Algoritmy a datové struktury I

Jan Hric, KTIML MFF UK
e-mail: Jan.Hric@mff.cuni.cz
http://ktiml.mff.cuni.cz/~hric/vyuka/alg/ads1pr.pdf

9. dubna 2025

Sylabus

- Prostředky pro popis složitosti algoritmů a operací nad datovými strukturami, tzv. asymptotická notace, O-notace
- Grafové algoritmy: prohledávaní, topologické třídění, SSK
- Extremální cesty v grafech (a dat.struk. halda)
- Minimální kostra
- (Binární stromy,) AVL-stromy, (Červenočerné stromy)
- (a,b)-stromy
- Hašování
- Metoda Rozděl a panuj
- Dynamické programování
- Třídění: Dolní odhad složitosti (problému) třídění, průměrný případ Quicksortu, randomizace Quicksortu, (lineární třídící algoritmy)

Literatura:

T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, Introduction to Algorithms, MIT Press, 1991

Martin Mareš, Tomáš Valla, Průvodce labyrintem algoritmů, CZ.NIC, z. s. p. o., Praha, 1. vydání, 2017, ISBN 978-80-88168-22-5, http://pruvodce.ucw.cz/, datum přístupu 8.4.2019

A. Koubková, J. Pavelka, Úvod do teoretické informatiky, Matfyzpress, 1998

L. Kučera, Kombinatorické Algoritmy, SNTL, Praha 1983

Měření a porovnávání algoritmů:
- časová složitost
- prostorová (paměťová) složitost
- komunikační složitost (napr. počet paketů)

Používáme funkce závislé na velikosti vstupních dat:
- od konkrétních dat k datům určité velikosti
- porovnáváme funkce

Jak měřit velikost vstupních dat? (i jiných dat)
rigorózně: počet bitů nutných k zapsání vstupních dat

Příklad: vstup jsou (přirozená) čísla $a_1, ..., a_n \in N$ velikost dat D v binárních zápisu je $|D| = \sum_{i=1}^{n} \lceil \log_2 a_i \rceil$

Odstranění závislosti na konkrétních datech:
- v nejhorším případě
- v průměrném případě (vzhledem k pravděpodobnostnímu rozložení vstupních dat)

Časová složitost algoritmu: funkce $f : N \to N$ taková, že $f(|D|)$ udává počet kroků algoritmu v závislosti na datech velikosti $|D|$.

Intuitivně: není podstatný přesný průběh funkce f ("až na multiplikativní a aditivní konstanty"), ale to, do jaké "třídy"funkce f patří (lineární, kvadratická, ...)

Co je krok algoritmu:
- teoreticky: operace daného abstraktního stroje (Turingův stroj, stroj RAM)
- zjednodušeně (budeme používat): krok algoritmu = operace proveditelná v konstantním čase (nezávislém na velikosti dat)
- aritmetické operace (+, -, *, ...)
- porovnání dvou hodnot (typicky čísel)
- přiřazení pro jednoduché datové typy (ale ne pro pole)

- tím se zjednoduší i měření velikosti dat (čísla mají pevnou maximální velikost)

Příklad: setřídit čísla $a_1, ..., a_n$: velikost dat $|D| = n$

Toto zjednodušení nevadí při porovnání algoritmů, ale může vést k chybě při zařazování algoritmů do tříd složitosti. (Přesnější měření kroku: cena operace závisí na velikosti zpracovávaných dat, tj. na počtu bitů/buněk)

Proč měřit časovou složitost: (slajd)
- proč někdy nepomůže rychlejší počítač:

Tabulka rychlosti růstu funkcí.

Funkce	n=10	n=30	n=50	n=100	n=1000
$\log n$	1	1,48	1,70	2	3
n	10	30	50	100	1000
$n \log n$	10	44.3	84.9	200	3000
n^2	100	900	2500	10 000	10^6
n^3	1000	$\approx 3 \cdot 10^4$	$\approx 10^5$	10^6	10^9
2^n	1024	$\approx 10^9$	$\approx 10^{15}$	$\approx 10^{30}$	$\approx 10^{301}$

Pokud jednotka je 1 ns: jeden den má přibližně 10^{14} ns, 1 rok má přibližně $10^{16.5}$ ns.

Asymptotická složitost

- zkoumá chování algoritmu na "velkých"datech (ignoruje konečný počet výjimek)
- zařazuje algoritmy do "kategorií"
- zanedbává multiplikativní a aditivní konstanty

Definice:

- $f(n)$ je asymptoticky menší nebo rovno $g(n)$, značíme $f(n) \in O(g(n))$,
  pokud $\exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c.g(n)$
- $f(n)$ je asymptoticky větší nebo rovno $g(n)$, značíme $f(n) \in \Omega(g(n))$,
  pokud $\exists c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c.g(n) \leq f(n)$
- $f(n)$ je asymptoticky stejné jako $g(n)$, značíme $f(n) \in \Theta(g(n))$,
  pokud $\exists c_1, c_2 > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$
- $f(n)$ je asymptoticky ostře menší než $g(n)$, značíme $f(n) \in o(g(n))$,
  pokud $\forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq c.g(n)$
- $f(n)$ je asymptoticky ostře větší než $g(n)$, značíme $f(n) \in \omega(g(n))$,
  pokud $\forall c > 0 \exists n_0 \forall n \geq n_0 : 0 \leq c.g(n) \leq f(n)$

Příklady kategorií funkcí: $\Theta(\underline{1})$, $\log(\log n), \log n, \log^2(n)$, $n^{1/2}$, n, $n \log n, n^2$, $n^3, 2^n, 2^{2n}$, $n!, n^n, 2^{2^n}, ...$

Pozn: některé dvojice funkcí nejdou porovnat

DC: (Značení $O(f+g)$) Dokažte: $max(f(n), g(n)) \in \Theta(f(n)+g(n))$.

