KOREKTNOST ALGORITMŮ

SLOŽITOST A KOREKTNOST ALGORITMŮ

MOTIVACE

PŘÍKLAD

najdi nejkratší cestu pro rozvoz čerstvé pizzy



ALGORITMUS???

řešení 1

vyber počáteční vrchol

$v \leftarrow$ počáteční vrchol

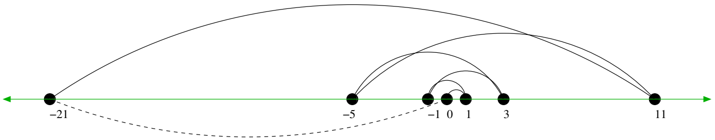
while existuje nenavštívený vrchol **do**

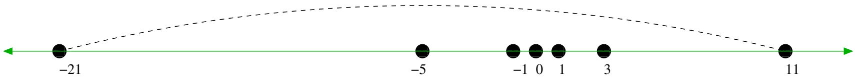
  vyber nenavštívený vrchol, který je nejblíž k v

  $v \leftarrow$ vybraný vrchol **od**

vrať se do počátečního vrcholu

return pořadí, v němž byly vrcholy navštíveny





řešení 2

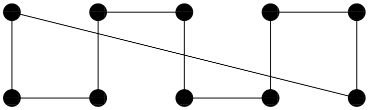
while existují vrcholy, které nejsou spojeny cestou **do**

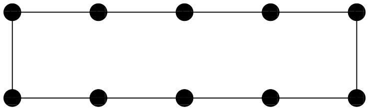
vyber vrcholy, které nejsou spojeny cestou

a jejichž vzdálenost je nejmenší

vybrané vrcholy spoj hranou **od**

přidaním hrany vytvoř cyklus





korektní algoritmus

prozkoumej každý z $n!$ Hamiltonovských cyklů grafu

vyber nejkratší cyklus

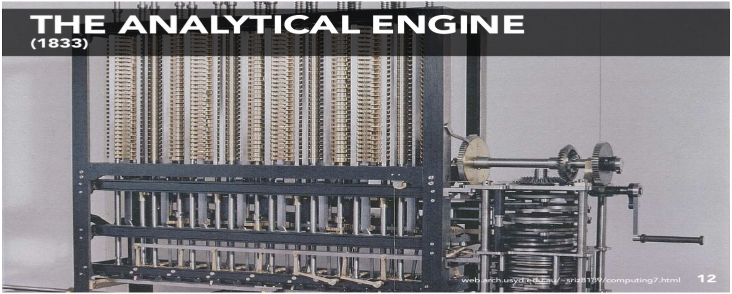
- algoritmus je korektní, protože prověří všechny možnosti
- složitost algoritmu je úměrná počtu všech Hamiltonovských cyklů a algoritmus je proto nepoužitelný již pro velmi malé grafy

SLOŽITOST A KOREKTNOST ALGORITMŮ

ANALÝZA SLOŽITOSTI

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?

— *Charles Babage, 1864*



kolikrát musíme zatočit klikou?

ČASOVÁ SLOŽITOST

časová složitost **výpočtu** je 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 N, jestliže prvek x se v posloupnosti nevyskytuje

Linear Search(A)

```
answer ← N
for i = 1 to n do
    if A[i] = x then answer ← i fi
od
return answer
```

optimalizace I

BETTER LINEAR SEARCH(A)

1 **for** $i = 1$ **to** n **do**
2   **if** $A[i] = x$ **then** **return** i **fi**
3 **od**
4 **return** N

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

optimalizace II

Sentinel Linear Search(A)

1 $\mathit{last} \leftarrow A[n]$
2 $A[n] \leftarrow x$
3 $i \leftarrow 1$
4 **while** $A[i] \neq x$ **do** $i \leftarrow i + 1$ **od**
5 $A[n] \leftarrow \mathit{last}$
6 **if** $i < n \vee A[n] = x$
7 **then return** i
8 **else return** N **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 vyhledávání

```
1 answer ← N
2 for i = 1 to n do
3     if A[i] = x
4        then answer ← i fi od
5 return answer
```

- označme t_i složitost 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

Better Linear Search(A)

for $i = 1$ **to** n **do if** $A[i] = x$ **then return** i **fi od**

return N

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

Sentinel Linear Search(A)

$last \leftarrow A[n], A[n] \leftarrow x, i \leftarrow 1$

while $A[i] \neq x$ **do** $i \leftarrow i + 1$ **od**

$A[n] \leftarrow last$

if $i < n \vee A[n] = x$ **then return** i **else return** N **fi**

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

rozdíl je v konstantních faktorech

SLOŽITOST A KOREKTNOST ALGORITMŮ

KOREKTNOST ALGORITMŮ

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

Better Linear Search(A)

for $i = 1$ **to** n **do if** $A[i] = x$ **then return** i **fi od**

return N

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.

SLOŽITOST A KOREKTNOST ALGORITMŮ

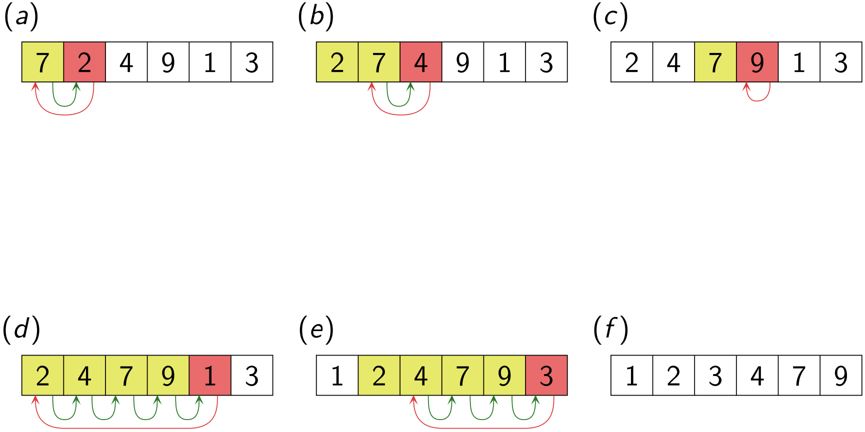
ŘAZENÍ VKLÁDÁNÍM

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 $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

ALGRITMUS ŘAZENÍ VKLÁDÁNÍM



Insert Sort(A)

Vstup: $A[1 \ldots n]$

```
for j = 2 to A.length do
    key ← A[j]
    // Vlož A[j] do seřazené postupnosti A[1 ... j − 1]
    i ← j − 1
    while i > 0 ∧ A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    od
    A[i + 1] ← key
od
```

Korektnost – 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 $A[j-1], A[j-2], \dots$ se 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

Insert Sort(A)	**cena**	**počet**
for $j = 2$ **to** $A.length$ **do**	c_1	n
$\quad key \leftarrow A[j]$	c_2	$n-1$
$\quad i \leftarrow j-1$	c_3	$n-1$
$\quad$ **while** $i > 0 \wedge A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad i \leftarrow i-1$ **od**	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] \leftarrow key$ **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 — nejlepší případ

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

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) \\
&\quad + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_4(n-1) + c_4(n-1) + c_7(n-1) \\
&= an + b
\end{aligned}
$$

lineární složitost

složitost — nejhorší případ

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

SLOŽITOST A KOREKTNOST ALGORITMŮ

ASYMPTOTICKÁ NOTACE

ASYMPTOTICKÁ NOTACE

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

příklad

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

$$
\begin{aligned}
&= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n \\
&\quad - (c_2 + c_3 + c_4 + c_7) \\
&= an^2 + bn + c
\end{aligned}
$$

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

TYPY NOTACÍ

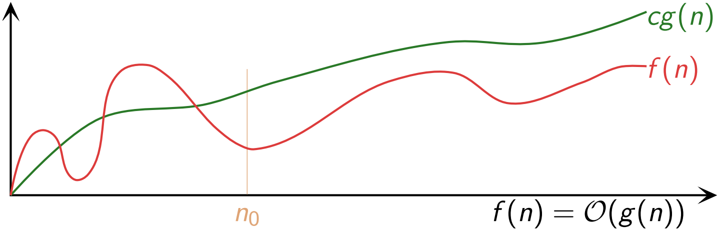
- $f \in \mathcal{O}(g)$ znamená, že $C \cdot g(n)$ je **horní hranicí** pro $f(n)$
- $f \in \Omega(g)$ znamená, že $C \cdot g(n)$ je **dolní hranicí** pro $f(n)$
- $f \in \Theta(g)$ znamená, že $C_1 \cdot g(n)$ je horní hranicí pro $f(n)$ a
  $C_2 \cdot g(n)$ je dolní hranicí pro $f(n)$

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

C, C_1, C_2 jsou konstanty nezávislé na n

$\mathcal{O}$ NOTACE

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



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

$\mathcal{O}$ notace - příklady

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

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

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

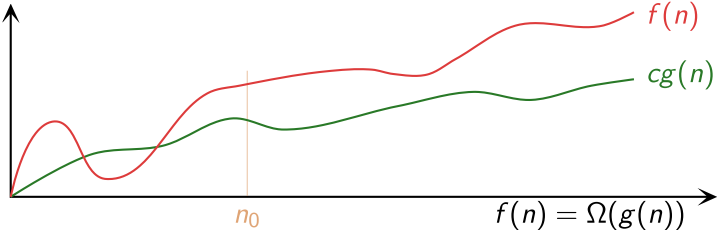
  protože $8n^2 - 88n + 8 < 1n^3$ pro všechna $n \geq 10$

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

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

Ω NOTACE

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



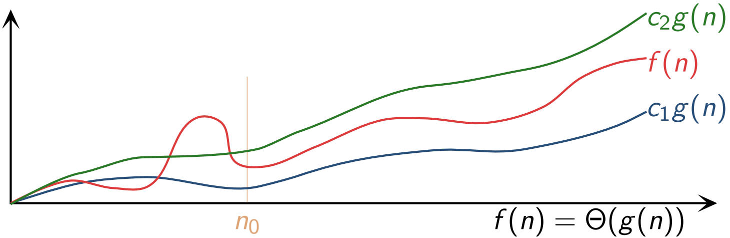
funkce f *neroste asymptoticky pomaleji* než funkce g

Ω notace - příklady

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

Θ NOTACE

$f \in \Theta(g)$ 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

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

- $8n^2 - 88n + 8 \notin \Theta(n^3)$ protože $8n^2 - 88n + 8 \notin \Omega(n^3)$

- $8n^2 - 88n + 8 \notin \Theta(n)$ protože $8n^2 - 88n + 8 \notin \mathcal{O}(n)$

Θ notace - příklad

chceme dokázat platnost vztahu $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$

- 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

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

reflexivita $f(n) \in \Theta(f(n))$ podobně pro $\mathcal{O}$ a Ω

symetrie $f(n) \in \Theta(g(n))$ právě když $g(n) \in \Theta(f(n))$

transpozice $f(n) \in \mathcal{O}(g(n))$ právě když $g(n) \in \Omega(f(n))$

poznámka: ne každá dvojice funkcí je asymptoticky srovnatelná

ROZDĚL A PANUJ

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.

ROZDĚL A PANUJ

MAXIMÁLNÍ A MINIMÁLNÍ PRVEK

PROBLÉM MAXIMÁLNÍHO A MINIMÁLNÍHO PRVKU

vstupem je pole $S[1 \dots n]$

MaxMin Iterative(S)

$max \leftarrow S[1]$
$min \leftarrow S[1]$
for $i = 2$ **to** n **do**
  if $S[i] > max$ **then** $max \leftarrow S[i]$ **fi**
  if $S[i] < min$ **then** $min \leftarrow S[i]$ **fi**
od

složitost výpočtu = počet porovnání prvků

celkem $2(n-1)$ porovnání

aplikace přístupu Rozděl a panuj

- posloupnost rozděl na dvě (stejně velké) podposloupnosti
- najdi minimální a maximální prvek v obou podposloupnostech
- kombinuj řešení podproblémů: maximálním prvek posloupnosti je větší z maximálních prvků podposloupností; symetricky pro minimálním prvek

$\textsc{MaxMin}(S, l, r)$

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

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

korektnost algoritmu

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

$n = 1, n = 2$ 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 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 algoritmu

- n je velikost (délka) vstupní posloupnosti
- podproblémy mají velikosti $\lfloor n/2 \rfloor$ a $\lceil n/2 \rceil$
- $T(n)$ je počet porovnání ve výpočtu na vstupu délky n

$$\begin{aligned} T(n) = {} & \text{složitost rozdělení} \\ & + \text{složitost řešení podproblémů} \\ & + \text{složitost kombinace} \end{aligned}$$

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

složitost algoritmu - explicitní vyjádření

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

indukcí vzhledem k n ověříme, že pro $n > 1$ platí $T(n) \leq \frac{5}{3}n - 2$

indukční základ $T(2) = 1 \leq \frac{5}{3} \cdot 2 - 2$

$T(3) = 0 + 1 + 2 \leq \frac{5}{3} \cdot 3 - 2$

indukční předpoklad nerovnost platí pro všechny hodnoty i, $2 \leq i < n$

platnost pro n

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

Kdo je nejrychlejší?

$\text{MIN1}(S, l, r)$

minimum $\leftarrow S[l]$

for $i = l + 1$ **to** r **do**

  if *minimum* $> S[i]$ **then** *minimum* $\leftarrow S[i]$ **fi od**

return *minimum*

$\text{MIN2}(S, l, r)$

if $r = l$ **then return** $S[r]$ **fi**

if $r > l$ **then** $A \leftarrow \text{MIN2}(S, l, \lfloor (l + r)/2 \rfloor)$

  $B \leftarrow \text{MIN2}(S, \lfloor (l + r)/2 \rfloor + 1, r)$

  return $\min(A, B)$ **fi**

$\text{MIN3}(S, l, r)$

if $r = l$ **then return** $S[r]$ **fi**

if $r > l$ **then** $A \leftarrow \text{MIN3}(S, l, r - 1)$

  return $\min(A, S[r])$ **fi**

ROZDĚL A PANUJ

SLOŽITOST REKURZIVNÍCH ALGORITMŮ

SLOŽITOST REKURZIVNÍCH ALGORITMŮ

- nechť $T(n)$ je časová složitost výpočtu na vstupu délky n
- složitost zapíšeme pomocí rekurentní rovnice, která vyjadřuje $T(n)$ pomocí složitosti výpočtů na menších vstupech
- pro malý vstup ($n \leq c$) je časová složitost ohraničená konstantou
- velký vstup rozdělíme na k podproblémů velikosti $n_1, \ldots, n_k$; řešení podproblému velikosti n_i má časovou složitost $T(n_i)$
- nechť $D(n)$ je složitost konstrukce podproblémů a $C(n)$ je složitost kombinaci řešení podproblémů a nalezení řešení původního problému

$$T(n) = \begin{cases} \Theta(1) & \text{pro } n \leq c \\ \sum_{i=1}^{k} T(n_i) + D(n) + C(n) & \text{jinak} \end{cases}$$

jak najít řešení (explicitní vyjádření funkce $T(n)$) rekurentní 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. $T(n) \in \mathcal{O}(n \log n)$
2. indukcí dokážeme, že $T(n) \leq cn \log n$ pro dostatečně velké n a vhodně zvolenou konstantu c

dokazujeme $T(n) \leq cn \log n$ pro rovnici

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

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 $T(2) \leq c \cdot 2 \log 2$ a současně platí $T(3) \leq c \cdot 3 \log 3$

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

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cnlogn && \text{za předpokladu } c \geq 1 \end{aligned}$$

dokázali jsme, že pro všechna $n \geq 2$ platí $T(n) \leq 2n \log n$

METODA REKURZIVNÍHO STROMU

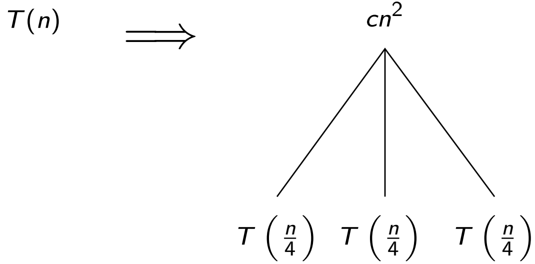
- „rozbalování rekurze“
- přehledný zápis pomocí stromu, jehož vrcholy vyjadřují složitost jednotlivých rekurzivních volání
- vrchol stromu je ohodnocen složitostí dekompozice a kompozice
- synové vrcholu odpovídají jednotlivým rekurzivním voláním
- výslednou složitost vypočítáme jako sumu ohodnocení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

metodu můžeme použít pro

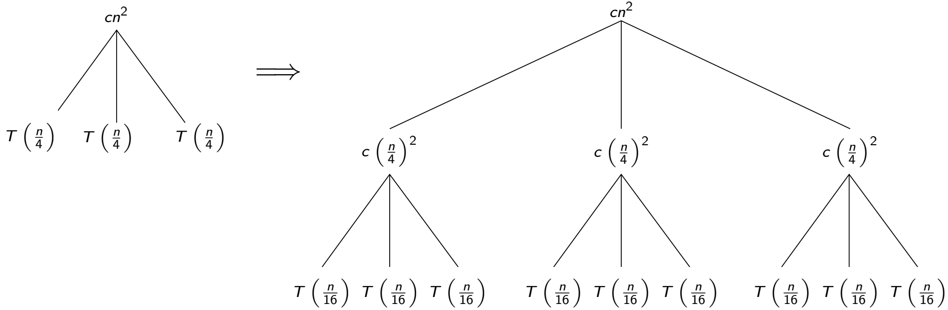
- nalezení přesného řešení (je nutné přesné počítání)
- pro získání odhadu na řešení rekurentní rovnice; pro důkaz řešení se pak použije substituční metoda

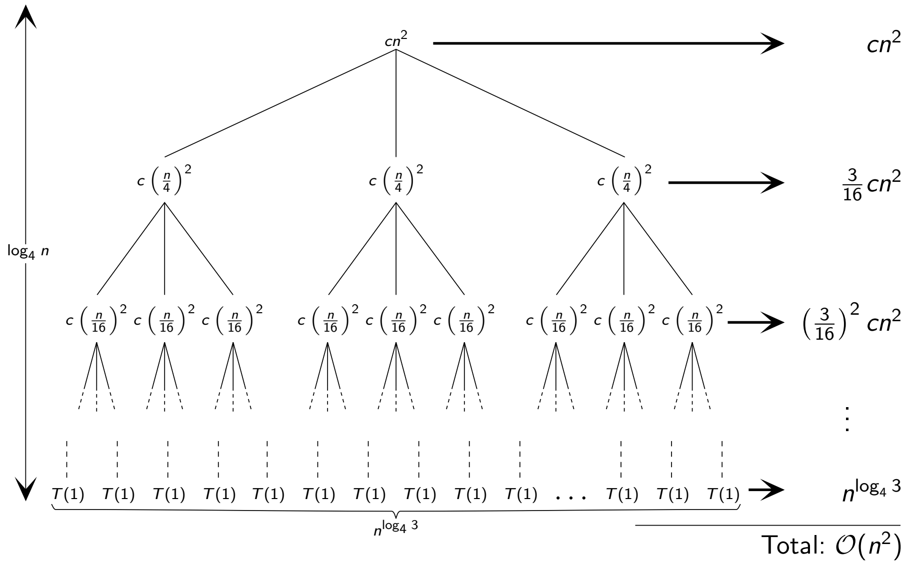
$$T(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 3T(\lfloor n/4 \rfloor) + cn^2 & \text{jinak} \end{cases}$$

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



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





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

- 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

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

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

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

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

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

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

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

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

$$T(n) = \begin{cases} \Theta(f(n)) & \text{když } af(n/b) = cf(n) \text{ pro konstantu } c < 1 \\ \Theta(n^{\log_b a}) & \text{když } af(n/b) = df(n) \text{ pro konstantu } d > 1 \\ \Theta(f(n)\log_b n) & \text{když } af(n/b) = f(n) \ . \end{cases}$$

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

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

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

příklady použití kuchařkové věty I

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

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

- $T(n) = 4T(n/2) + n^2 \implies T(n) \in \Theta(n^2 \log n)$
  případ 2, $a = 4, b = 2, c = 2,\ 4 = 2^2$

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

příklady použití kuchařkové věty II

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

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

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

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

příklady použití kuchařkové věty III

Hanojské věže

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

MergeSort

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

násobení celých čísel

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

Strassenův algoritmus pro násobení celých čísel

$$T(n) = 7T(n/2) + \mathcal{O}(n^2)$$
$$T(n) \in \mathcal{O}(n^{\log_2 7})$$

ROZDĚL A PANUJ

JAK NEPOUŽÍVAT REKURZI

JAK NEPOUŽÍVAT REKURZI

nedefinovaný výpočet *chybí základ rekurze*

$\textsc{Bad\_Factorial}(n)$

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

nekonečný výpočet

$\textsc{Awful\_Factorial}(n)$

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

$\textsc{Beta}(n)$

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

neefektivní výpočet

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

$\textsc{RecFibo}(n)$

if $n < 2$ **then return** n

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

algoritmus $\textsc{RecFibo}$ má exponenciální časovou zložitost

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

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

neefektivní výpočet - pokračování

iterativní algoritmus pro výpočet Fibonacciho posloupnosti

$\textsc{IterFibo}(n)$

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

algoritmus $\textsc{IterFibo}$ má lineární časovou zložitost, $T(n) \in \Theta(n)$

ROZDĚL A PANUJ

PROBLÉM MAXIMÁLNÍ PODPOSLOUPNOSTI

JAK POUŽÍVAT REKURZI - 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*?

definice podproblémů

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

```
leftsum ← −∞
sum ← 0
for i = mid downto low do
    sum ← sum + A[i]
    if sum > leftsum then leftsum ← sum
                          maxleft ← i fi od
rightsum ← −∞
sum ← 0
for j = mid + 1 to high do
    sum ← sum + A[j]
    if sum > rightsum then rightsum ← sum
                           maxright ← j fi od
return (maxleft, maxright, leftsum + rightsum)
```

algoritmus

FIND_MAXIMUM_SUBARRAY(*A*, *low*, *high*)

```
if high = low
   then return (low, high, A[low])
    else mid = ⌈(low + high)/2⌉
           (leftlow, lefthigh, leftsum) ← F_M_S(A, low, mid)
           (rightlow, righthigh, righttsum) ← F_M_S(A, mid + 1, high)
           (crosslow, crosshigh, crosssum) ← F_M_C_S(A, low, mid, high)
if leftsum ≥ rightsum ∧ leftsum ≥ crosssum
   then return (leftlow, lefthigh, leftsum) fi
if leftsum ≤ rightsum ∧ rightsum ≥ crosssum
   then return (rightlow, righthigh, rightsum)
    else return (crosslow, crosshigh, crosssum) fi
```

složitost algoritmu

procedura $\textsc{Find\_Max\_Crossing\_Subarray}(A, \mathit{low}, \mathit{mid}, \mathit{high})$

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

algoritmus $\textsc{Find\_Maximum\_Subarray}$

- dekompozice a kompozice v konstantním čase
- řešení problému (C) v čase $\Theta(n)$
- $$T(n) = \begin{cases} \Theta(1) & \text{pro } n = 1 \\ 2T(n/2) + \Theta(n) & \text{jinak} \end{cases}$$
- $T(n) = \Theta(n \log n)$

ROZDĚL A PANUJ

REKURZIVNÍ VS ITERATIVNÍ PŘÍSTUP

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

TAIL REKURZE

- každý rekurzivní algoritmus lze převést na iterativní
- simulace zásobníku volání
- jednoduchý přepis v případě *tail rekurze*

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 $x \cdot y$

přepis tail rekurze na iterativní algoritmus — příklad 1

$\mathrm{F}(x, y)$

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 $i = 1$ **to** y **do**
$ret \leftarrow ret \cdot y + ret$ **od**
return ret

přepis tail rekurze na iterativní algoritmus — příklad 2

$\textsc{Bin Search}(x, A, \mathit{left}, \mathit{right})$

if $\mathit{right} = \mathit{left}$ **then return** $A[\mathit{left}] == x$ **fi**
if $\mathit{right} < \mathit{left}$ **then** $\mathit{mid} = \lfloor(\mathit{left} + \mathit{right})/2\rfloor$
    if $A[\mathit{mid}] = x$ **then return** *true* **fi**
    if $A[\mathit{mid}] < x$ **then** $\mathit{left} \leftarrow \mathit{mid} + 1$
        else $\mathit{right} \leftarrow \mathit{mid}$ **fi**
    $\textsc{Bin Search}(x, A, \mathit{left}, \mathit{right})$
fi

$\textsc{Bin Search}(x, A, \mathit{left}, \mathit{right})$

while $\mathit{right} < \mathit{left}$ **do** $\mathit{mid} = \lfloor(\mathit{left} + \mathit{right})/2\rfloor$
    if $A[\mathit{mid}] = x$ **then return** *true* **fi**
    if $A[\mathit{mid}] < x$ **then** $\mathit{left} \leftarrow \mathit{mid} + 1$
        else $\mathit{right} \leftarrow \mathit{mid}$ **fi**
od

Část II

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

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 SLÉVÁNÍM

ŘAZENÍ SLÉVÁNÍM (MERGE SORT)

rozděl posloupnost na dvě stejně velké podposloupnosti

vyřeš obě podposloupnosti (rekurzivně)

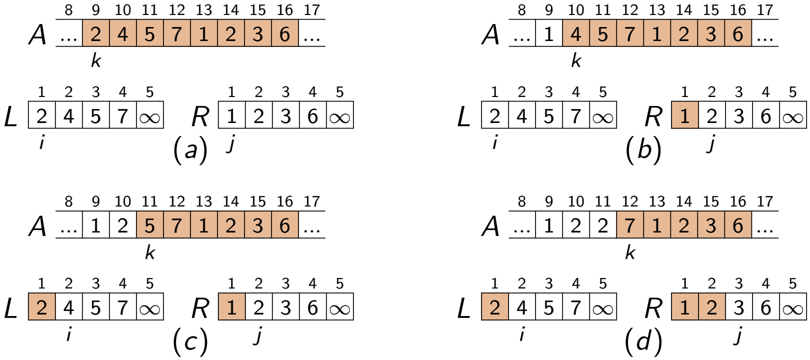
kombinuj dvě seřazené podposloupnosti do jedné

spojení dvou seřazených posloupností - Merge

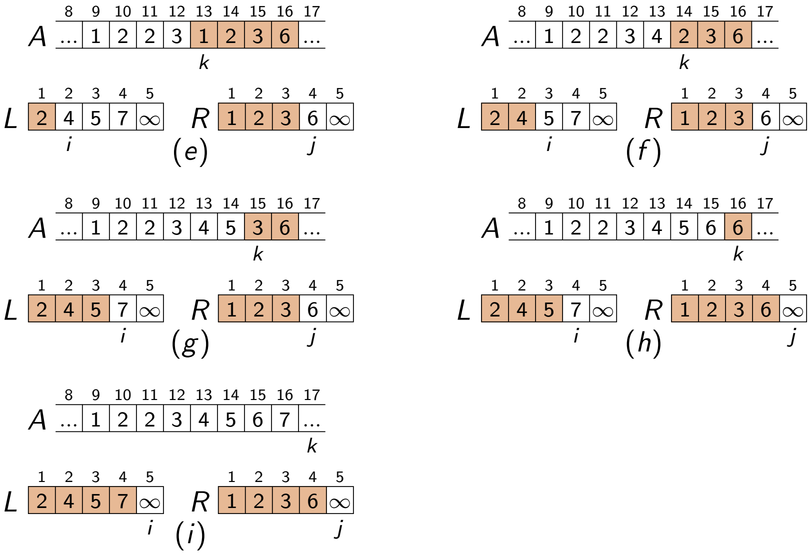
otázkou je, jak spojit dvě seřazené posloupnosti do jedné, která bude seřazená

- při slévání porovnáváme vedoucí prvky obou posloupností
- menší z porovnávaných prvků přesuneme do výslední posloupnosti

- procedura MERGE má 4 parametry: pole A 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)


(e)
A … 1 2 2 3 1 2 3 6 …
L 2 4 5 7 ∞
R 1 2 3 6 ∞
(f)
A … 1 2 2 3 4 2 3 6 …
L 2 4 5 7 ∞
R 1 2 3 6 ∞
(g)
A … 1 2 2 3 4 5 3 6 …
L 2 4 5 7 ∞
R 1 2 3 6 ∞
(h)
A … 1 2 2 3 4 5 6 6 …
L 2 4 5 7 ∞
R 1 2 3 6 ∞
(i)
A … 1 2 2 3 4 5 6 7 …
L 2 4 5 7 ∞
R 1 2 3 6 ∞

$\textsc{Merge}(A, p, q, r)$

předpoklad: posloupnosti $A[p \dots q]$ a $A[q+1 \dots r]$ jsou seřazené

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
//nechť $L[1 \dots n_1 + 1]$ a $R[1 \dots n_2 + 1]$ jsou nová pole
for $i = 1$ **to** n_1 **do** $L[i] \leftarrow A[p + i - 1]$ **od**
for $j = 1$ **to** n_2 **do** $R[j] \leftarrow A[q + j]$ **od**
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k = p$ **to** r **do**
    if $L[i] \leq R[j]$ **then** $A[k] \leftarrow L[i]$
                        $i \leftarrow i + 1$
                else $A[k] \leftarrow R[j]$
                        $j \leftarrow j + 1$ **fi**
od

korektnost procedury MERGE

invariant for cyklu

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 $\Theta(n)$

algoritmus Merge Sort

- využívá proceduru $\textsc{Merge}$
- pro seřazení celé posloupnosti voláme $\textsc{Merge Sort}(A, 1, A.length)$

$\textsc{Merge Sort}(A, p, r)$

```
if p < r then q ← ⌊(p + r)/2⌋
              Merge Sort(A, p, q)
              Merge Sort(A, q + 1, r)
              Merge(A, p, q, r) fi
```

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 $\textsc{Merge}$ je $\Theta(n)$

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

složitost algoritmu $\textsc{Merge Sort}$ je $T(n) \in \Theta(n \log n)$

ŘAZENÍ SLÉVÁNÍM

PROBLÉM INVERZÍ

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

rozděl posloupnost rozdělíme na dvě (stejně velké) podposloupnosti

vyřeš v každé z podposloupností spočítáme inverze

kombinuj k počtu inverzí z podposloupností připočítáme inverze mezi prvky různých podposloupností

- cílem je navrhnout algoritmus s lepší složitostí než je složitost naivního algoritmu
- jestliže chceme, aby časová složitost rekurzivního algoritmu byla $T(n) \in \mathcal{O}(n \log n)$, tak musí platit $T(n) \leq 2T(n/2) + c \cdot n$, tj. složitost výpočtu rozdělování a kombinace nesmí být větší než lineární

jak spočítat inverze mezi prvky různých posloupností 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
- složitost není lineární

problém inverzí - kombinuj - pokus 2

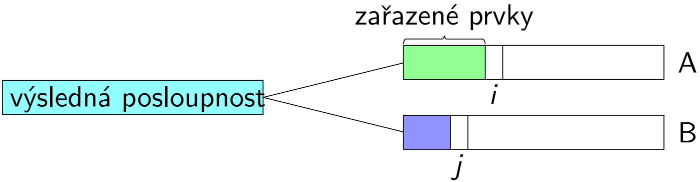
vstupem jsou posloupnosti $A = (a_1, a_2, \ldots, a_k)$ a $B = (b_1, b_2, \ldots, b_l)$

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 $\textsc{Merge}$

- 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 $\mathbf{a_i} < \mathbf{b_j}$, tak do výsledné posloupnosti přesuneme a_i

  $a_i < b_j < b_{j+1} < b_{j+2} < \dots$
  a_i není v inverzi se žádným z $b_j, b_{j+1}, b_{j+2} \dots$

- jestliže $\mathbf{a_i} > \mathbf{b_j}$, tak do výsledné posloupnosti přesuneme b_j

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

Merge_and_Count(A, B)

```
i ← 1; j ← 1
//i (j) je index prvného nezařazeného prvku z A (B)
Count ← 0
//Count je počet nalezených inverzí
while seznamy A, B jsou neprázdné do
        porovnej a_i a b_j
        menší z prvků zařaď do výsledného seznamu
        if b_j < a_i then zvyš Count o počet nezařazených prvků z A fi
        zvyš index i resp. j od
if jeden seznam je prázdný
   then zařaď zbývající prvky do výsledného seznamu fi
return Count a výsledný seznam
```

SORT_AND_COUNT(L)

if *length*(L) $= 1$
  then $r \leftarrow 0$
  else $A \leftarrow$ levá polovina L
    $B \leftarrow$ pravá polovina L
    $(r_A, A) \leftarrow$ SORT_AND_COUNT(A)
    $(r_B, B) \leftarrow$ SORT_AND_COUNT(B)
    $(r, L) \leftarrow$ MERGE_AND_COUNT(A, B)
    $r \leftarrow r + r_A + r_B$ **fi**
return (r, L)

složitost algoritmu

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

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

QUICKSORT

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

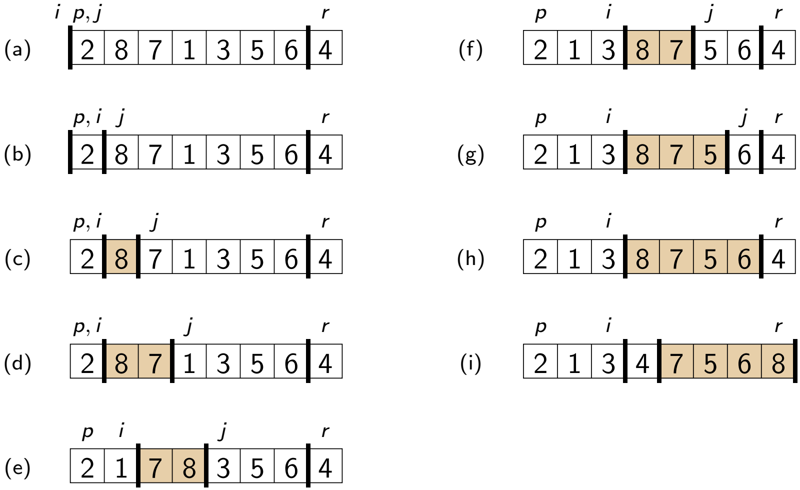
rozděl posloupnost na dvě posloupnosti poloviční velikosti.