DC: pro $c_m, c_a > 0$ a $g(n) = c_m.f(n) + c_a$ dokažte $g \in O(f)$ (a $g \in \Theta(f)$)

Příklad tvrzení a důkazu:
Pokud $f \in O(h)$ a $g \in O(h)$, potom $f + g \in O(h)$.
Dk-idea: máme c_f, n_f, c_g, n_g pro c, n_0 z předpokladů o f a g z definice "O". Zvolíme $c = c_f + c_g$, $n_0 = max(n_f, n_g)$, potom pro každé $n \geq n_0$ platí $0 \leq f(n) + g(n) \leq c_f.h(n) + c_g.h(n) = (c_f + c_g).h(n) = c.h(n)$, QED

Aplikace: odhad složitosti sekvence příkazů.

Pozn: nevhodný zápis $f = O(g)$

Pozn: vliv multiplikativní a aditivní konstanty v asymptotické notaci:
Porovnání $n + 100$ a n^2, $2^{100}n$ a n^2
Složitost $f(n) = 10n$ je lepší než $g(n) = 1n \log n$ až pro $n > 2^{10}$
Složitost $f(n) = 5n^{1.9}$ je lepší než $g(n) = 1n^2$ až pro $n > 5^{10}$
Složitost $f(n) = 2^{1000}n$ je lepší než $g(n) = 2^n$ pro $n \geq 1010$
Malá data je někdy vhodné zpracovat jednodušším algoritmem s menší režií, i když má větší asymptotickou složitost.

Haldy ... u grafových algoritmů

Grafové algoritmy

Reprezentace grafu $G = (V, E)$, kde V je množina vrcholů a E je množina hran

1. Matice sousednosti
   $A = (a_{ij})$ typu $|V| \times |V|$
   $a_{ij} = 1$ pokud $(v_i, v_j) \in E$
   $= 0$ jinak

2. Seznamy sousedů
   pole `sousedé` velikosti V
   pro $u \in V$ je `sousedé[u]` hlava seznamu obsahující vrcholy v, pro které platí $(u, v) \in E$
   - paměť: $O(|V| + |E|) = O(\max(|V|, |E|))$
   - lze použít pro varianty:

   (a) neorientovaný graf

   (b) (hranově) ohodnocený graf: cena: $E \to R$

   (c) seznamy sousedů generované dynamicky (šetří paměť)

3. Výpočtem, tj. *implicitně* zadaný graf: `jeHrana(G,u,v)` vrací *boolean*, `sousedi(G,v)` vrací sousedy (pole/seznam, i líně/iterátorem)

Použití grafů: mapa ulic města, graf výpočtu (závislostí mezivýsledků), toky v sítích (dopravní, dodavatelsko-odběratelské; bude v ADS2), závislosti činností v plánování, odkazy objektů na haldě (pro garbage collector), stavový prostor prohledávání

Prohledávání grafů

Dvě metody:

- prohledávání do hloubky (DFS - Depth First Search)
- prohledávání do šířky (BFS - Breadth First Search)

Prohledávání do hloubky

Vstup: graf $G = (V, E)$, zadaný pomocí seznamů sousedů
pomocné datové struktury:

- $\pi(v)$ . . . otec vrcholu v ve stromu prohledávání
- $p(v)$ . . . pořadí, v němž jsou vrcholy $v \in V$ navštíveny
- pořadí . . . globální proměnná, slouží k číslování vrcholů

Algoritmus DFS(G)

```
 1 forall u in V do p(u)<-0, pi(u)<-NIL od
 2 pořadí <- 0
 3 forall u in V do
 4   if  p(u)=0 then Navštiv(u) fi
 5                od

 6 procedura Navštiv(u)
 7 pořadí++; p(u)<-pořadí
 8 forall v in sousedé[u] do
 9   if p(v) = 0 then
10     pi(v) <- u
11     Navštiv(v)
12   fi
13 od. // uzavření před výstupem
```

Časová složitost: $O(m+n), n = |V|, m = |E|$

- graf průchodu do hloubky (DFS-les):

$G_\pi = (V, E_\pi)$, kde

$E_\pi = \{(\pi(v), v) | v \in V \wedge \pi(v) \neq NIL\}$

- Nerekurzivní implementace DFS: pamatuji si zásobník vrcholů, u každého vrcholu u:`Navštiv:6` jeho souseda v:`Navštiv:8` aktuálně navštěvovaného (např. odkaz do seznamu sousedů).

Paměťová složitost: velikost zásobníku může být až $O(n)$.

(Př.: Vyplňování jednobarevné plochy jinou barvou.)

- Rozlišování vrcholů: nenavštívené, otevřené (má určené pořadí otevření), uzavřené (má určené pořadí uzavření).

Df: uzavřený vrchol v: všechny hrany vedoucí z v jsme prohledali

- Pro některé aplikace potřebujeme místo časů otevření (tj. pořadí) časy uzavření (nebo obojí).