vyřeš obě podposloupnosti (rekurzivně) seřaď

kombinuj spoj dvě seřazené podposloupnosti do jedné

Quicksort — principy

- 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


(a) i p, j r
2 8 7 1 3 5 6 4
(b) p, i j r
2 8 7 1 3 5 6 4
(c) p, i j r
2 8 7 1 3 5 6 4
(d) p, i j r
2 8 7 1 3 5 6 4
(e) p i j r
2 1 7 8 3 5 6 4
(f) p i j r
2 1 3 8 7 5 6 4
(g) p i j r
2 1 3 8 7 5 6 4
(h) p i r
2 1 3 8 7 5 6 4
(i) p i r
2 1 3 4 7 5 6 8

$\textsc{Quicksort}(A, p, r)$

if $p < r$
  then $q \leftarrow \textsc{Partition}(A, p, r)$
    $\textsc{Quicksort}(A, p, q - 1)$
    $\textsc{Quicksort}(A, q + 1, r)$ **fi**

$\textsc{Partition}(A, p, r)$

$pivot \leftarrow A[r]$
$i \leftarrow p - 1$
for $j = p$ **to** r **do**
  if $A[j] \leq pivot$ **then** $i \leftarrow i + 1$
    vyměň $A[i]$ a $A[j]$ **fi** **od**
return i

Quicksort — korektnost

chceme dokázat, že procedura $\textsc{Partition}$ 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 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

Invariant cyklu

na začátku 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*

Invariant cyklu

na začátku 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 $A[i] = pivot$

Quicksort — složitost

složitost v nejhorším případě

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

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

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

složitost v nejlepším případě

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

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

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

průměrná složitost

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

Quicksort — Hoare Partition

- pivotem je první prvek posloupnosti
- postupujeme od obou konců posloupnosti až do chvíle, než jsou detekovány prvky, které jsou v opačném pořadí vůči pivotu; prvky si vymění svou pozici
- funkce vrátí index j takový, že všechny prvky v $A[p \dots j]$ jsou menší anebo rovny prvkům v $A[j+1 \dots r]$

Hoare Partition(A, p, r)

$x \leftarrow A[p]$
$i \leftarrow p - 1$
$j \leftarrow r + 1$
while *true* **do**
  repeat $j \leftarrow j - 1$ **until** $A[j] \leq x$ **od**
  repeat $i \leftarrow i + 1$ **until** $A[i] \geq x$ **od**
  if $i < j$ **then** swap $A[i]$ a $A[j]$
    else return j **fi**
od

Quicksort — alternativní rozdělování

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

Quicksort — iterativní verze

Tail Recursive Quicksort(A, p, r)

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

Iterative Quicksort(A, p, r)

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

SLOŽITOST PROBLÉMU ŘAZENÍ

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

- búno vstupní posloupnost obsahuje vzájemně různé prvky
- každé porovnání určí větší ze dvou prvků
- výpočet algoritmu můžeme popsat *rozhodovacím stromem*, každý vnitřní uzel stromu reprezentuje jedno porovnání a jeho synové odpovídající výsledku porovnání $<$ resp. $>$
- výpočet na konkrétním vstupu představuje cestu v rozhodovacím stromě z kořene do listu; jeho složitost je úměrná délce cesty
- každý list jednoznačně určuje seřazení vstupních prvků
- algoritmus musí mít možnost vypočítat každou možnou permutaci vstupních prvků; počet různých permutací je $n!$
- strom musí mít alespoň $n!$ listů $\Rightarrow$ má hloubku alespoň $\log(n!) \in \Omega(n \log n)$

ŘAZENÍ HALDOU

ŘAZENÍ HALDOU

ŘAZENÍ HALDOU

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

- efektivně najít největší prvek
- efektivně z ní odstranit největší prvek

KOŘENOVÝ STROM

- strom s vyznačeným vrcholem r nazýváme 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

definice viz učebný text Matematické Základy Informatiky prof. Hliněného

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

- 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

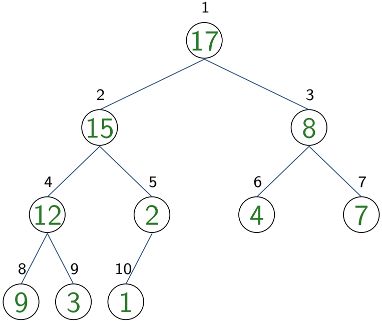
- je **úplný** binární strom s vlastností haldy
- binární strom je úplný, pokud jsou všechny jeho hladiny kromě poslední úplně zaplněny a v poslední hladině leží listy co nejvíce vlevo

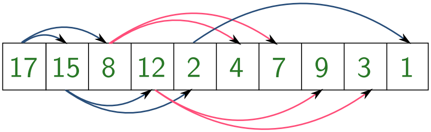
maximová vs. minimová halda

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 $A[i]$
- klíč levého syna uzlu k bude uložen v poli na pozici $2k$ a klíč pravé syna uzlu k na pozici $2k+1$
- klíč otce uzlu k se bude nacházet na pozici $\lfloor k/2 \rfloor$





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 $2i + 1$

VYBUDOVÁNÍ HALDY

- vstupem je posloupnost klíčů uložená v poli $A[1 \ldots n]$
- 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 $\mathcal{O}(n \log n)$

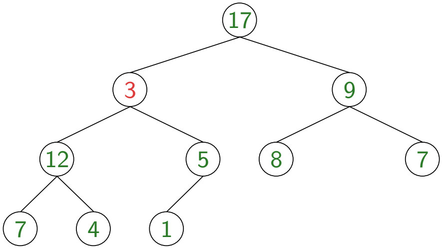
vybudování haldy — varianta A

- postupujeme od uzlu n k uzlu 1
- nechť i je aktuálně spracovávaný uzel; pak všechny uzly j, pro $i < j \leq n$, splňují vlastnost haldy
- po spracování uzlu i splňují vlastnost haldy všechny uzly $j \geq i$

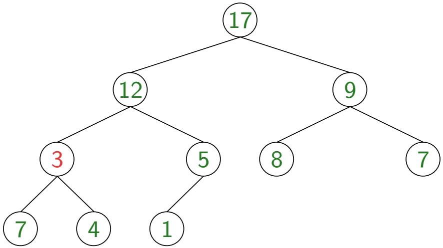
procedura $\textsc{Max\_Heapify}(A, i)$

- 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 i má vlastnost haldy
- úprava je založena na přesunu klíče $A[i]$ směrem dolů

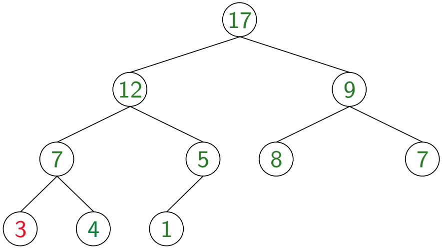
MAX_HEAPIFY$(A, 2)$



(a)



(b)



(c)

MAX_HEAPIFY(A, i)

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ A.heap_size ∧ A[l] > A[i]
   then largest ← l
    else largest ← i fi
if r ≤ A.heap_size ∧ A[r] > A[largest]
   then largest ← r fi
if largest ≠ i
   then swap A[i] a A[largest]
          MAX_HEAPIFY(A, largest) fi
```

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

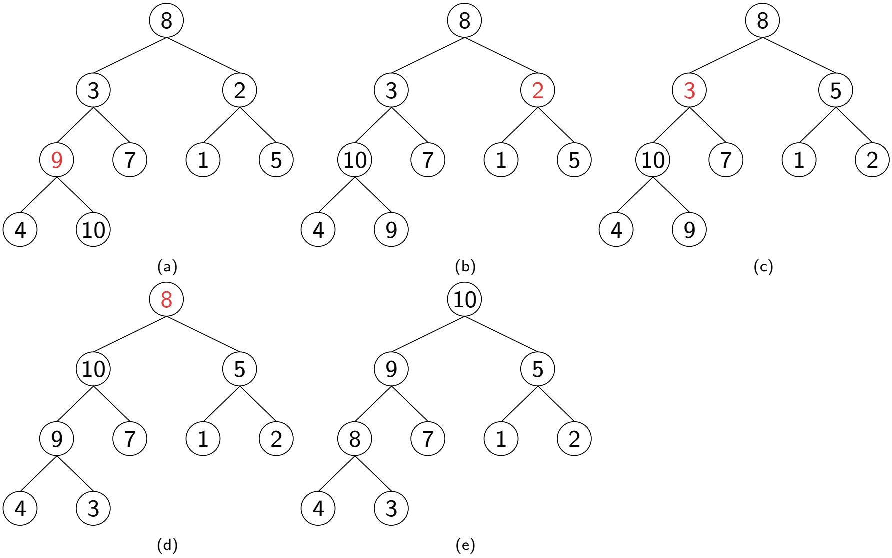
korektnost indukcí vzhledem k hloubce stromu

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

Build_Max_Heap(A)

$A.heap\_size \leftarrow A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
    Max_Heapify(A, i) **od**

BUILD_MAX_HEAP(A)



Build_Max_Heap — korektnost

invariant cyklu

Na začátku každé iterace **for** cyklu je každý z uzlů $i+1, i+2, \ldots, n$ kořenem haldy.

inicializace na začátku je $i = \lfloor n/2 \rfloor$, uzly $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots$,n jsou listy a jsou tedy kořeny (triviální) haldy

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

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

Build_Max_Heap — složitost

- pro strom s hloubkou h je složitost $\textsc{Max\_Heapify}$ $\mathcal{O}(h)$, t.j. je ohraničená funkcí $c \cdot h$ (c je konstanta)
- 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 nejvýše

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

při zjednodušování výrazu jsme využili rovnost

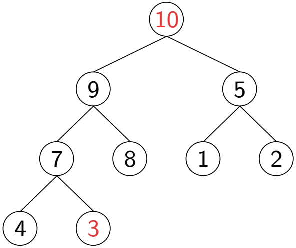
$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

ALGORITMUS ŘAZENÍ HALDOU, HEAPSORT

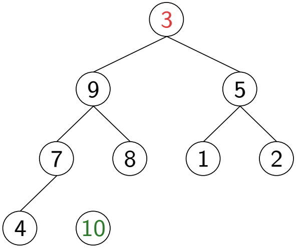
- použitím procedury Build_Max_Heap vybudujeme haldu nad polem $A[1 \ldots n]$
- maximální prvek pole A je uložený v kořeni $A[1]$ a proto ho můžeme přesunout na jeho finální pozici $A[n]$ (vyměníme prvky $A[1]$ a $A[n]$)
- prvek, který jsme přesunuli do kořene, může porušit vlastnost haldy a pro obnovení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

```
Heapsort(A)
Build_Max_Heap(A)
for i = A.length downto 2 do vyměň A[1] a A[i]
    A.heap_size ← A.heap_size − 1
    Max_Heapify(A, 1) od
```

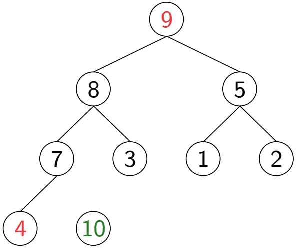
Heapsort



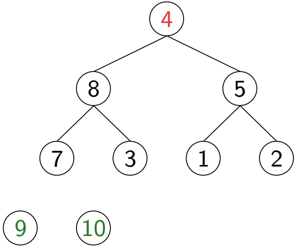
(a)



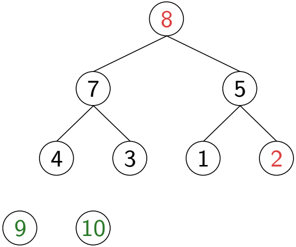
(b)



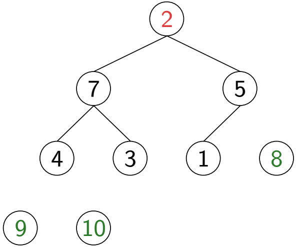
(c)



(d)



(e)



(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

optimalizace

- 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

varianty

- strom vyšší arity
- ve vrcholu stromu uložených několik klíčů
- haldy binomiální, Fibonacciho ...

ŘAZENÍ HALDOU

PRIORITNÍ FRONTA

PRIORITNÍ FRONTA

- datový typ pro reprezentaci množiny prvků, nak kterými je definováno uspořádání
- efektivní realizace 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 $k \geq x$
- **prioritní frontu implementujeme jako haldu**

alternativně můžeme definovat prioritní frontu vůči minimálnímu prvku a pro implementaci využít minimovou haldu

prioritní fronta — Maximum

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

$\textsc{Heap\_Maximum}(A)$

return $A[1]$

prioritní fronta — Extract_Max

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

$\textsc{Heap\_Extract\_Max}(A)$

if $A.heap\_size < 1$ **then return** prázdná fronta **fi**

$max \leftarrow A[1]$

$A[1] \leftarrow A[A.heap\_size]$

$A.heap\_size \leftarrow A.heap\_size - 1$

$\textsc{Max\_Heapify}(A, 1)$

return max

prioritní fronta — Heap_Increase_Key(A, i, 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 tak, že nový prvek posouváme směrem ke kořeni
- složitost operace je $\mathcal{O}(\log n)$

$\textsc{Heap\_Increase\_Key}(A, i, key)$

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

prioritní fronta — 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
- složitost operace je $\mathcal{O}(\log n)$

$\textsc{Max\_Heap\_Insert}(A, key)$

$A.heap\_size \leftarrow A.heap\_size + 1$
$A[A.heap\_size] \leftarrow -\infty$
$\textsc{Heap\_Increase\_Key}(A, A.heap\_size, key)$

ŘAZENÍ V LINEÁRNÍM ČASE

ŘAZENÍ V LINEÁRNÍM ČASE

COUNTING SORT

ŘAZENÍ POČÍTÁNÍM - COUNTING SORT

předpoklad

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

k není závislé na délce posloupnosti

Counting sort — idea

- vstupní posloupnost je uložena v $A[1 \dots n]$
- výstupní (seřazená) posloupnost je uložena v $B[1 \dots n]$
- pole $C[0 \dots k]$ se využívá v průběhu výpočtu

- pro každou hodnotu $i = 0, 1, \dots, k$ spočítáme, kolik je ve vstupní posloupnosti čísel i, výsledný počet uložíme do $C[i]$
- pro každou hodnotu $i = 0, 1, \dots, k$ 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

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

	1	2	3	4	5	6	7	8
A		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Counting_Sort(A, B, k)