- Aplikace DFS:

komponenty grafu

existence kružnice; test, zda graf je strom/les

Klasifikace hran grafu G:

stromová hrana - vede do nového vrcholu

zpětná hrana - vede do už navštíveného (neuzavřeného) vrcholu

dopředná hrana (u, v) - pouze v orientovaném grafu
- vede do uzavřeného vrcholu v, kde pořadí$(u) <$ pořadí(v)

příčná hrana - pouze v orientovaném grafu
- vede do uzavřeného vrcholu v, kde pořadí$(u) >$ pořadí(v)

- pozn: (programování řízené událostmi:) některé algoritmy lze popsat tak, že jednotlivé druhy hran, případně události otevření a uzavření vrcholu, mají své výkonné procedury
- generický alg. a označení vrcholů barvami: černá - uzavřený, tj. prohledaný, šedý - otevřený, tj. prohledávaný, bílá - neotevřený. Udržování invariantu: následníci černého jsou šedí. Pokud projdu všechny šedé vrcholy (a přebarvím je na černo), pak právě všechny dosažitelné vrcholy grafu (ze zvolené šedé počáteční množiny) jsou černé (a nedosažitelné bílé).
- aplikace: garbage collector (čistění dyn. paměti), serializace dyn. paměti
- impl. (pro implicitní grafy, pragmatická kombinace DFS a BFS): iterativní prohlubování (iterative deepening)

Prohledávání do šířky

Vstup: graf $G = (V, E)$, zadaný pomocí seznamů sousedů a vrchol s, ve kterém začíná prohledávání
pomocné datové struktury:

- $\pi(v)$ ... otec vrcholu v ve stromu prohledávání
- $d(v)$ ... vzdálenost z vrcholu s do v
- F ... fronta (neuzavřených vrcholů), operace Přidej, Odeber, funkce Prázdná

Algoritmus BFS(G)

```
 1 forall u in V\{s} do d[u]<- +inf, pi(u)<-NIL od
 2 d[s]<-0; pi[s]<-NIL; Pridej(F,s);
 3 while not Prazdna(F) do
 4   Odeber(F,u);
 5   forall v in sousede(u) do
 6     if d(v) = +inf then
 7       d(v) <- d(u)+1;
 8       pi(v) <- u;
 9       Pridej(F,v);
10     fi
11   od
12 od.
```

Časová složitost: $O(m + n), n = |V|, m = |E|$

- strom průchodu do šířky (BFS-strom):

$G_\pi = (V_\pi, E_\pi)$
$V_\pi = \{v \in V | \pi(v) \neq NIL\} \cup \{s\}$
$E_\pi = \{(\pi(v), v) | v \in V_\pi \setminus \{s\}\}$

- aplikace: nejkratší cesta (na jednotkově ohodnoceném grafu), $d(v)$ je délka nejkratší cesty z s do v;

- rekonstrukce cesty (bude i jinde): pomocí π odzadu

Detekce mostů

Most je hrana, jejímž odstraněním se zvětší počet komponent grafu.

Charakteristika mostu: most neleží na kružnici.

Testování hrany (u, v), zda je most v G: z definice pomocí DFS: V $G \backslash (u, v)$ najdu komponentu souvislosti pro u. Pokud v ní leží v, pak hrana (u, v) není most.

- Varianta: hledám cestu z u do v.

- Složitost: $O(n + m)$ jako DFS.

- Impl.: V obou variantách můžu ukončit prohledávání hned po nalezení vrcholu, resp. cesty. Ale optimalizace nezmění složitost v nejhorším případě.

Chceme určit všechny mosty na jeden průchod DFS.

Pozorování: Zpětná hrana není most. Zpětná hrana (v, u) zakáže stromovým hranám na cestě z u do v být mostem. Ale nechceme pro jednotlivé v procházet cestu samostatně.

Idea: určíme a zapamatujeme si pro každý vrchol u hodnotu $low(u)$: nejmenší čas otevření vrcholu, který je dosažitelný zpětnou hranou z podstromu u (včetně u).

Test a update pro stromovou hranu $h = (u, v)$: Pokud $low(v) \geq p[v]$, je h most: podmínka znamená, že žádná zpětná hrana z podstromu v nevede nad v.
Pokud se vracíme z v do u (po uzavření v), měníme $low(u) :=$

$min(low(u), low(v))$ (Pamatujeme si menší čas uzavření, tj. vrchol blíž ke kořeni.)

Složitost: konstantní čas na jednu hranu, tj. celkem $O(n + m)$ jako DFS.

- Podobnou metodu lze použít na hledání *artikulací*, tj. vrcholů, jejichž odstranění zvětší počet komponent. (Samostatně nutno testovat, zda je kořen artikulace.)

Topologické uspořádání

Df: Posloupnost $v_1, v_2 \ldots v_n$ je topologické uspořádání vrcholů orientovaného grafu G, pokud pro každou hranu: $(v_i, v_j) \in E \rightarrow i \leq j$

T: Graf G lze topologicky uspořádat $\leftrightarrow$ G je acyklický (tzv. DAG) $\leftrightarrow$ DFS(G) nenajde zpětnou hranu

- DAG: Directed Acyclic Graph
- pojmy:

- vrchol, poprvé navštívený algoritmem DFS, se nazývá otevřeným
- otevřený vrchol se stane uzavřeným, když je dokončeno zpracování seznamu jeho sousedů

Algoritmus Topologické uspořádání(G)

1. volej DFS(G) a spočítej časy uzavření vrcholů
2. `if` existuje zpětná hrana `then`
3. `return` "G není acyklický";
4. každý vrchol, který je uzavřen, ulož na začátek spojového seznamu S

5. `return` S.

- Časová složitost: $\Theta(|V| + |E|)$, v tom:
- prohledávání do hloubky: $\Theta(|V| + |E|)$
- vkládání vrcholů do výstupního seznamu S: $|V| \cdot O(1)$
- (nevhodná impl: třídění hran podle $k(u)$ v $\Theta(n \log n)$)

Silně souvislé komponenty (SSK)

Df: Orientovaný graf je silně souvislý, pokud pro každé dva vrcholy u, v existuje orientovaná cesta z u do v a současně z v do u.