```
//inicializace C[0 ... k]
for i = 0 to k do
    C[i] ← 0 od
for j = 1 to A.length do
    C[A[j]] ← C[A[j]] + 1 od
//C[i] obsahuje počet čísel rovných i
for i = 1 to k do
    C[i] ← C[i] + C[i − 1] od
//C[i] obsahuje počet čísel menších nebo rovných i
for j = A.length downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] − 1 od
```

Counting sort — 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 $\Theta(k + n)$

- algoritmu je stabilní, protože prvky se stejnou hodnotou se ve výstupní posloupnosti vyskytují ve stejném pořadí jako ve vstupní posloupnosti
- stabilita algoritmu Counting sort se využíva v algoritmu Radix sort

Counting 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 $\Theta(n+k)$

ŘAZENÍ V LINEÁRNÍM ČASE

RADIX SORT

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)

for $i = 1$ **to** d **do**
  použij stabilní řazení a seřaď položky podle $i$te číslice
od

Radix Sort — složitost

Danou posloupnost n čísel, z nichž každé má d číslic, přičemž číslice můžou nabývat k různých hodnot, seřadí algoritmus RADIX_SORT 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 a k jsou konstanty, tak časová složitost číslicového řazení je lineární

Radix Sort — řazení binárních čísel

- 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 $2^r - 1$
- pro řazení ve skupině použijeme Counting sort pro $k = 2^r - 1$

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 $\Theta(n + k)$.

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

$\lfloor b < \log n \rfloor$ pro $r = b$ celková složitost číslicového řazení $\Theta(n)$

$\lfloor b \geq \log n \rfloor$ 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 $\Theta(n)$

ŘAZENÍ V LINEÁRNÍM ČASE

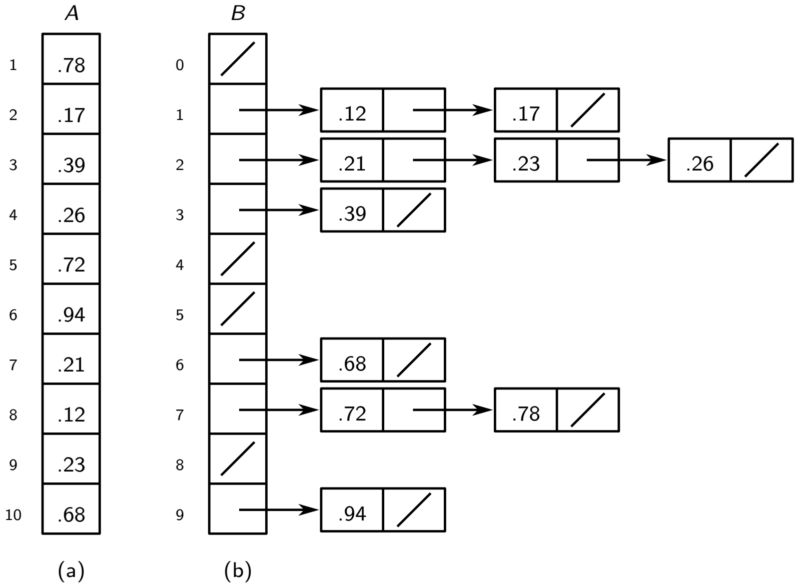
BUCKET SORT

PŘIHRÁDKOVÉ ŘAZENÍ - BUCKET SORT

předpoklad

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

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


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

//$B[0 \dots n-1]$ je nové pole
$n \leftarrow A.length$
for $i = 0$ **to** $n-1$ **do**
    $B[i] \leftarrow$ prázdny seznam **od**
for $i = 1$ **to** n **do**
    přidej $A[i]$ do seznamu $B[\lfloor n \cdot A[i] \rfloor]$ **od**
for $i = 0$ **to** $n-1$ **do**
    seřaď prvky seznamu $B[i]$ použitím řazení vkládáním **od**
spoj seznamy $B[0], B[1], \dots, B[n-1]$ 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 $\Theta(n)$

Část III

Datové struktury

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

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

$\textsc{Search}(S, k)$ pro množinu S a klíč k vrátí ukazatel x takový, že $x.key = k$ resp. $\textsc{Nil}$, když objekt s klíčem k není obsažen v množině S

$\textsc{Insert}(S, x)$ do množiny S vloží objekt s ukazatelem x

$\textsc{Delete}(S, x)$ z množiny S odstraní objekt s ukazatelem x

$\textsc{Maximum}(S)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je maximální

$\textsc{Minimum}(S)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je minimální

$\textsc{Successor}(S, x)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel na objekt, jehož klíč následuje bezprostředně za klíčem $x.key$, resp. hodnotu $\textsc{Nil}$ když klíč $x.key$ je maximální

$\textsc{Predecessor}(S, x)$ symetricky k $\textsc{Successor}$

VYHLEDÁVACÍ STROMY

VYHLEDÁVACÍ STROMY

MOTIVACE

PROBLÉM REZERVACÍ

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

- množina rezervací R
- požadavek t na může být potvrzen, 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: $k = 3$, aktuální čas je 20, $R = \{21, 26, 29, 36\}$

- rezervace 24 není validní (nemůže být potvrzena), protože je v kolizi s rezervací 26 z R
- rezervace 15 není validní, protože aktuální čas je 20
- rezervace 33 je validní

? datový typ 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?

potřebujeme současně efektivní vyhledávání i vkládání!

VYHLEDÁVACÍ STROMY

BINÁRNÍ VYHLEDÁVACÍ STROMY

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 (BVS)

- kořenový strom, v němž každý uzel má nejvýše dva následníky
- každý uzel stromu představuje jeden objekt, obsahující
  - klíč *key*
  - ukazatele *left*, *right* a *p* na levého syna, pravého syna a na otce; ukazatel má hodnotu *Nil* právě když uzel nemá příslušného syna, resp. otce
  - případné další data
- pro všechny uzly binárního vyhledávacího stromu platí

  jestliže x je uzel BVS a

  y je uzel v levém podstromu uzelu x, tak platí $y.key \leq x.key$

  y je uzel v pravém podstromu uzelu x, tak platí $y.key \geq x.key$

BVS — PROCHÁZENÍ STROMU

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

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

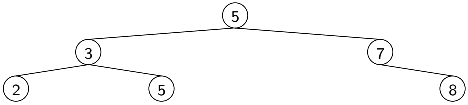
BVS — VÝPIS KLÍČŮ

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

preorder hodnotu klíče uloženého v kořeni vypíšeme **před** vypsáním klíčů uložených v jeho levém a pravém podstromě

postorder hodnotu klíče uloženého v kořeni vypíšeme **po** vypsání klíčů uložených v jeho levém a pravém podstromě



inorder (2 3 5 5 7 8), preorder (5 3 2 5 7 8), postorder (2 5 3 8 7 5)

Inorder_Tree_Walk(x)

```
if x ≠ Nil
   then Inorder_Tree_Walk(x.left)
        print x.key
        Inorder_Tree_Walk(x.right)
fi
```

- Inorder_Tree_Walk($T.root$) vypíše klíče uložené v BVS T v pořadí od nejmenšího po největší
- časová složitost je $\Theta(n)$, kde n je počet uzlů 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

$\textsc{Tree\_Search}(x, k)$

if $x = Nil \ \vee \ k = x.key$
    then return x **fi**
if $k < x.key$
    then return $\textsc{Tree\_Search}(x.left, k)$
     else return $\textsc{Tree\_Search}(x.right, k)$ **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

$\textsc{Tree\_Minimum}(x)$

while $x.\mathit{left} \neq \mathit{Nil}$ **do** $x \leftarrow x.\mathit{left}$ **od**
return x

$\textsc{Tree\_Maximum}(x)$

while $x.\mathit{right} \neq \mathit{Nil}$ **do** $x \leftarrow x.\mathit{right}$ **od**
return x

BVS — PŘEDCHŮDCE A NÁSLEDNÍK

- předpokládáme, že všechny klíče uložené v stromě jsou vzájemně různé
- 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)*

pro stromy, které můžou obsahovat uzly se stejnými klíči, se pojmy a operace definují analogicky

BVS — následník uzlu x

- jestliže pravý podstrom uzlu x je *neprázdný*, tak následníkem x je uzel jeho pravého podstromu s nejmenší klíčem
- jestliže pravý podstrom uzlu x 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 $y.key > x.key$ (x *patří do levého podstromu uzlu* y)

$\textsc{Tree\_Successor}(x)$

if $x.right \neq Nil$
    then return $\textsc{Tree\_Minimum}(x.right)$ **fi**
$y \leftarrow x.p$
while $y \neq Nil \ \wedge \ x = y.right$
        do $x \leftarrow y$
        $y \leftarrow y.p$ **od**
return y

BVS — PŘIDÁNÍ NOVÉHO UZLU

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

TREE_INSERT(T, z)

```
y ← Nil
x ← T.root
while x ≠ Nil do
        y ← x
        if z.key < x.key then x ← x.left
                          else x ← x.right fi
od
z.p ← y
if y = Nil then T.root ← z
           else if z.key < y.key then y.left ← z
                                 else y.right ← z fi
fi
```

BVS — ODSTRANĚNÍ UZLU

při odstraňování uzlu z, můžou nastat 3 případy

případ 1

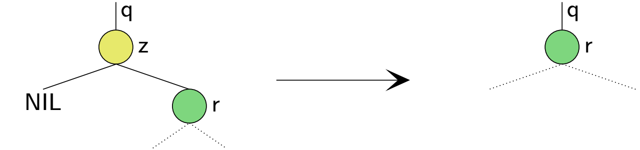
***z* nemá žádného syna**

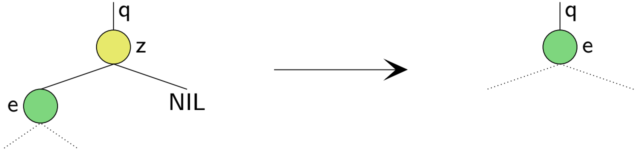
uzel odstraníme

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

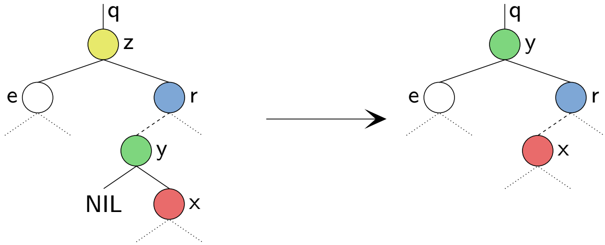




BVS — odstranění uzlu - případ 3

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 uzel s nejmenším klíčem v pravém podstromě uzlu z
- y nemá levého syna, proto ho můžeme přesunout na pozici z



- $\textsc{Transplant}$ 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

$\textsc{Transplant}(T, u, v)$

```
if u.p = Nil then T.root ← v
              else if u = u.p.left then u.p.left ← v
                                   else u.p.right ← v
                   fi
fi
if v ≠ Nil then v.p ← u.p fi
```

```
TREE_DELETE(T, z)
if z.left = Nil
   then TRANSPLANT(T, z, z.right)
   else if z.right = Nil
           then TRANSPLANT(T, z, z.left)
           else y ← TREE_MINIMUM(z.right)
                if y.p ≠ z then TRANSPLANT(T, y, y.right)
                                y.right ← z.right
                                y.right.p ← y
                fi
                TRANSPLANT(T, z, y)
                y.left ← z.left
                y.left.p ← y
        fi
fi
```

SLOŽITOST OPERACÍ NAD BVS

- 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

- vyvážený binární vyhledávací strom je BVS, jehož hloubka je logaritmická
  (*vůči počtu klíčů*)
- složitost operací nad vyváženým binárním vyhledávacím stromě je logaritmická

příklady vyvážených binárních vyhledávacích stromů

- 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*"
- otázka účelnosti 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

VYHLEDÁVACÍ STROMY

INTERVALOVÉ STROMY

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

$\textsc{Interval\_Insert}(T, x)$ do množiny intervalů T přidá objekt reprezentující interval x

$\textsc{Interval\_Delete}(T, x)$ z množiny intervalů T odstraní objekt reprezentující interval x

$\textsc{Interval\_Search}(T, i)$ 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
- uzel i má atributy $i.low$, $i.high$, a $i.max$
- jako klíč je použita hodnota $i.low$

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

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

intervalové stromy — přidání nového intervalu

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

intervalové stromy — 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 procházení cesty od uzlu y do kořene
- celková složitost operace odstranění intervalu zůstává asymptoticky stejná

intervalové stromy — vyhledávání intervalu

INTERVAL_SEARCH(T, i)

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

složitost

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

intervalové stromy — vyhledávání intervalu

INTERVAL_SEARCH(T, i)

$x \leftarrow T.root$
while $x \neq Nil$ $\wedge$ intervaly i a $(x.low, x.high)$ se nepřekrývají **do**
    if $x.left \neq Nil \wedge x.left.max \geq i.low$
        then $x \leftarrow x.left$
        else $x \leftarrow x.right$ **fi od**
return x

korektnost, případ 1 - ve vyhledávání postupujeme doprava

co když hledaný interval je v levém podstromu??

- předpokládejme, že 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 s žádným intervalem v levém podstromu

intervalové stromy — vyhledávání intervalu

INTERVAL_SEARCH(T, i)

$x \leftarrow T.root$
while $x \neq Nil \;\wedge$ intervaly i a $(x.low, x.high)$ se nepřekrývají **do**
  if $x.left \neq Nil \;\wedge\; x.left.max \geq i.low$
    then $x \leftarrow x.left$
    else $x \leftarrow x.right$ **fi od**
return x

korektnost, případ 2 - ve vyhledávání postupujeme doleva

co když hledaný interval je v pravém podstromu??

- předpokládejme, že žádný interval levého podstromu se nepřekrývá s intervalem i
- v levém podstromu leží interval $\langle c, d\rangle$ takový, že $d = x.left.max$
- i a $\langle c, d\rangle$ se nepřekrývají a tedy $i.high < c$
- pro každý $\langle a, b\rangle$ 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

ČERVENO ČERNÉ STROMY

ČERVENO ČERNÉ STROMY

ČERVENO ČERNÉ STROMY

ČERVENO ČERNÉ STROMY

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

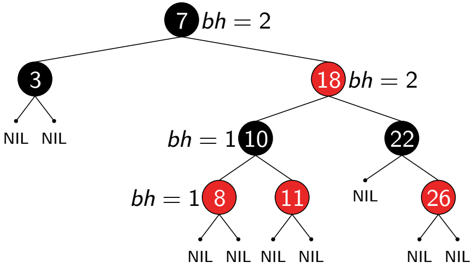
1. kořen stromu je černý
2. listy stromu nenesou žádnou hodnotu, tj. jsou označeny *Nil*, a mají černou barvu
3. když je uzel červený, tak jeho otec je černý
4. pro každý uzel x stromu platí, že všechny cesty z uzlu x do listů obsahují stejný počet černých uzlů

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!)



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

Každý podstrom s kořenem x má alespoň $2^{bh(x)} - 1$ vnitřních uzlů.

- důkaz indukcí k výšce h uzlu x
- pro $h = 0$ je x list; $bh(x) = 0$ a současně počet vnitřních uzlů podstromu s kořenem x je 0
- nechť x má výšku $h > 0$ 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

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

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 tvrzení plyne

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

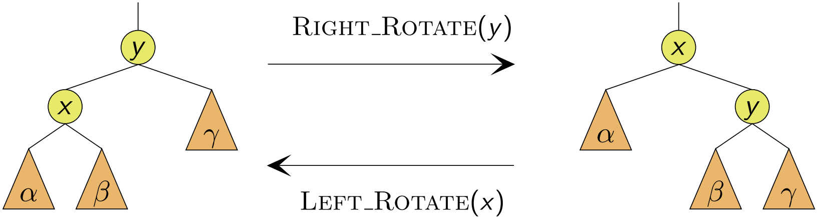
- po úpravě $\log_2(n+1) \geq h/2$, a tedy $h \leq 2\log_2(n+1)$

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 $\mathcal{O}(1)$

LEFT_ROTATE(T, x)

```
y ← x.right
x.right ← y.left
if y.left ≠ Nil
   then y.left.p ← x fi
y.p ← x.p
if x.p = Nil
   then T.root ← y
    else if x = x.p.left
             then x.p.left ← y
              else x.p.right ← y fi fi
y.left ← x
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
- vlastnost 4. (*stejný černý výška*) zůstává v platnosti
- může dojít k porušení vlastností 1. (*kořen stromu nemusí být černý*) a vlastnosti 3. (*otec červeného uzlu nemusí být černý*)
- po vložení uzlu vykonáme korekce, které obnoví platnost všech vlastností

```
RB_INSERT(T, a)
TREE_INSERT(T, a)
a.color ← red
while a ≠ T.root ∧ a.p.color = red
      do if a.p = a.p.p.left
            then d ← a.p.p.right
                  if d.color = red
                     then  případ 1
                     else if a = a.p.right
                             then  případ 2
                             else  případ 3
                          fi
                  fi
            else  stejně jako THEN se záměnou left a right
         fi
od
T.root.color ← black
```

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

$a.p.color \leftarrow black$

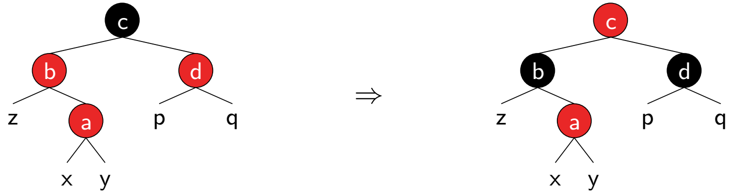
$a.p.p.color \leftarrow red$

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

přidání nového uzlu - korekce - případ 1

- uzel *a* je červený
- jeho otec *b* je červený a je levým synem svého otce (*pravý syn symetricky*)
- 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

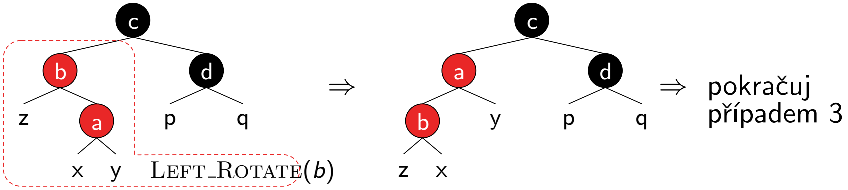


stromy z, x, y, p, q mají černý kořen a všechny mají stejnou černou výšku

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**

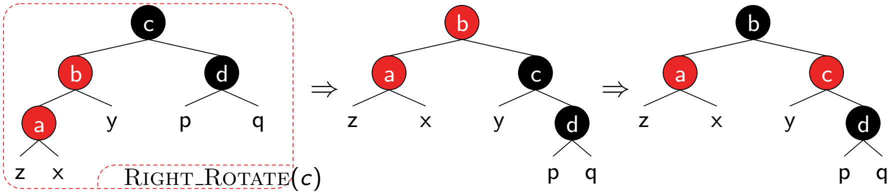
- proveď levou rotaci kolem otce *b* uzlu *a*
- pokračuj na případ 3



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

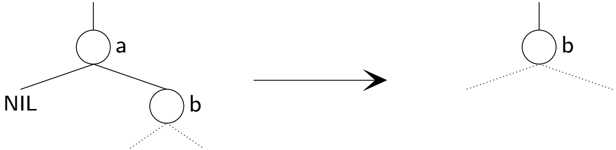
- uzel x ze stromu odstraníme stejným postupem jako z binárního vyhledávací stromu
- v případě, že odstraněný uzel měl červenou barvu, vlastnosti stromu zůstávají zachované
- v případě, že měl černou barvu, může dojít k porušení vlastnosti 4 (stejná černá výška)
- černou barvu z odstraněného uzlu přesouváme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

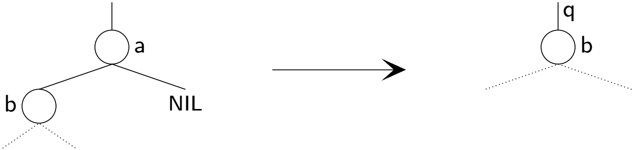
odstranění uzlu *a* - případy 1 a 2

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



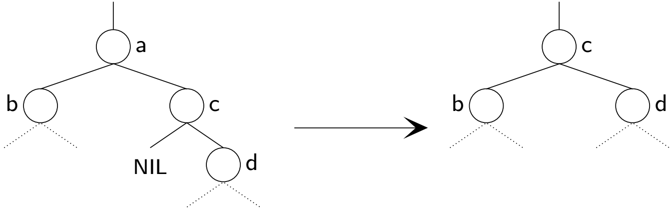


odstranění uzlu *a* - případ 3

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

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

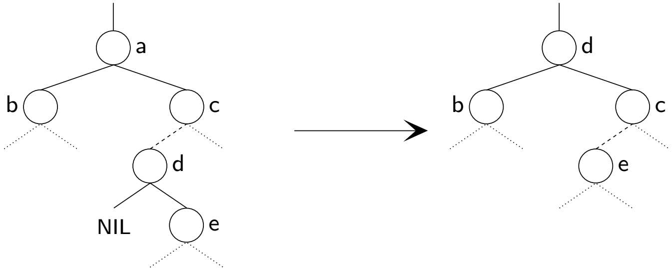
- po přesunu obarvíme *c* barvou uzlu *a*
- jestliže *c* měl původně černou barvu, tak černou barvu dostane jeho pravý syn, tj. syn má dvě barvy (červenou a černou anebo černou a černou)
- problém dvou barev vyřešíme při korekci



odstranění uzlu *a* - případ 4

a **má dva syny**, následník uzlu *a* **není** jeho pravým 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 *d* barvou uzlu *a*
- jestliže *d* 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



KOREKCE DVOU BAREV

bazální případ

uzel *a* má červenou a černou barvou

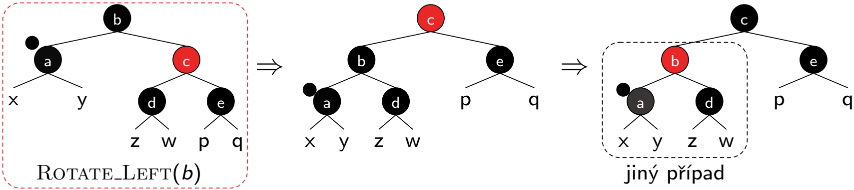
- obarvi uzel *a* černou barvu

korekce dvou barev - případ 1

uzel *a* má dvě černé 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



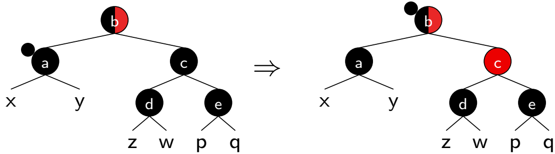
stromy x, y, z, w, p, q neporušují žádnou vlastnost červeno černého stromu

korekce dvou barev - případ 2

uzel a má dvě černé 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

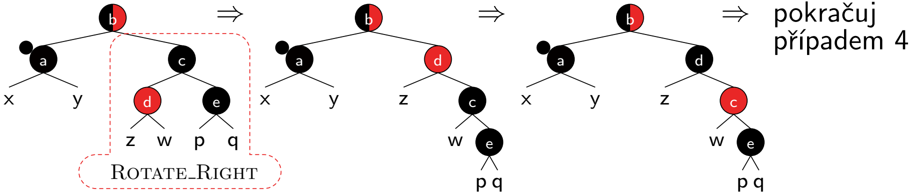


korekce dvou barev - případ 3

uzel *a* má dvě černé 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*, t.j. mezi uzly *d* a *c*
- pokračuj případem 4

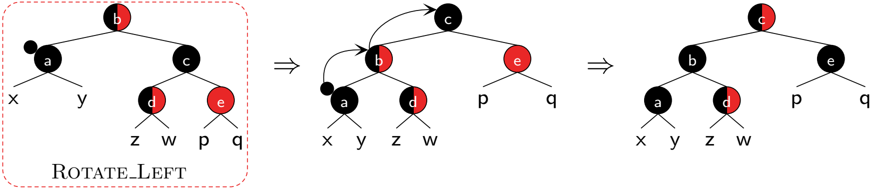


korekce dvou barev - případ 4

uzel *a* má dvě černé barvy

bratr *c* uzlu *a* má černou barvu, jeho pravý syn *e* má červenou barvu

- proveď levou rotaci kolem otce *b* uzlu *a*
- uzel *c* dostane barvu uzlu *b*
- uzel *b* dostane černou barvu z uzlu *a*
- uzel *e* změní barvu na černou



ČERVENO ČERNÉ STROMY

RANK PRVKU

POŘADÍ (RANK) PRVKU

problém ranku

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

jak efektivně určit rank prvku?

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

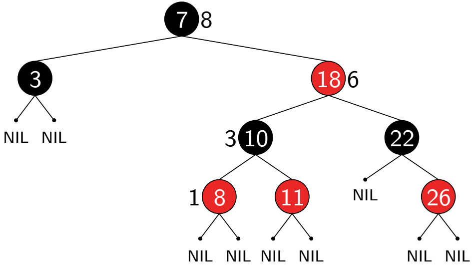
- efektivní implementaci standardních operací nad červeno černým stromem
- efektivní implementaci operace $\mathrm{RB\_S{\small ELECT}}(x, i)$, která najde i-ty nejmenší klíč v podstromě s kořenem x
- efektivní implementaci operace $\mathrm{RB\_R{\small ANK}}(T.x)$, která určí rank klíče uloženého v 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 $x.size$, který udává 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)
  r ← x.left.size + 1
  if i = r then return x
         else if i < r then return RB_Select(x.left, i)
                       else return RB_Select(x.right, i − r) fi fi
```

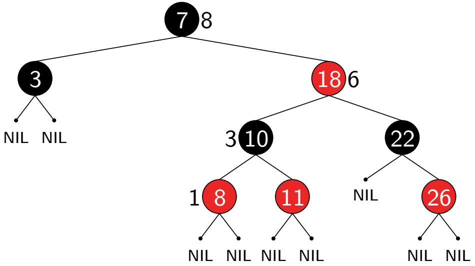
korektnost

- 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 — složitost

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

RB_Rank(T, x)

```
r ← x.left.size + 1
y ← x
while y ≠ T.root
      do if y = y.p.right
            then r ← r + y.p.left.size + 1 fi
      y ← y.p od
return r
```

určení ranku daného prvku — 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 $x = y$

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 $x.key$)

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

určení ranku daného prvku — složitost

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

PŘIDÁNÍ NOVÉHO UZLU

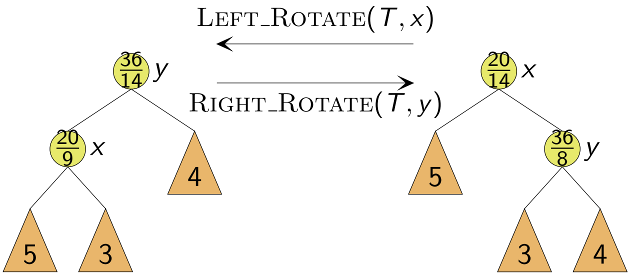
přidání uzlu postupujeme od kořene do listu, kde vytvoříme nový uzel, přitom se změní (o 1) pouze velikost podstromů těch uzlů, kterými procházíme

korekce stromu změna barvy uzlu nemění velikost podstromu

při rotaci se může změnit velikost podstromů

proceduru $\textsc{Left\_Rotate}$ doplníme o příkazy

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



ODSTRANĚNÍ UZLU

odstranění uzlu 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

B-STROMY

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

- v databázových systémech a aplikacích, kde objem zpracovávaných dat není možné uchovávat v operační paměti
- počet klíčů uložených v uzlu se může pohybovat od jednotek po tisíce; cílem je minimalizovat počet přístupů na disk
- existují různé varianty, podrobněji viz např. PV062
- Bayer, McCreight 1972

STUPEŇ B STROMU

minimální stupeň stromu je 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
- 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 B stromu 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, 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 $\log_t t^h \leq \log_t \frac{n+1}{2}$ $\square$

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

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

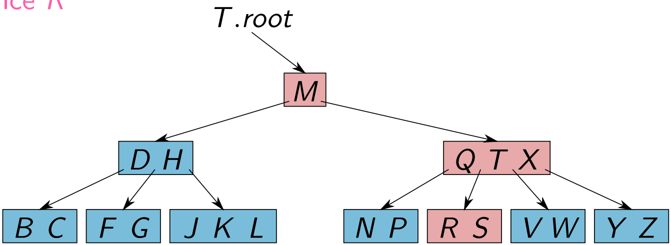
OPERACE NAD B STROMEM

- 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
- kořen B stromu je vždy uložený v operační paměti
- 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 $T.root$ 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



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)

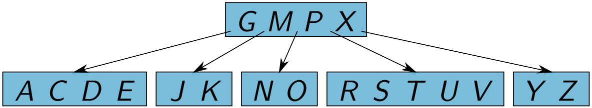
```
x ← ALLOCATE_NODE()
x.leaf ← true
x.n ← 0
DISK_WRITE(x)
T.root ← x
```

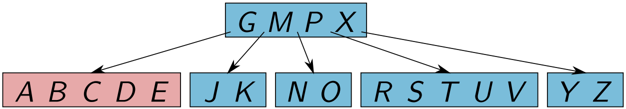
celková složitost operace $\mathcal{O}(1)$

PŘIDÁNÍ KLÍČE

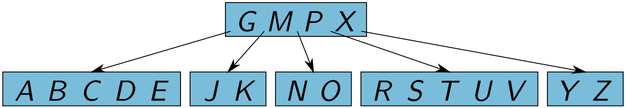
- podobně jako u BVS hledáme list, do kterého uložíme nový klíč
- nemůžeme vytvořit nový list (jako u BVS), protože bychom porušili vlastnost minimálního počtu klíčů v uzlu
- klíč vložíme do existujícího listu
- když vložením klíče dojde k porušení vlastnosti maximálního počtu klíčů, tak list rozdělíme 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

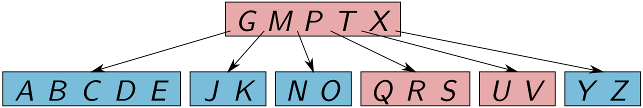
přidání klíče *B* do listu, který není plný, minimální stupeň stromu je 3



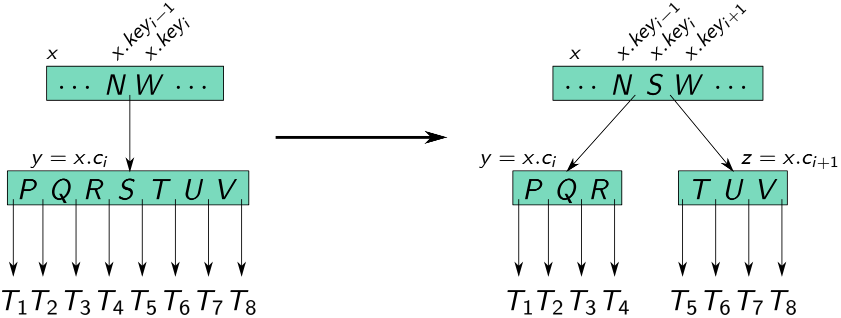


přidání klíče Q do plného listu, minimální stupeň stromu je 3





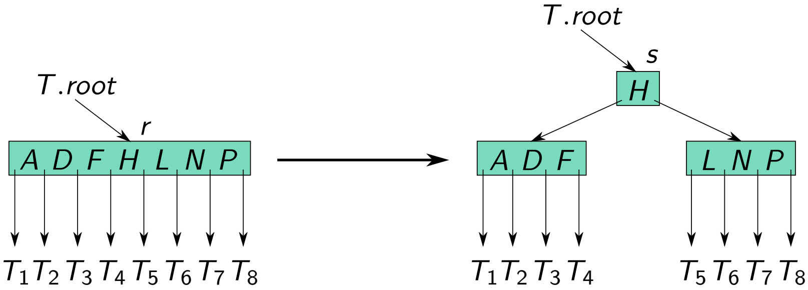
rozdělení uzlu - schéma, minimální stupeň stromu je 4



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

```
B-TREE_SPLIT(x, i)
  z ← ALLOCATE_NODE()
  y ← x.c_i
  z.leaf ← y.leaf
  z.n ← t − 1
  for j = 1 to t − 1 do z.key_j ← y.key_{j+t} od
  if ¬y.leaf then for j = 1 to t do z.c_j ← y.c_{j+t} od fi
  y.n ← t − 1
  for j = x.n + 1 downto i + 1 do x.c_{j+1} ← x.c_j od
  x.c_{i+1} ← z
  for j = x.n downto i do x.key_{j+1} ← x.key_j od
  x.key_i ← y.key_t
  x.n ← x.n + 1
  DISK_WRITE(y)
  DISK_WRITE(z)
  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 $\textsc{Disk\_Write}$ a $\textsc{Disk\_Read}$ je $\mathcal{O}(1)$

PŘIDÁNÍ KLÍČE - OPTIMALIZACE

základní varianta

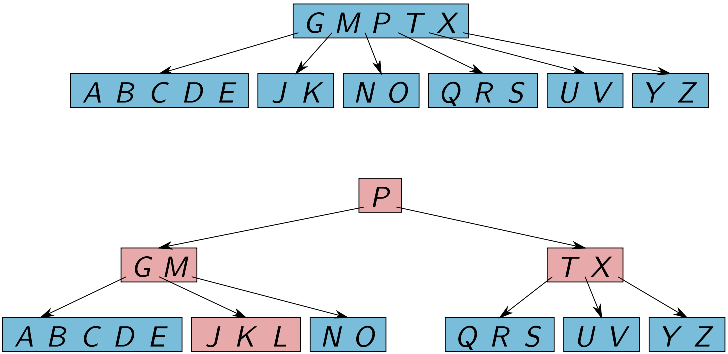
- rozdělení uzlu způsobí navýšení počtu následníků předchůdce rozdělovaného uzlu
- pokud se tím poruší vlastnost maximálního počtu následníků, tak musíme (rekurzivně) rozdělit i předchůdce
- proces rozdělování se v nejhorším případě zastaví až v kořeni stromu

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
- 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