Silně souvislá komponenta grafu je maximální podgraf, který je silně souvislý.

- pozn: Každý vrchol grafu patří do právě jedné komponenty a ty tvoří rozklad množiny vrcholů

- opačný graf $G^T = (V, E^T)$, kde $E^T = \{(u, v)|(v, u) \in E\}$

- $k(v), v \in V$: pořadí, v jakém jsou vrcholy uzavírány algoritmem DFS

Algoritmus SSK(G)

1. Algoritmem DFS(G) urči časy uzavření $k(v)$ pro všechny $v \in V$
2. Vytvoř G^T
3. Uspořádej vrcholy do klesající posloupnosti podle $k(v)$ a v tomto pořadí je zpracuj algoritmem DFS(G^T)
4. Silně souvislé komponenty grafu G jsou právě podgrafy indukované vrcholovými množinami jednotlivých DFS-stromů z minulého kroku.

Korektnost SSK

Lemma 1. Nechť C, C' jsou dvě různé silně souvislé komponenty grafu G. Pokud existuje cesta v G z C do C', pak neexistuje cesta z C' do C.

Značení: Pro $U \subseteq V(G)$ položme $p(U) = min\{p(u)|u \in U\}$ a $k(U) = max\{k(u)|u \in U\}$, kde
$p(u)$ je čas prvního navštívení vrcholu u (otevření)
$k(u)$ je čas uzavření u

Lemma 2. Nechť C, C' jsou dvě různé silně souvislé komponenty grafu G. Existuje-li hrana z C do C', pak $k(C) > k(C')$.

Věta: Vrcholové množiny DFS-stromů vytvořených algoritmem SSK(G) při průchodu do hloubky grafem G^T odpovídají vrcholovým množinám silně souvislých komponent grafu G.

Dk: Indukcí podle počtu projitých stromů.
- z jednoho (prvního navštíveného) vrcholu u komponenty C projdu celou komponentu v G^T i v G, $k(u) = \max\limits_{v \in C} k(v) (= k(C))$
- pokud se dostanu hranou mimo komponentu (do vrcholu v), je v už uzavřený.

DC: Ukažte, že pokud ve druhém průchodu v algoritmu SSK(G) procházíme graf G (místo G^T) v pořadí rostoucích časů uzavření (místo klesajících), nedostaneme správné komponenty.

Složitost SSK: $\Theta(|V| + |E|)$

Df: Kondenzace grafu: Mějme orientovaný graf $G = (V, E)$ a nechť $V_1 \dots V_k$ jsou množiny vrcholů odpovídající silným komponentám grafu G. Orientovaný graf $C(G)$ se nazývá *kondenzace* grafu G a je definován $C(G) = (\{V_1 \dots V_k\}, E')$, kde $(V_i, V_j) \in E'$, pokud existují vrcholy $x \in V_i$ a $y \in V_j$

takové, že $(x, y) \in E$.

- Kondenzace $C(G)$ grafu G neobsahuje cyklus a teda je DAG.

- DC: další aplikace prohledávání DFS (pomocí lowlink: minimum $p(x)$ konců dosažitelných zp.h.): určení mostů, artikulací, určení reprezentantů SSK, výroba kondenzace

Problém nejkratší cesty

Používáme:
- orientovaný graf $G = (V, E)$
- ohodnocení hran $c : E \to R$
- cena orientované cesty P, $P = v_0, v_1 \ldots v_k$ je

$$c(P) = \sum_{i=1}^{k} c(v_{i-1}, v_i)$$

- Cena nejkratší cesty z u do v

$$\delta(u, v) = \min\{c(P) | P \text{ je cesta z } u \text{ do } v\}$$

- nejkratší cesta z u do v je libovolná cesta P z u do v, pro kterou $c(P) = \delta(u, v)$
- $\delta(u, v) = \infty$ znamená, že (orientovaná) cesta neexistuje
- Zavedeme: $\infty + r = \infty, \infty > r \quad$ pro $\forall r \in R$

Varianty problému Najít nejkratší cestu

1. z u do v, $\quad u, v \in V$ pevné
2. z u do x, $\quad$ pro každé $x \in V$, u pevné
3. z x do y, $\quad$ pro každé $x, y \in V$

Pomocná operace UvolněníHrany(u,v)

```
if d(v) > d(u) + c(u,v)
  then d(v) <- d(u)+ c(u,v)
       pi(v) <- u
fi.
```

Rekonstrukce cesty: (opět) pomocí předchůdců π

Extremální cesty v acyklickém grafu

- z jednoho vrcholu do ostatních, verze 2)
- nejkratší cesta v DAG je vždy dobře definovaná, protože i když se vyskytují záporné hrany, neexistují záporné cykly
- Idea: využijeme topologické uspořádání vrcholů

Nejkratší cesta v acyklickém grafu:

Vstup:
acyklický orientovaný graf $G = (V, E)$
$c : E \to R$ ohodnocení hran
$s \in V$ počáteční vrchol
Výstup
$d(v), \pi(v)$ pro všechna $v \in V$
kde $d(v) = \delta(s, v)$

Algoritmus
1. Topologicky uspořádat vrcholy G
2. Inicializace(G,s)
3. `for` každý vrchol u v pořadí topologického uspořádání
4. `do forall` $v \in$`sousedé[u] do`
5. UvolněníHrany(u,v)
6. `od od`.

- Časová složitost: $\Theta(|V| + |E|)$, protože:
- topologické uspořádání: $\Theta(|V| + |E|)$
- zpracování $|V|$ vrcholů, v cyklu: každý jednou v $O(1)$
- zpracování $|E|$ hran ve vnitřním cyklu: každá jednou, při zpracování poč. vrcholu hrany; vlastní zpracování hrany v $O(1)$ při reprezentaci v poli
- vztah k dynamickému programování: na hledání nejkratší

(obecně nejoptimálnější) cesty se lze v případě DAGu dívat jako na algoritmus dynamického programování: pokud mám optimální cesty pro všechny předchůdce, dokážu určit optimální cestu do aktuálního vrcholu. Graf může být zadán implicitně a lze použít obecné optimalizace a varianty algoritmů pro dynamické programování (rekurze s tabelací, převod rekurze na iteraci, průběžné zahazování mezivýsledků).

- v informatice: DAG může sloužit pro popis řídící struktury acyklických výpočtů (např. v dynamickém programování, při transformaci dat, ...) nebo závislostní struktury (tj. návazností) dat (např. podle času, příčinná souvislost)

Aplikace: PERT - Kritická cesta

Problém: Je dána množina *úloh* a délka každé z nich. Některé dvojice úloh mohou na sobě záviset, tzn. jedna úloha musí skončit dřív, než druhá začne. Cílem je zjistit nejkratší čas, ve kterém mohou všechny úlohy skončit.

- reprezentace: Hrany grafu odpovídají úlohám, ohodnocení hran odpovídá času trvání (vykonávání) úlohy. Pokud hrana (u, v) vstupuje do vrcholu v a hrana (v, x) vystupuje z v, musí být úloha (u, v) ukončena před vykonáváním úlohy (v, x). Graf je DAG.

- cesta reprezentuje úlohy, které se musí vykonat v určitém pořadí.

- *Kritická cesta* je nejdelší cesta v grafu. Odpovídá nejdelšímu času pro vykonávání uspořádané posloupnosti úloh. Cena kritické cesty je dolní odhad pro celkový čas dokončení všech úloh.

Algoritmus nalezení kritické cesty, dvě možnosti.

1. opačné ceny hran a hledání nejkratší cesty
2. hledání *nejdelší* cesty v DAG (změna "∞"na "$-\infty$"v inicializaci a "$<$"na "$>$"v UvolněníHrany)

- proč je hledání nejdelší cesty *v DAG* možné efektivně?: platí princip optimality: libovolná část nejdelší cesty je opět nejdelší mezi příslušnými vrcholy.

- impl: hrany s nulovou cenou pro závislosti

- DC: přirozenější reprezentace: úlohy jsou vrcholy s ohodnocením, hrany odpovídají závislostem: hrana (u, v) znamená, že činnost ve vrcholu u se vykonává před činností v.

- pozn: Činnosti na (nějaké) kritické cestě musí být vykonány včas, jejich zpoždění se projeví celkovým zpožděním konce. Činnosti mimo kritickou cestu mají rezervu, resp. čas (nejdřívějšího) možného začátku a (nejpozdějšího) nutného konce. Rezerva je rozdíl těchto časů zmenšený o trvání činnosti. Tyto časy začátku, resp. konce, lze spočítat pomocí průchodu DAG zepředu, resp. zezadu, grafu (pro víc počátečních, resp. koncových vrcholů)

(2022 nepř.) Aplikace: Viterbiho dekódování, zjedn.

Hledání cesty v DAG (ze Z do S), která nejpravděpodobněji vygeneruje danou posloupnost písmen P (nad abecedou Σ). Pro zjednodušení, graf je vrstvený, tj. vrcholy lze přiřadit do vrstev a hrany vedou pouze mezi sousedními vrstvami. Hrany jsou ohodnoceny pravděpodobností přechodu mezi vrcholy, vrcholy (kromě Z a S) mají pravděpodobnosti vypsání

pro jednotlivá písmena ze Σ. V jednom vrcholu se vypisuje jedno písmeno a pak se pokračuje hranou dál. Pro zjednodušení, pravděpodobnosti jsou nenulové a indukované grafy mezi vrstvami úplné (a tedy cesta vždy existuje).

Cílem (algoritmu) je najít takovou cestu ze Z do S, při které se vypíše P s největší možnou pravděpodobností.

(Varianty a zobecnění, aplikace, počítání s log a podtečení, notace argmax ...)

Aplikace Viterbiho dekódování: Převod řeči na text, strojový překlad, hledání podobných genů, ... (Hledání nejpravděpodobnější posloupnosti (skrytých) událostí, která způsobí určitou posloupnost pozorování ve známém (tj. natrénovaném) HMM – Hidden Markov Model)

Dijkstrův algoritmus (1959) pro cesty z jednoho vrcholu do všech (nebo jednoho konkrétního)

Vstup:
$G = (V, E)$ orientovaný graf
$c : E \to R_0^+$ nezáporné ohodnocení hran
$s \in V$ počáteční vrchol

Výstup:
$d(v), \pi(v)$ pro $\forall v \in V$, tž. $d(v) = \delta(s, v)$
$\pi(v)$ je předchůdce vrcholu v na (nějaké) nejkratší cestě z s do v

Algoritmus

```
01 forall v in V(G) do
02   d(v) <- infty % inicializace
03   pi(v) <- NIL   od
04 d(s) <- 0
05 D <- empty
06 N <- V
07 while N <> empty do
08   u <- OdeberMin(N)
09   D <- D union {u}
10   forall v in Sousede[u] & v in N do
11     UvolněníHrany(u,v)
12   od
13 od.
```