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

```
B-Tree_Insert_Nonfull(x, k)
i ← x.n
if x.leaf
   then while i ≥ 1 ∧ x.key_i > k
              do x.key_{i+1} ← x.key_i
                 i ← i − 1 od
        x.key_{i+1} ← k
        x.n ← x.n + 1
        Disk_Write(x)
   else while i ≥ 1 ∧ x.key_i > k do i ← i − 1 od
        i ← i + 1
        Disk_Read(x.c_i)
        if x.c_i.n = 2t − 1 then B-Tree_Split(x, i)
                                 if x.key_i < k then i ← i + 1 fi fi
        B-Tree_Insert_Nonfull(x.c_i, k)
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

- 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 (resp. předchůdce) odstraníme z listu ve kterém se původně nacházel

samotné mazání klíče se vždy realizuje v listu

odstranění klíče k z listu x

list x obsahuje alespoň t klíčů anebo je kořenem stromu

- klíč k odstraníme

odstranění klíče k z listu x

list x není kořenem a obsahuje přesně $t-1$ klíčů

- po odstranění klíče k klesne počet klíčů v x pod minimum $t-1$
- vezmi toho bratra y uzlu x, který má více klíčů
- vytvoř seznam obsahující klíče z uzlů x a y a navíc ten klíč z otce p uzlu x, který tvoří hranici mezi x a y
- jestliže seznam obsahuje alespoň $2t-1$ klíčů
  - seznam rozdělíme na 3 části: *Left*, *Middle* a *Right*, kde *Middle* je medián seznamu, *Left* jsou 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* vložíme do uzlu x, klíče *Right* do y • uzly x a y mají alespoň $t-1$ klíčů, počet klíčů v uzlu p zůstal nezměněný
- jestliže seznam obsahuje $2t-2$ klíčů
  - uzly x a y nahradíme jediným uzlem obsahujícím všechny klíče seznamu
  - nový uzel 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 - OPTIMALIZACE

motivace

- po odstranění klíče z listu může klesnout počet klíčů listu pod minimální hranici $t - 1$
- upravíme strom tak, aby každý uzel obsahoval alespoň $t - 1$ klíčů
- může nastat situace, že při úpravě stromu musíme projít od listu až ke kořeni stromu (*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

optimalizace

- při vyhledávání klíče k, který má být odstraněn, postupujeme od kořene směrem dolů
- vždy, když procházíme přes uzel, který má přesně $t - 1$ klíčů, tak uděláme takovou korekci, která zvýší počet klíčů v uzlu na t

optimální odstranění klíče k - pravidla

postupně přecházíme uzly stromu od kořene; nechť x je aktuální uzel

případ 1 - x je list

- jestliže strom neobsahuje klíč k tak ukonči výpočet
- jestliže x obsahuje klíč k, odstraň k z listu x
- jestliže x obsahuje předchůdce resp. následníka k' klíče k, tak nahraď klíč k klíčem k' a odstraň k' z listu x

případ 2 - x je vnitřní uzel a obsahuje k

- zapamatuj si uzel x; po dosažení listu klíč k nahradí jeho následník nebo předchůdce
- nechť y (z) je syn uzlu x takový, že v podstromu s kořenem y (z) leží předchůdce (následník) klíče k
- 2a jestliže y resp. z má alespoň t klíčů, tak aktuální uzel se změní na y resp. z
- 2b v opačném případě přesuň do uzlu y klíč k a všechny klíče z uzlu z, aktuální uzel se změní na y

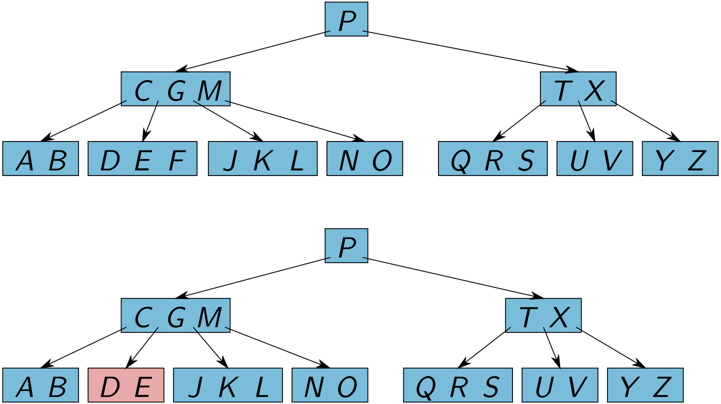
optimální odstranění klíče k - pravidla, pokračování

případ 3 - x je vnitřní uzel a neobsahuje k

- nechť y je syn uzlu x takový, že k musí být v podstromu s kořenem y
- 3a jestliže y obsahuje $t-1$ klíčů a jeho pravý nebo levý bratr obsahuje alespoň t klíčů, tak zvyš počet klíčů v y a to tak, že přesuneš klíč z x do y, přesuneš klíč z bratra do y a přesuneš příslušný ukazatel na následníka z bratra do y
- 3b jestliže y i jeho jeho pravý a levý bratr obsahují jen $t-1$ klíčů, tak přesuň do y jeden klíč z x a všechny klíče z jednoho z bratrů
- aktuální uzel se změní na y

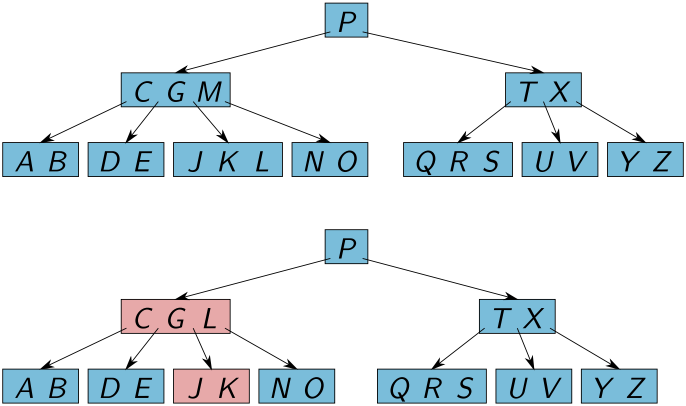
odstranění klíče F – případ 1

minimální stupeň stromu je 3



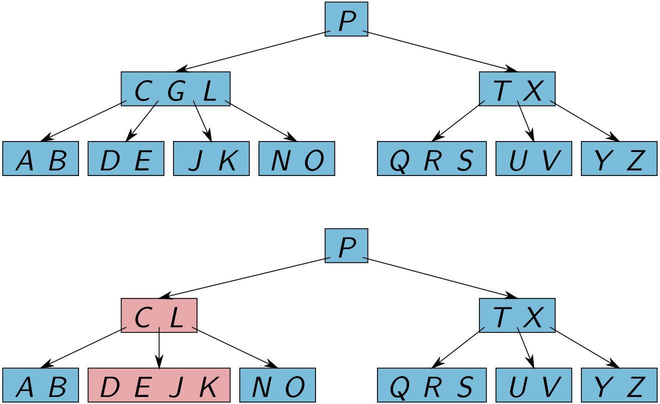
odstranění klíče M – případ 2a

minimální stupeň stromu je 3



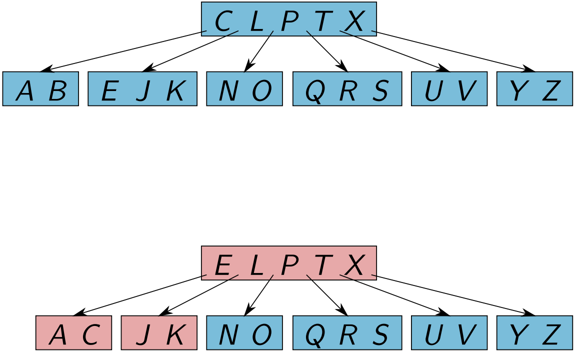
odstranění klíče G – případ 2b

minimální stupeň stromu je 3



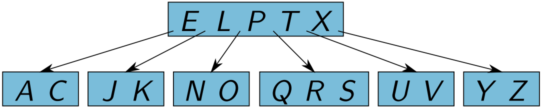
odstranění klíče B – případ 3a

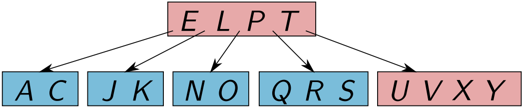
minimální stupeň stromu je 3



odstranění klíče Z – případ 3b

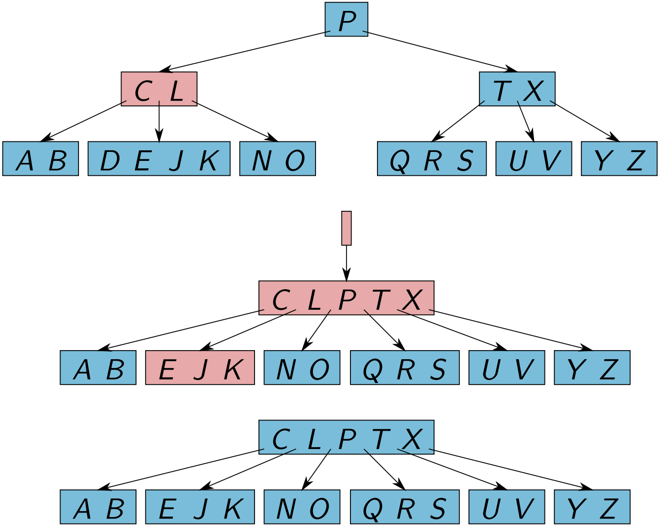
minimální stupeň stromu je 3





odstranění klíče D – případ 3b

minimální stupeň stromu je 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 2a, 2b)
- pro každý uzel se vykoná nanejvýš jedna operace $\textsc{Disk\_Write}$ a jedna operace $\textsc{Disk\_Read}$
- celková složitost operace 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

HAŠOVANÍ

SLOVNÍK

dynamický datový typ pro reprezentaci množiny objektů s operacemi

$\textsc{Insert}(S, x)$ 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 klíč, které se dají vzájemně porovnávat; při použití vyváženého stromu je složitost operací $\mathcal{O}(\log n)$

cíl: složitost všech operací

v nejhorším případě $\Theta(n)$

v očekávaném případě $\mathcal{O}(1)$

HAŠOVANÍ

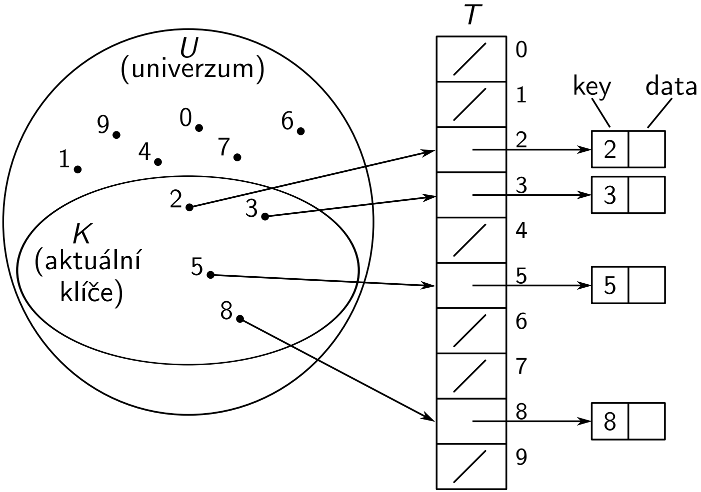
PŘÍMÉ ADRESOVÁNÍ

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 ($\textsc{Nil}$)
- 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

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)*

výběr hašovací funkce

- minimalizovat počet kolizí
- efektivní výpočet funkce

VLASTNOSTI HAŠOVACÍCH FUNKCÍ

problém 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 ze zvolené množiny hašovacích funkcí

složitost volba množiny hašovacích funkcí a způsob výběru hašovací funkce určují složitost (v nejhorším i očekávaném případě) jednotlivých operací množina $\mathcal{H}$ hašovacích funkcí z U do $\{0, 1, \ldots, m-1\}$ je

uniformní když pro uniformně vybranou funkci z $\mathcal{H}$ a každý klíč je pravděpodobnost zahašování klíče na každou z pozic tabulky stejná,

$$Pr_{h \in \mathcal{H}}[h(x) = i] = \frac{1}{m} \text{ pro každý klíč } x \text{ a pozici } i$$

univerzální když pro každou dvojici klíčů je pravděp. kolize co nejmenší

$$Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m} \text{ pro každou dvojici klíčů } x \neq y$$

téměř univerzální

$$Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{2}{m} \text{ pro každou dvojici klíčů } x \neq y$$

k-**uniformní** když pro každých k vzájemně různých klíčů $x_1 \ldots, x_k$ a hodnot $i_1, \ldots, i_k$ je pravděpodobnost kolize stejná

$$Pr_{h \in \mathcal{H}}[\bigwedge_{j=1}^{k} h(x_j) = i_j] = \frac{1}{m^k}$$

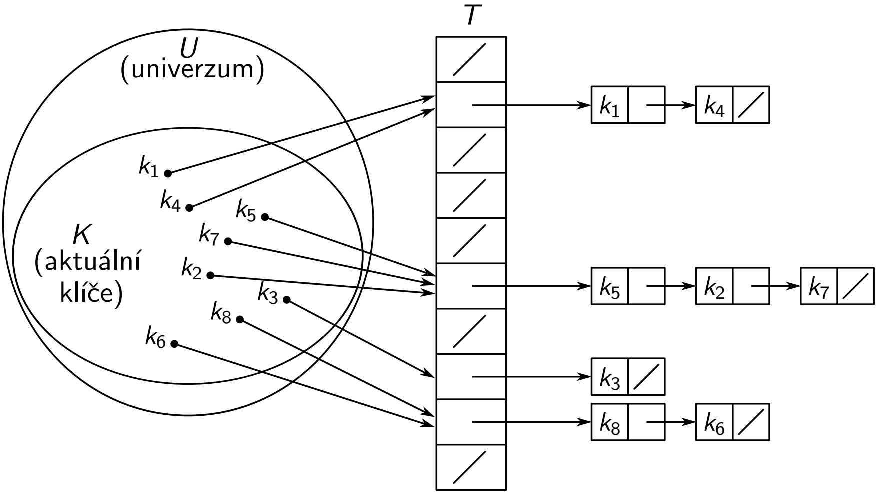
HAŠOVANÍ

ZŘETĚZENÉ HAŠOVÁNÍ

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

nechť $l(x)$ označuje délku seznamu $T[h(x)]$ klíčů zahašovaných na pozici $h(x)$

složitost v nejhorším případě

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

SEARCH konstantní složitost výpočtu $h(x)$ plus konstantní složitost za každý prvek seznamu $T[h(x)]$, celkově $\mathcal{O}(1 + l(x))$

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

složitost v očekávaném případě

záleží od výběru hašovací funkce

zřetězené hašování — očekávaná složitost

- je úměrná očekávaná délce $E[l(x)]$ seznamu $T[h(x)]$
- pro každou dvojici klíčů x, y nechť $C_{x,y}$ je náhodní proměnná, $C_{x,y} = 1$ když $h(x) = h(y)$ a $C_{x,y} = 0$ když $h(x) \neq h(y)$
- délka seznamu je rovna počtu kolizí, $l(x) = \sum_{y \in T} C_{x,y}$
- když h je vybraná uniformně z univerzální množiny hašovacích funkcí, tak

$$E[C_{x,y}] = Pr[C_{x,y} = 1] = \begin{cases} 1 & \text{pro } x = y \\ 1/m & \text{jinak} \end{cases}$$

$$E[l(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

- hodnotu n/m nazýváme faktor naplnění, označujeme α
- pro *téměř univerzální* funkce je $E[l(x)] \leq 2\alpha$
- místo seznamu můžeme použít jinou datovou strukturu, např. vyvážený binární vyhledávací strom

HAŠOVANÍ

PŘÍKLADY HAŠOVACÍCH FUNKCÍ

METODA DĚLENÍ

předpoklad: klíčem je číslo

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

výhody

- rychlost

nevýhody — závislost na volbě 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

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

METODA BINÁRNÍHO NÁSOBENÍ

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

pro zvolenou konstantu A, $0 < A < 1$,

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

Množina funkcií h_A,pro $0 < A < 1$, je téměr univerzální.

metoda binárního násobení — postup výpočtu hodnoty $h_A(k)$ $= \lfloor m\ (k\ A \mod 1) \rfloor$

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

jestliže 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
- protože $m = 2^p$, 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

- $w = 5, m = 8, w = 3$
- hašujeme klíč $k = 21$
- vybíráme konstantu $0 < A < 1$ tvaru $s/2^w$, vybereme $A = 13/32$

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
- $h_A(k) = 4$

METODA NÁSOBENÍ

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

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

- množina $\mathcal{H} = \{h_{ab} \colon a \in \{1, 2, \dots p-1\}, b \in \{0, 1, \dots p-1\}\}$ je univerzální množinou hašovacích funkcí

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*

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 $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0)$

HAŠOVANÍ

OTEVŘENÁ ADRESACE

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 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 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

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 použití uniformních hašovacích funkcí.

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 použití uniformních hašovacích funkcí.

HAŠOVANÍ

KUKAČČÍ HAŠOVÁNÍ

KUKAČČÍ HAŠOVÁNÍ (CUCKOO HASHING)

- pro hašování se používají dvě tabulky velkosti m a dvě hašovací funkce $h_1, h_2 : U \longrightarrow \{0, 1, \dots m-1\}$
- 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

HAŠOVANÍ

DOKONALÉ HAŠOVÁNÍ

DOKONALÉ HAŠOVÁNÍ (PERFECT HASHING)

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

první úroveň

- zřetězené hašování
- velikost tabulky je lineární vůči počtu klíčů

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

předpokladem pro dosažení konstantní složitosti je vhodný výběr hašovacích funkcí!

Část IV

Grafové algoritmy

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

GRAF A JEHO REPREZENTACE

graf $G = (V, E)$

- orientovaný / neorientovaný
- ohodnocené hrany / vrcholy
- jednoduché / násobné hrany

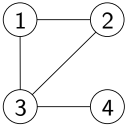
reprezentace grafu

- seznam následníků
- matice sousednosti

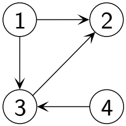
složitost grafových algoritmů je funkcí počtu vrcholů a hran

používáme zjednodušenou notaci, např. $\mathcal{O}(V + E)$ resp. $\mathcal{O}(n + m)$

MATICE SOUSEDNOSTI



$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$



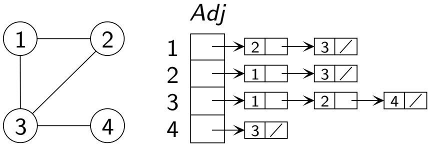
$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

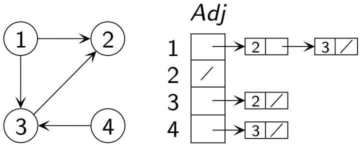
- matice $A = (a_{ij})$ rozměrů $|V| \times |V|$, kde

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

- 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 $|V|$, seznam pro uložení všech následníků vrchola
- 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 seznamu jinou datovou strukturu podporující vyhledávaní, vkládání a odebíraní, např. vyhledávací strom, hašovací tabulka

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 GRAFŮ A GRAFOVÁ SOUVISLOST

PRŮZKUM GRAFU

PRŮZKUM GRAFU

pro daný graf G a vrchol s 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)*

základní způsoby průzkumu grafu

- do šířky
- do hloubky

jaké další informace o grafu zjistíme v průběhu průzkumu?

PRŮZKUM GRAFU DO HLOUBKY

Theseus si před bludištěm uváže jeden konec nitě na strom a vstoupí dovnitř. Na první křižovatce (vrcholu) 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 vrátí se do výchozího vrcholu.

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 zásobníku. Namotávání nitě při návratu zpět odpovídá odebrání vrcholu ze zásobníku.

PRŮZKUM GRAFU DO ŠÍŘKY

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.

implementace

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

Průzkum grafu do šířky a do hloubky se liší pouze použitím fronty a zásobníku.

NE

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

PRŮZKUM DO ŠÍŘKY

PRŮZKUM DO ŠÍŘKY - STRATEGIE

cílem je prozkoumat všechny vrcholy dosažitelné z daného vrcholu s

- postupujeme od iniciálního vrcholu s po *vrstvách*
- $L_0 = \{s\}$
- $L_1 =$ všechny vrcholy, do kterých vede hrana z s
- $L_2 =$ všechny vrcholy, které nepatří do L_0 ani do L_1 a vede do nich hrana z vrcholu patřícího do L_1
- $L_{i+1} =$ všechny vrcholy, které nepatří do žádné z předcházejících úrovní a vede do nich hrana z vrcholu patřícího do L_i

implementace

- použití fronty
- atribut, který indikuje, zda vrchol již byl objeven (=vložen do fronty)

BFS(G, s)