Korektnost Dijkstrova alg.
Lemma 1. Optimální podstruktura

Je-li $v_1 \dots v_k$ nejkratší cesta z v_1 do v_k, potom $v_i \dots v_j$ je nejkratší cesta z v_i do v_j pro $\forall i, j, 1 \leq i < j \leq k$

Lemma 2. Trojúhelníková nerovnost
$\delta(s, v) \leq \delta(s, u) + c(u, v)$ pro každou hranu (u, v)
- po provedení inicializace je ohodnocení vrcholů měněno jen prostřednictvím UvolněníHrany

Lemma 3. Horní mez
$d(v) \geq \delta(s, v)$ pro každý vrchol v a po dosažení hodnoty $\delta(s, v)$ se $d(v)$ už nemění

Lemma 4. Uvolnění cesty
Je-li $v_0 \dots v_k$ nejkratší cesta z $s = v_0$ do v_k, potom po uvolnění hran v pořadí $(v_0, v_1), (v_1, v_2) \dots (v_{k-1}, v_k)$ je $d(v) = \delta(s, v)$

Věta. Pokud Dijkstrův algoritmus provedeme na orientovaném ohodnoceném grafu s nezáporným ohodnocením hran, s počátečním vrcholem s, pak po ukončení algoritmu platí $d(u) = \delta(s, u)$ pro všechny vrcholy $u \in V$
Idea: $d(y) = \delta(s, y)$ pro vrchol vkládaný do D

ad Optimální podstrukt. (Bellmanův princip optimality):
- (platí a) využíva se v hladových alg. a dynamickém progr.
- stačí si pamatovat optimální hodnotu (neoptimální hodnoty a podstruktury se v řešení nevyskytují)
- optimální řešení lze zrekonstruovat (! odzadu; pomocí rovnosti cen)

Pro analýzu složitosti potřebujeme haldy:

Haldy

Datová struktura pro implementaci ADT prioritní fronta (ADT: Abstraktní Datový Typ)
- základní operace pro prioritní frontu:

- Vytvoř() vytvoří prázdnou množinu, tj. reprezentaci
- Přidej(M,x) přidá x do množiny M
- Min(M) vrátí ukazatel na prvek v M s minimálním klíčem
- OdeberMin(M) vrátí ukazatel jako Min(M) a navíc tento prvek odebere z M

- známá implementace: binární halda (v heapsortu)
- udržujeme invariant haldy ("haldovou podmínku"): vrchol má menší klíč než (obě) děti
- reprezentace v poli: vrchol i má 2 potomky na adrese $2i$ a $2i + 1$, kořen je na adrese 1
- reprezentace v binárním stromě s ukazateli (když se postaráme o vyvážení, tj. o logaritmickou hloubku)

Další operace pro haldu:

- SníženíKlíče(M,x,k) sníží hodnotu klíče prvku x v M na k
- Vymaž(M,x) odstraní prvek x z M
- Sjednocení($M1$,$M2$) vytvoří novou haldu pro množinu $M1 \cup M2$

- implementace operací (v binární haldě kromě Sjednocení): probubláváním zdola/shora (!při "shora"vyměňuji prvek za *menší* z dětí, používá se pouze v binární haldě (jinak slož. $\omega(\log n)$)
- složitost operací v binární haldě odpovídá hloubce $O(\log n)$ (protože máme konstantní (tj. shora omezený) počet dětí)
- postupná výroba haldy ("stačí"v heapsortu): $\Theta(n \log n)$, protože přidáváme $n/2$ prvků do hloubky $\Theta(\log n)$
- lépe: "dávková"výroba haldy z n prvků: v $O(n)$. (Odpovídá první fázi heapsortu.) Příklad výhodnosti *líné* implementace datové struktury. Idea: buduji malé haldy zdola.
- DC: Dokažte, že celkový počet operací je $O(n)$, tedy počet operací na 1 prvek je $O(1)$ (amortizovaná složitost). Počet operací je $\leq \sum_{i=1}^{h} i \cdot \frac{1}{2^i}$
- (dávkové zrušení haldy v $O(n)$ bez udržování invariantů, resp. přenechám garbage collectoru, např. v min. kostře)
- DC: udržování vyváženého tvaru (binární) pointrové haldy
- zařadím na správné místo (různé impl.), pak vybublám
- metapozn: Dobrá implementace/optimalizace funguje, i když ji neumíme dokázat (vůbec nebo těsně). (Porovnání vykonaných operací - jsou v jiném pořadí, měření)

(2022 Nepřednášeno: Binomiální stromy, binom. haldy ...)
- Binomiální stromy: B_0 je jeden vrchol, B_k se skládá ze dvou stromů B_{k-1}, strom s větším kořenem přivěšený pod menší. Číslo k nazýváme řád B_k.
- vlastnosti: B_k má právě 2^k vrcholů, počet synů (šířka) je maximálně $k = \log |B_k|$, hloubka je max. k. Odebráním

kořene (tj. OdeberMin) vzniknou binomiální stromy $B_0 ... B_{k-1}$ (v tomto pořadí).
- probubláváme pouze směrem nahoru: SníženíKlíče (DecreaseKey), Vymaž (Delete), čas $O(\log n)$
- Binomiální halda: binomiální stromy v seznamu (nebo poli), každý řád nejvýš jednou - dovoluje lib. počet prvků v haldě
- operace Sjednocení: spojuji stromy stejného řádu v pořadí od nejmenšího, čas $O(\log n)$, lze implementovat líně
- operace Přidej, OdeberMin se převedou na Sjednocení (Fibonacciho halda, idey)
- v DecreaseKey odebírám vrchol (s podstromy) v $O(1)$, protože mám nového kandidáta na minimum, tj. neprobublávám
- z podstromu nedovoluji odebrat příliš mnoho vrcholů (vhodným rychlým počítáním), případně snižuji řád, tím zaručuji exponenciální počet vrcholů vzhledem k řádu stromu
- při budování spojuji stromy stejného řádu a zvyšuji řád (jako v binomiálních h.)

- **Dijkstrův alg.: časová složitost:** $n = |V|, m = |E|$

operace:

OdeberMin se vykonává n-krát

SníženíKlíče (v rámci UvolněníHrany) se vykonává m-krát

$\rightarrow T(\text{Dijkstra}) = n.T(\text{OdeberMin}) + m.T(\text{Snížení Klíče})$

	T(OdeberMin)	T(SníženíKlíče)	T(Dijkstra)
pole	$O(n)$	$O(1)$	$O(n^2)$
binární halda (a binomiální)	$O(\log n)$	$O(\log n)$	$O(m.\log n)$
Fibonacciho halda	$O(\log n)$ (amortizovaná)	$O(1)$ (amortizovaná)	$O(n \log n + m)$

Amortizovaná složitost - idea: nepočítáme složitost každé operace samostatně, ale počítáme *nejhorší* případ posloupnosti operací. Při zachování správnosti a složitosti operací nemusíme vyžadovat (hladové) splnění invariantů po každé operaci, ale můžeme dovolit postupné zhoršování datové struktury a následně ji dávkově nebo postupně přebudovat. Přitom na náročné přebudování si našetříme kredit předchozími operacemi v posloupnosti, jen musíme zajistit, že kredit zůstává nezáporný.

S tímto principem se potkáme u hašování, konkrétně při změnách velikosti hašovací tabulky.

Bellmanův-Fordův algoritmus

Počítá nejkratší cesty z jednoho vrcholu v libovolně ohodnoceném grafu

Vstup: Orientovaný graf $G = (V, E)$
ohodnocení $c : E \to R$
počáteční vrchol $s \in V$

Výstup:
"NE" pokud G obsahuje záporný cyklus dosažitelný z s
"ANO", $d(v)$, $\pi(v)$ pro každé $v \in V$ jinak

Algoritmus:

```
 1 Inicializace(G,s)
 2 for i <- 1 to |V|-1 do  %pevný počet cyklů
 3   forall (u,v) in E do
 4     UvolněníHrany(u,v)
 5   od
 6 od
 7 forall (u,v) in E do  % závěrečná kontrola
 8   if d[v] > d[u] + c(u,v) then return "NE"
 9 od
10 return "ANO" % není záp. cyklus
```

- proč vadí (dosažitelné) záporné cykly, minimální cesta vs. minimální sled

- minimální vs. maximální cesta, pro nejdelší cesty neplatí Bellmanův princip optimality (!při reprezentaci "z u do v" bez mezilehlých vrcholů)

- časová složitost: $O(|V||E|)$

Lemma. Pro graf $G = (V, E)$ s počátečním vrcholem s a cenou $c : E \to R$, ve kterém není záporný cyklus dosažitelný z s, skončí alg. Bellman-Ford tak, že platí $d(v) = \delta(s, v)$ pro všechny vrcholy v dosažitelné z s.

Idea dk: Indukcí podle počtu vnějších cyklů. Po i-tém cyklu jsou minimální cesty délky i správně spočítány.

- pokud je v grafu záporný cyklus, neplatí trojúhelníková nerovnost pro cesty
- optimalizovaná impl.: znovuotevírání vrcholů při změně hodnoty, (generický alg. se strategií), expanze pouze otevřených vrcholů

Floydův-Warshallův algoritmus

Řeší verzi 3), nalézt $\delta(u, v)$ pro každé $u, v \in V$
Invariant: $\delta_k(i, j)$ je délka nejkratší cesty z i do j, jejíž všechny vnitřní vrcholy jsou v množině $\{1, 2 \ldots k\}$ (pro lib. pevné očíslování vrcholů)

$$\begin{aligned} \delta_k(i,j) &= c(i,j) \quad \text{pro } k = 0 \text{ (pouze přímé hrany)} \\ &= \min\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\} \text{ pro } k > 0 \end{aligned}$$

- Dokazujeme: 1) na začátku invariant platí, 2) po změně k invariant platí, 3) na konci invariant platí a (!)zahrnuje všechny cesty.

- !Past: k neodpovídá počtu hran budované cesty a indukce nejde podle počtu hran. (Tento "přímočarý"alg. má horší složitost a bude za chvíli)

Vstup:
orientovaný graf $G = (V, E)$
nezáporné ohodnocení hran $c : E \to R_0^+$ (v matici)

Výstup:
matice D, π, kde $D[i,j] = \delta(i,j)$ a $\pi[i,j]$ je předchůdce vrcholu j na nejkratší cestě z i do j

Algoritmus:

```
 1 for i <- 1 to n do
 2   for j <- 1 to n do
 3     pi[i,j] <- NIL
 4     if i=j then D[i,j] <- 0  % "smyčky"
 5       else if not (i,j) in E
 6         then D[i,j] <- infty  % hr. neexistuje
 7         else D[i,j] <- c(i,j) % c. jednohranová
 8              pi[i,j] <- i % poslední vrchol
 9     fi fi
10 od od
11 for k <- 1 to n do % přes mezilehlé v.
12   for i <- 1 to n do   % přes matici
13     for j <- 1 to n do
14       if D[i,k] + D[k,j] < D[i,j] then
15         D[i,j] <- D[i,k] + D[k,j]  % opt.
16         pi[i,j] <- pi[k,j]   % a odp. vrchol
17       fi
18 od od od.
```