```
foreach u ∈ V \ {s} do u.visited ← false od
s.visited ← true
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅ do
        u ← Dequeue(Q)
        foreach v ∈ Adj[u] do
            if not v.visited
                then v.visited ← true
                        Enqueue(Q, v) fi
        od
od
```

PRŮZKUM DO ŠÍŘKY - 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 $\mathcal{O}(V + E)$

PRŮZKUM DO ŠÍŘKY - ATRIBUTY VRCHOLU V

$v.color$

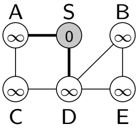
- v průběhu výpočtu je vrchol postupně objeven (je zařazen do fronty) a prozkoumán (všechny sousedící vrcholy jsou objeveny)
- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán
- vrchol má **šedivou** barvu právě když je dosažitelný z iniciálního vrcholu,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

$v.\pi$

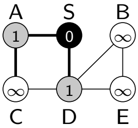
- vrchol, ze kterého byl vrchol v objeven

$v.d$

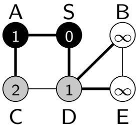
- délka (počet hran) cesty z s do v, na které byl v objeven



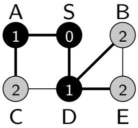
Q: S



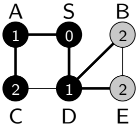
Q: AD



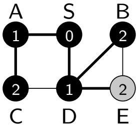
Q: DC



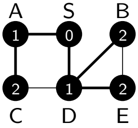
Q: CBE



Q: BE



Q: E



Q: $\emptyset$

BFS(G, s)

```
foreach u ∈ V \ {s}
   do u.color ← white;   u.d ← ∞;   u.π ← Nil od
s.color ← gray;   s.d ← 0;   s.π ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅ do
        u ← Dequeue(Q)
        foreach v ∈ Adj[u] do
           if v.color = white
              then v.color ← gray
                   v.d ← u.d + 1
                   v.π ← u
                   Enqueue(Q, v) fi
        od
        u.color ← black
od
```

BFS A DÉLKA NEJKRATŠÍ CESTY

Délka nejkratší cesty z s do v v neohodnoceném grafu, 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.

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

důkaz - viz Dijkstrův algoritmus

BFS STROM A NEJKRATŠÍ CESTY

algoritmus BFS definuje přes atributy π graf předchůdců (BFS strom)

pro graf $G = (V, E)$ a iniciální vrchol s je graf předchůdců $G_\pi = (V_\pi, E_\pi)$ definovaný předpisem

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

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

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

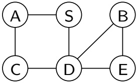
BFS STROM A HRANY GRAFU

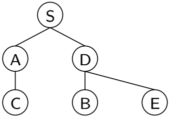
Nechť G je orientovaný graf a s jeho vrchol. Pak po provedení BFS průzkumu grafu G z vrcholu s pro každou hranu (u, v) grafu platí

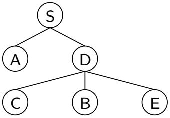
- $v.d = u.d + 1$ a hrana patří do BFS stromu
- $v.d = u.d + 1$ a hrana nepatří do BFS stromu
- $v.d = u.d$
- $v.d < u.d$

Pro hrany neorientovaného grafu platí některá z prvních tří možností.

BFS strom grafu není určen jednoznačně, závisí od pořadí, ve kterém zkoumáme následníky vrcholu







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 vrchol s nejmenší prioritou
- 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 prioritou
- 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

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

BIPARTITNÍ GRAFY

BIPARTITNÍ GRAFY

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, ...

BIPARTITNÍ GRAFY

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, . . .

Bipartitní graf neobsahuje cyklus liché délky.

TESTOVÁNÍ BIPARTITNOSTI S VYUŽITÍM BFS

- 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

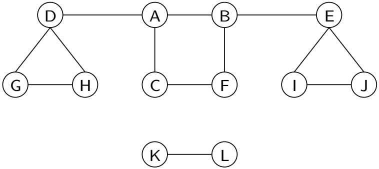
existují dva vrcholy spojeny hranou a patřící do stejné vrstvy

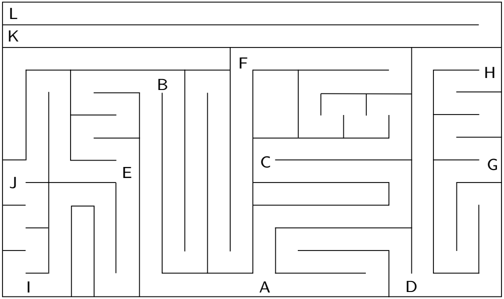
- nechť u, v jsou vrcholy takové, že $u, v \in L_i$ a $\{u, v\} \in E$
- nechť y je nejmenší společný předchůdce vrcholů u, v v BFS stromu
- cesta z y do u, hrana $\{u, v\}$ a cesta z v do y tvoří cyklus, jehož délka je lichá, protože cesty z y do u a z v do y mají stejnou délku
- graf není bipartitní

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

PRŮZKUM DO HLOUBKY

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 DO HLOUBKY - ATRIBUTY VRCHOLU V

v.color

- v průběhu výpočtu je vrchol postupně objeven (*je vložen do zásobníku*) a prozkoumán *(všechny sousedící vrcholy jsou objeveny)*
- vrchol má **černou** barvu právě když je dosažitelný z iniciálního vrcholu a byl již prozkoumán, tj. byly prozkoumáni všichni následníci 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.π

- vrchol, ze kterého byl vrchol v objeven

v.d

- značka udávající čas první návštěvy (objevení) vrcholu *(discovery time)*

v.f

- značka udávající čas ukončení průzkumu vrcholu *(finishing time)*

DFS(G)

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

Dfs_Visit(G, u)

```
time ← time + 1
u.d ← time
u.color ← gray
foreach v ∈ Adj[u] do
    if v.color = white then v.π ← u
                            Dfs_Visit(G, v) fi od
u.color ← black
time ← time + 1
u.f ← time
```

PRŮZKUM DO HLOUBKY - 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 $\mathcal{O}(V + E)$

iterativní implementace

```
DFS_Iterative_Visit(G, u)
S ← ∅
S.push(u)
time ← time + 1; u.d ← time
u.color ← gray
while S ≠ ∅ do
        u ← S.pop()
        if existuje hrana (u, v) taková, že v.color = white
          then S.push(u)
               S.push(v)
               v.color ← gray
               v.π ← u
               time ← time + 1; v.d ← time
          else u.color ← black
               time ← time + 1; u.f ← time fi
od
```

DFS STROM

- analogicky jako u BFS definují atributy $.\pi$ graf předchůdců G_π
- $G_\pi = (V, E_\pi)$
  $E_\pi = \{(v.\pi, v) \mid v \in V \text{ a } v.\pi \neq Nil\}$
- protože prohledáváme celý graf, který nemusí být nutně souvislý, graf předchůdců je DFS les, který se skládá z DFS stromů

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 $.f$ v klesajícím pořadí

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

vlastnosti časových značek

dosažitelnost

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 je dosažitelný z u právě když v čase $u.d$ existuje cesta z u do v obsahující jenom bílé vrcholy

vlastnosti časových značek — 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, nebo smyčka

při průzkumu hrany je vrchol v šedivý

$v.d \leq u.d < u.f \leq v.f$

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

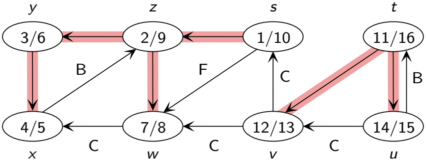
$u.d < v.d < v.f < u.f$

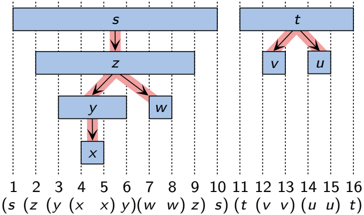
příčná hrana (*cross edge*) všechny ostatní hrany

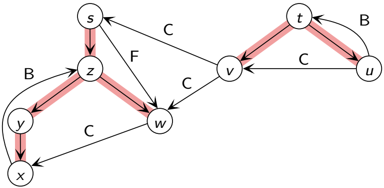
při průzkumu hrany je vrchol v černý

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

všechny hrany v neorientovaném grafu jsou buď stromové anebo zpětné







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

APLIKACE A ALGORITMY VYUŽÍVAJÍCÍ DFS

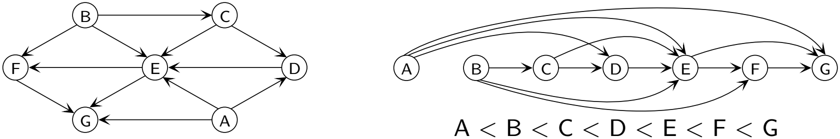
- topologické uspořádání
- komponenty souvislosti
- artikulace a mosty
- testování planarity
- hledání cesty v bludišti
- generování bludiště

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

TOPOLOGICKÉ USPOŘÁDÁNÍ

TOPOLOGICKÉ USPOŘÁDÁNÍ VRCHOLŮ GRAFU

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



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

otázky

- existuje v daném grafu topologické uspořádání?
- jak najít topologické uspořádání?

Orientovaný graf G má topologické uspořádání právě když je acyklický.

$\Rightarrow$ existence topologického uspořádání implikuje acykličnost

$\Leftarrow$ acyklický graf má topologické uspořádání

- tvrzení platí pro $n = 1$
- předpokládejme platnost pro všechny grafy s $k < n$ vrcholy
- nechť G má n vrcholů
- v G najdi vrchol v, do kterého nevstupuje žádná hrana (*kdyby takový neexistoval, tak bychom mohli z libovolného vrcholu jít donekonečna „pozpátku" a našli bychom cyklus*)
- graf $G \setminus v$, který vznikne z G odstraněním vrcholu v, je acyklický a má $n - 1$ vrcholů
- dle indukčního předpokladu má $G \setminus v$ topologické uspořádání
- topologické uspořádání vrcholů grafu G má na prvním místě vrchol v následovaný topologickým uspořádáním vrcholů grafu $G \setminus v$

naivní algoritmus

TOPOLOGICAL_SORT_VISIT(G)

$n \leftarrow |V|$
for $i = 1$ **to** n **do**
    $v \leftarrow$ libovolný vrchol, do kterého nevstupuje žádná hrana
    $S[i] \leftarrow v$
    odstraň z G vrchol v a všechny jeho hrany
od
return $S[1 \dots n]$

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

existuje efektivnější algoritmus (ideálně s lineární složitostí)?

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

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

$\Leftarrow$ nechť žádná hrana není zpětná

- předpokládejme, že 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 — spor

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)?

- vrchol u je šedivý
- vrchol 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 $v.f < u.f$

PRŮZKUM GRAFŮ A GRAFOVÁ SOUVISLOST

SILNĚ SOUVISLÉ KOMPONENTY

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
- silně souvislá komponenta grafu je 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

jak 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 v neorientovaném grafu můžeme použít BFS nebo DFS
- jednotlivé DFS (BFS) stromy korespondují s komponentami souvislosti
- složitost $\mathcal{O}(V + E)$

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 $rev(G)$, na který aplikujeme $\text{DFS\_Visit}(rev(G), u)$
- silně souvislá komponenta obsahující u je průnikem množin $Reach(u) \cap Reach^{-1}(u)$
- časová složitost výpočtu je $\mathcal{O}(V + E)$

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

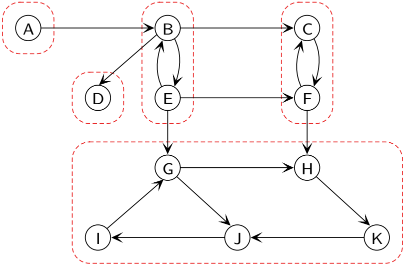
výpočet všech silně souvislých komponent orientovaného grafu

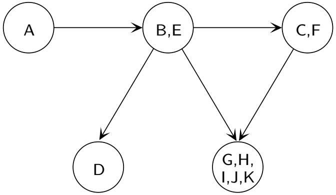
- můžeme *zabalit* popsaný postup
  podobně jako je $\mathrm{DFS\_V\scriptsize{ISIT}}(G, u)$ *zabalené do* DFS
- v nejhorším případě má graf $|V|$ komponent souvislosti a detekce každé z nich má časovou složitost $\mathcal{O}(E)$
- celková časová složitost výpočtu je $\mathcal{O}(V \cdot E)$

existuje efektivnější algoritmus, optimálně s lineární časovou složitostí ?

KOMPONENTOVÝ GRAF

- komponentový graf (*graf silně souvislých komponent*, $scc(G)$) je orientovaný graf, který vznikne kontrakcí každé silně souvislé komponenty grafu do jednoho vrcholu a kontrakcí paralelních hran do jedné hrany
- $scc(G)$ je orientovaný acyklický graf, jeho vrcholy můžeme topologicky uspořádat
- kořenem grafu $scc(G)$ nazýváme vrchol do kterého nevstupuje žádná hrana; kořenu grafu $scc(G)$ odpovídá kořenová silně souvislá komponenta grafu G
- listem grafu $scc(G)$ nazýváme vrchol ze kterého nevystupuje žádná hrana; listu grafu $scc(G)$ odpovídá listová silně souvislá komponenta grafu$G$
- komponentový graf transponovaného grafu $rev(G)$ je práve transponovaný graf $rev(scc(G))$





kořenová komponenta A

listové komponenty D, GHIJK

- z vrcholu u listové silně souvislé komponenty C jsou dosažitelné právě a pouze vrcholy z C
- DFS průzkum z u navštíví všechny vrcholy z C a žádné jiné; na tomto pozorování můžeme postavit algoritmus pro detekci komponent

- potřebujeme (efektivně) najít vrchol patřící listové komponentě
- využijeme fakt, že listová komponenta grafu G je kořenovou komponentou grafu $rev(G)$
- pro hledání kořenové komponenty využujeme fakt, že vrchol s nejvyšší časovou značkou $.f$ leží v kořenové komponentě; tento fakt je důsledkem obecnějšího pozorování

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 pak se teprve vrátí do v

ALGORITMUS KOSARAJU SHARIR

vstup: orientovaný graf $G(V, E)$

výstup: silně souvislé komponenty grafu G

1. prozkoumej graf G průzkumem do hloubky
2. uspořádaj vrcholy grafu G podle časové značky $.f$ v klesajícím pořadí *(reverse postorder)*
3. dokud graf G není prázdný
    - nechť v je vrchol s nejvyšší časovou známkou $.f$ v G
    - v transponovaném grafu $rev(G)$ průzkumem do hloubky najdi množinu C všech vrcholů dosažitelných z v
    - vrcholy množiny C tvoří silně souvislou komponentu grafu G
    - odstraň vrcholy C z grafu G

KOSARAJU_SHARIR($G(V, E)$)

```
foreach u ∈ V do u.color ← white od
foreach u ∈ V do if u.color = white then PUSH_DFS(G, u) fi od
foreach u ∈ V do u.color ← white od
count ← 0
while S ≠ ∅ do
        u ← S.pop()
        if u.color = white then count ← count + 1
                                LABEL_DFS(Rev(G), u, count) fi
od
```

```
Push_DFS(G, u)
u.color ← gray
foreach v ∈ Adj[u] do
    if v.color = white then Push_DFS(G, v) fi
od
u.color ← black
S.push(u)
```

```
Label_DFS(G, u, count)
u.color ← gray
foreach v ∈ Adj[u] do
    if v.color = white then Label_DFS(G, v, count) fi
od
u.color ← black
u.label ← count
```

- algoritmus Kosaraju Sharir má časovou složitost $\mathcal{O}(V + E)$
- další algoritmy lineární časové složitosti pro dekompozici grafu na silně souvislé komponenty: Tarjan, Gabow

NEJKRATŠÍ CESTY

NEJKRATŠÍ CESTY

FORMULACE PROBLÉMU NEJKRATŠÍCH CEST

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

v je dosažitelný z u (značíme $u \rightsquigarrow v$) právě když existuje cesta $p = \langle v_0, v_1, \ldots, v_k \rangle$ taková, že $v_0 = u$ a $v_k = v$

DÉLKA CESTY

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

délka cesty $p = \langle v_0, v_1, \dots, v_k \rangle$ 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

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

nejkratší cesta z u do v je libovolná cesta p z u do v t.ž. $w(p) = \delta(u, v)$

pro neohodnocené grafy se délka cesty definuje jako počet hran cesty

VARIANTY PROBLÉMU NEJKRATŠÍ CESTY

nejkratší cesty z daného vrcholu do všech vrcholů *tato přednáška*

single source shortest path, SSSP

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 *tato přednáška*

speciální případ SSSP, nejsou známy asymptoticky rychlejší algoritmy než pro SSSP

nejkratší cesty mezi všemi dvojicemi vrcholů *ADS II*

řešení opakovanou aplikací algoritmu pro SSSP

existují efektivnější algoritmy

nejdelší, nejširší, nejspolehlivější . . . cesty *viz literatura*

ALGORITMY PRO SSSP - ORIENTOVANÉ GRAFY

neohodnocený graf

průzkum do šířky, BFS

acyklický graf

relaxace hran respektující topologické uspořádáni

graf s nezáporným ohodnocením hran

Dijkstrův algoritmus a jiné

obecný graf

algoritmus Bellmana Forda a jiné

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

NEJKRATŠÍ CESTA VS NEJKRATŠÍ JEDNODUCHÁ CESTA

graf neobsahuje hrany záporné délky

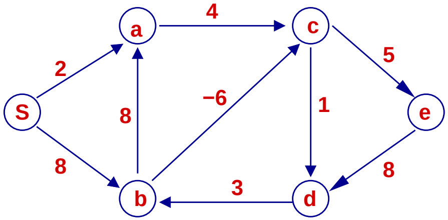
jestliže mezi dvojicí vrcholů existuje cesta,tak mezi nima existuje taková nejkratší cesta, která je jednoduchá

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 úloha rozhodnout, zda graf obsahuje cyklus záporné délky a když ne, tak najít nejkratší (jednoduchou) cestu

VLASTNOSTI NEJKRATŠÍCH CEST

Každá podcesta nejkratší cesty je nejkratší cestou.

- nechť p je nejkratší cesta z u do v
  $w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$

$$\begin{tikzcd} u \arrow[r, squiggly, "p_{ux}"] & x \arrow[r, squiggly, "p_{xy}"] & y \arrow[r, squiggly, "p_{yv}"] & v \end{tikzcd}$$

- 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

NEJKRATŠÍ CESTY

GENERICKÝ SSSP ALGORITMUS

otázka efektivní reprezentace nejkratších cest z daného vrcholu do všech vrcholů grafu

strom nejkratších cest

pozor na rozdíl mezi stromem nejkratších cest a nejlevnější kostrou grafu!

reprezentace stromu nejkratších cest z vrcholu s

atribut vzdálenost, $v.d$ *distance*

- 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, $v.\pi$ *parent*

- 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)

foreach $v \in V$ **do** $v.d \leftarrow \infty$; $v.\pi \leftarrow Nil$ **od**

$s.d \leftarrow 0$

RELAX(u, v, w)

$v.d \leftarrow u.d + w(u, v)$

$v.\pi \leftarrow u$

GENERIC_SSSP(G, w, s)

INIT_SSSP(G, s)

$S \leftarrow s$

while $S \neq \emptyset$ **do**

    vyber u z S

    foreach $(u, v) \in E$ **do**

        if $v.d > u.d + w(u, v)$

            then RELAX(u, v, w)

                $S \leftarrow S \cup \{v\}$ **fi od 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

  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

NEJKRATŠÍ CESTY

ALGORITMUS BELLMANA A FORDA

ALGORITMUS BELLMANA A FORDA

- algoritmus pro hledání nejkratších cest z daného vrcholu s do všech vrcholů grafu
- graf může obsahovat hrany záporné délky
- algoritmus vrátí hodnotu *false* právě, když graf neobsahuje cyklus záporné délky dosažitelný z vrcholu s
- v opačném případě vrátí hodnotu *true* a vypočítá nejkratší cesty

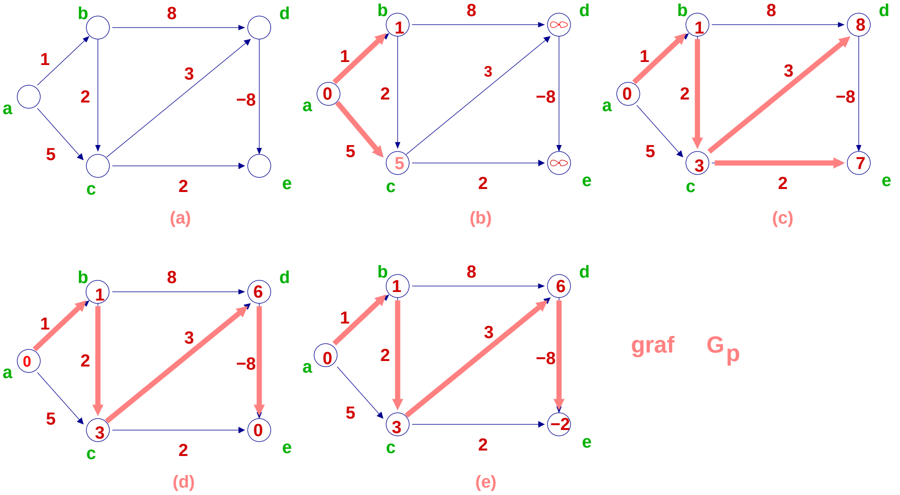
- algoritmus je založený relaxaci hran
- vždy, když vrcholu u zlepšíme hodnotu $u.d$, tak relaxujeme všechny hrany vycházející z vrcholu u

- pro přehlednost rozdělujeme výpočet do *iterací*; v jedné iteraci se relaxují všechny hrany grafu

```
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:

- *jestliže v iteraci* **for** *cyklu v řádcích 2 - 6 nebyla nalezena žádná napjatá hrana, výpočet můžeme ukončit s návratovou hodnotou True*
- *hranu (u, v) relaxujeme v iteraci $i + 1$ pouze pokud v iteraci i byla změněna hodnota $u.d$*



hledáme nejkratší cesty z vrcholu a

v každé iteraci cyklu algoritmu 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 algoritmu Bellmana a Forda 1

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

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- $\sum_{i=1}^{k} v_{i-1}.d = \sum_{i=1}^{k} v_i.d$ protože $v_0 = v_k$
- $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- spor s předpokladem o délce cyklu c

korektnost algoritmu Bellmana a Forda 2

Pro každý vrchol v platí

1. $v.d$ je délka nějaké cesty z s do v
2. hodnota $v.d$ neroste
3. po i iteracích platí, že $v.d \leq$ délka nejkratší cesty z s do v obsahující nejvýše i hran

důkaz tvrzení 3 indukcí vzhledem k i

- tvrzení platí pro $i = 0$
- předpokládejme platnost po iteraci i
- nechť p je cesta z s do v mající nejvýše $\leq i + 1$ hran
- nechť (x, v) je poslední hrana p a nechť $\overline{p}$ je podcesta p z s do x
- dle indukčního předpokladu po iteraci i platí $x.d \leq w(\overline{p})$
- po relaxaci hrany (x, v) v iteraci $i + 1$ platí
  $v.d \leq w(x, v) + w(\overline{p}) = w(p)$

korektnost algoritmu Bellmana a Forda 3

Když graf G **neobsahuje** cyklus záporné délky dosažitelný z s, tak algoritmus vrátí hodnotu *True* a pro každý vrchol v platí $v.d = \delta(s, v)$.

- z neexistence cyklu záporné délky plyne existence jednoduché nejkratší cesty
- rovnost $v.d = \delta(s, v)$ je důsledkem vlastnosti 3
- po ukončení výpočtu platí pro každou hranu $(u, v) \in E$

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= u.d + w(u, v) \end{aligned}$$

t.j., žádná hrana není napjatá a test na řádku 8 nevrátí hodnotu *False*

korektnost algoritmu Bellmana a Forda 4

V průběhu výpočtu algoritmu ukazatele $.\pi$ určují orientovanou cestu z s do v délky $v.d$.

NEPLATÍ

korektnost algoritmu Bellmana a Forda 5

Orientovaný cyklus C v grafu předchůdců G_p má zápornou délku.

- z $v.\pi = u$ plyne $v.d \geq u.d + w(u, v)$
- nechť $c = \langle v_1, \ldots, v_k, v_1 \rangle$ je orientovaný cyklus v grafu G_p a nechť (v_k, v_1) je ta hrana cyklu, která byla do G_p přidána jako poslední
- bezprostředně před přidáním hrany (v_k, v_1) do G_p platilo

$$\begin{aligned} v_2.d &\geq v_1.d + w(v_1, v_2) \\ v_3.d &\geq v_2.d + w(v_2, v_3) \\ \vdots \quad & \quad \vdots \qquad \vdots \\ v_k.d &\geq v_{k-1}.d + w(v_{k-1}, v_k) \\ v_1.d &> v_k.d + w(v_k, v_1) \end{aligned}$$

- sečtením nerovností

  dostáváme $w(v_1, v_2) + w(v_2, v_3) + \ldots + w(v_k, v_1) < 0$

korektnost algoritmu Bellmana a Forda 6

Když graf G **neobsahuje** cyklus záporné délky dosažitelný z s, tak po ukončení výpočtu graf předchůdců G_p obsahuje nejkratší cesty.

- G_p neobsahuje cyklus
- pro každé v, ukazatele $.\pi$ jednoznačně určují cestu $\langle v_1, \ldots, v_k \rangle$ kde $v_1 = s$ a $v_k = v$
- po ukončení výpočtu žádná hrana není napjatá, t.j.

$$
\begin{aligned}
v_2.d &= v_1.d + w(v_1, v_2) \\
v_3.d &= v_2.d + w(v_2, v_3) \\
\vdots \quad & \quad \vdots \quad\quad \vdots \\
v_k.d &= v_{k-1}.d + w(v_{k-1}, v_k)
\end{aligned}
$$

- sečtením rovností dostáváme $v.d = v_k.d = s.d + w(v_1, v_2) + w(v_2, v_3) + \ldots + w(v_{k-1}, v_k)$

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?

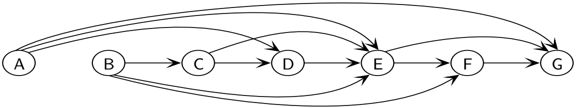
- lehce najdeme graf a takové pořadí relaxace jeho hran, pro které je nutných $V - 1$ iterací
- otázka vhodného pořadí relaxace hran
- vhodné pořadí hran dokážeme určit pro speciální typy grafů
  - acyklické grafy
  - grafy bez záporných hran

NEJKRATŠÍ CESTY

ACYKLICKÉ GRAFY

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



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

DAG(G, w, s)

```
najdi topologické uspořádání vrcholů grafu G
INIT_SSSP(G, s)
foreach vrchol u v topologickém uspořádání do
   foreach (u, v) ∈ E do
      if v.d > u.d + w(u, v) then RELAX(u, v, w) fi
   od
od
```

- časová složitost $\Theta(V + E)$
- topologické uspořádání garantuje, že hrany *každé* cesty jsou relaxované v pořadí, v jakém se vyskytují na cestě

NEJKRATŠÍ CESTY

DIJKSTRŮV ALGORITMUS

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*

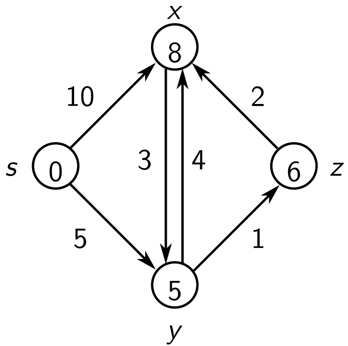
- řeší problém SSSP pro grafy **s nezáporným ohodnocením hran**

DIJKSTRA(G, w, s)

```
INIT_SSSP(G, s)
S ← ∅
Q ← V
while Q ≠ ∅ do
        u ← EXTRACT_MIN(Q)
        S ← S ∪ {u}
        foreach (u, v) ∈ E do
           if v.d > u.d + w(u, v) then RELAX(u, v, w) fi
        od
od
```

- S je množina vrcholů, pro které je již vypočtena délka nejkratší cesty
- Q je 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 Dijkstrova algoritmu

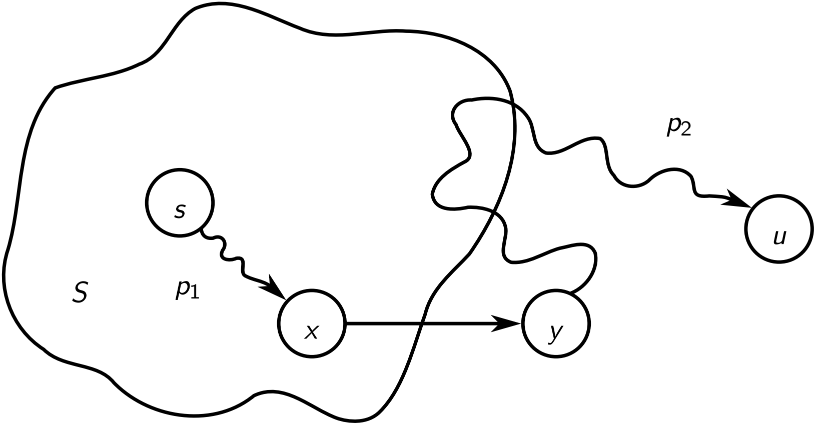
Invariant: Na začátku iterace **while** cyklu platí $v.d = \delta(s, v)$ pro všechny vrcholy $v \in S$.

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 máme prokázat, že když u přesuneme do S, tak $u.d = \delta(u.s)$

- když u není dosažitelný z s tak $u.d = \delta(s, u) = \infty$
- v opačném případě nechť p je nejkratší cesta z s do u; cestu p můžeme dekomponovat na dvě cesty $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$ 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), $s \overset{p_1}{\leadsto} x \to y$ 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 $u.d \leq y.d = \delta(s, y) \leq \delta(s, u) \implies u.d \leq \delta(s, u)$



s
S
p_1
x
y
p_2
u

složitost Dijkstrova algoritmu

$\Theta(V)$ operací INSERT (do prioritní fronty přidá nový objekt)

$\Theta(V)$ operací EXTRACT_MIN (z fronty odstraní objekt s minim. klíčem)

$\Theta(E)$ operací DECREASE_KEY (objektu v prioritní frontě sníží hodnotu klíče)

složitost algoritmu závisí od implementace prioritní fronty Q

pole složitost $\Theta(V \cdot V + E \cdot 1) = \Theta(V^2)$

INSERT $\Theta(1)$, EXTRACT_MIN $\Theta(V)$, DECREASE_KEY $\Theta(1)$

binární halda složitost $\Theta(V \log V + E \log V)$

INSERT $\Theta(\log V)$, EXTRACT_MIN $\Theta(\log V)$, DECREASE_KEY $\Theta(\log V)$

Fibonacciho halda složitost $\Theta(V \log V + E)$

INSERT $\Theta(1)$, EXTRACT_MIN $\Theta(\log V)$, DECREASE_KEY $\Theta(1)$

NEJKRATŠÍ CESTY

NEJKRATŠÍ CESTA MEZI DVĚMA VRCHOLY

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 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 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

- vytvoříme 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 mezi s a t je pro všechny cesty z s do t stejný

- 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

modifikace Dijkstrova algoritmu

jestliže se vrcholu u, který již byl odebrán z prioritní fronty, změní hodnota $u.d$, tak vrchol u znovu vložíme do prioritní fronty

pro graf, ve kterém hrany mohou mít zápornou délku

- jestliže z počátečního vrcholu není dosažitelný cyklus záporné délky, tak modifikovaný Dijkstrův algoritmus najde korektní řešení, ale složitost výpočtu je v nejhorším případě až exponenciální vůči velikosti grafu
- jestliže z počátečního vrcholu je dosažitelný cyklus záporné délky, tak výpočet modifikovaného Dijkstrova algoritmu je nekonečný

NEJKRATŠÍ CESTY

LINEÁRNÍ NEROVNICE

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)

motivace

- 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

algoritmická složitost

- existují polynomiální algoritmy pro řešení úlohy lineárního programování
- pro řešení speciálních případů úlohy lineárního programování existují mnohem rychlejší algoritmy
- problém lineárních nerovnic je příkladem takovéto speciální úlohy
- ukážeme algoritmus založený na SSSP

PROBLÉM LINEÁRNÍCH NEROVNIC

- je daná množina nerovnic tvaru $x - y \leq b$, kde x, y jsou proměnné a b je konstanta
- úkolem je najít takové hodnoty proměnných, 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

pro danou množinu M lineárních nerovnic nad proměnnými $x_1, \ldots, x_n$ definujeme orientovaný graf $G = (V, E)$

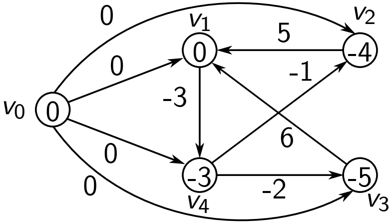
- $V = \{v_0, v_1, \ldots, v_n\}$ (*jeden vrchol pro každou proměnnou plus vrchol v_0*)
- $E = \{(v_i, v_j) \mid x_j - x_i \leq b \in M\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$

hrany grafu ohodnotíme tak, že

- $w(v_0, v_i) = 0$, pro $1 \leq i \leq n$
- $w(v_i, v_j) = b$ právě když $x_j - x_i \leq b \in M$

příklad

$$\begin{aligned} x_1 - x_2 &\leq 5 \\ x_1 - x_3 &\leq 6 \\ x_2 - x_4 &\leq -1 \\ x_3 - x_4 &\leq -2 \\ x_4 - x_1 &\leq -3 \end{aligned}$$



nejkratší cesty v grafu lineárních nerovnic

Pro daný systém lineárních nerovnic a k němu odpovídající graf lineárních nerovnic $G = (V, E)$ platí:

1. když G nemá cyklus záporné délky, tak přípustným řešením systému nerovnic je
$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$$
2. když G má cyklus záporné délky, tak systém nemá přípustné řešení.

zdůvodnění — část 1.

- neexistence cyklu záporné délky implikuje, že pro každou hranu (v_i, v_j) grafu platí $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$
- hraně (v_i, v_j) odpovídá nerovnost $x_j - x_i \leq w(v_i, v_j)$, která je pro hodnoty $x_i = \delta(v_0, v_i)$ a $x_j = \delta(v_0, v_j)$ splněna

zdůvodnění — část 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
   $x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$