- časová složitost: $O(n^3)$, paměťová $O(n^2)$
- implementačně: jedna matice D (místo dvou) pro všechny δ_k; každá hodnota v D má dosvědčující cestu
- adaptace při obecném (tj. i záporném) ohodnocení: dodatečný závěrečný test jako v alg. Bellman-Ford pro detekci

záporných cyklů
- záporné cykly se projeví jako záporné číslo na diagonále
- kompaktní reprezentace předchůdců v matici v $O(n^2)$ šetří paměť (naivně: paměť $O(n^3)$)
- F.-W. alg. je alg. dynamického programování, počítá se iterativně zdola a mezivýsledky se přepisují
- (AG: s podobným postupem se setkáte při převodu konečného automatu na regulární výraz: pracuje s konečnou reprezentací nekonečně mnoha posloupností - nelze postupovat indukcí "podle délky")

Algoritmy „násobení matic“ (pro všechny cesty, v.3)

- postupujeme indukcí podle počtu hran na nejkratší cestě

Definujme d_{ij}^k = minimální cena cesty z i do j s nejvýše k hranami

$d_{ii}^k = 0$

$k = 1$: $d_{ij}^1 = c(i,j)$ pokud hrana (i,j) existuje

$d_{ij}^1 = \infty$ jinak

ind. krok $k-1 \to k$: $d_{ij}^k = \min(d_{ij}^{k-1}, \min_{1\le l\le n}\{d_{il}^{k-1} + c(l,j)\}) = \min_{1\le l\le n}\{d_{il}^{k-1} + c(l,j)\}$ protože $c(j,j) = 0$

Hodnoty d_{ij}^k jsou v matici $D^{(k)}$, hodnoty $c(i,j)$ v matici C (vstupní). Počáteční (nebo koncová :-) podmínka $D^{(1)} = C$.

Potom $D^{(k+1)} = D^{(k)} \otimes C$, kde pro maticové násobení $\otimes$ používáme skalární součin, v němž je:
- násobení nahrazeno sčítáním a
- sčítání nahrazeno minimem

Pokud v G nejsou záporné cykly, potom je každá nejkratší cesta jednoduchá (tj. bez cyklů),

$\rightarrow$ každá nejkratší cesta má nejvýše $n-1$ hran
$\rightarrow D^{(n-1)} = D^{(n)} = D^{(n+1)} = \ldots = D$

Pomalá verze algoritmu: $n-2$ maticových násobení $\otimes$ matic řádu n
$\rightarrow$ složitost $(n-2).\Theta(n^3) = \Theta(n^4)$

Rychlá verze algoritmu: využijeme asociativitu operace $\otimes$ a počítáme pouze mocniny
$\rightarrow \lceil \log_2(n-2) \rceil$ násobení matic, celková složitost $\Theta(n^3 \log n)$

Pozn: Pro $\otimes$ nelze použít "rychlé"násobení matic (Strassenův alg. v $o(n^3)$)

Tranzitivní uzávěr grafu

Algoritmy lze adaptovat na tranzitivní uzávěr grafu.

Pro $G = (V, E)$ je $G^* = (V, E^*)$ *tranzitivní uzávěr* G, kde $E^* = \{(i,j) |$ existuje cesta z i do j v $G\}$.

Konstruovaná matice dosažitelnosti obsahuje boolovské hodnoty false a true (anebo 0/1, respektive) a používají se boolovské operace *and* a *or* (anebo obvyklý skalární součin s $*$ a $+$, respektive).

Impl. triky: počítáme s čísly modulo $n+1$, po fázích převádíme na 0/1, lze použít Strassenův alg. Pokud má (klasické) násobení matic slož. $\Theta(n^\alpha)$, pak tranz. uzávěr grafu má slož. $\Theta(n^\alpha \log(n))$.

Algoritmy pro všechny cesty lze získat n-násobným spuštěním algoritmů (pro každý vrchol) pro nejkratší cesty z 1 zdroje (Dijkstra, kritická cesta v DAG). Analogicky, tranzitivní uzávěr lze získat n-násobným spuštěním prohledávání z každého vrcholu.

Minimální kostra grafu

- Vstup: souvislý graf $G = (V, E)$ s hranovým ohodnocením $w : E \to R$

- kostra: podgraf T splňující $V(T) = V$, který je stromem

- minimální kostra: minimalizuje $w(T) = \sum_{e \in E(T)} w(e)$ mezi všemi kostrami

Algoritmus: Jarník 1930, Prim 1959
Vstup: souvislý graf, počáteční vrchol r
Výstup: hrany minimální kostry v E

```
 1 Q <- V(G) ; E <- {}
 2 forall u in Q
 3   do key[u] <- infty
 4 key[r] <- 0
 5 pi[r] <- NIL
 6 while Q <> 0
 7   do u <- OdeberMin(Q)
 7a    if u <> r
 7b      then E <- E union {(pi[u],u)}
 8     forall v in sousede(u)
 9       do if v in Q & w(u,v)<key[v]
10            then pi[v] <- u
11                 key[v] <- w(u,v) // (A)
12          fi
13   od  od.
```

Hrany E tvoří strom, který je podmnožinou min. kostry. Přidáváme (7b) nejlehčí hranu (7) mezi stromem a zbytkem vrcholů Q (6). Po přidání vrcholu u případně (9) upravíme

(10-11) jeho okolí (8) mimo strom (9), tj. v Q. Operace SníženíKlíče implementuje změnu klíče (11) označenou (A).

Suboptimální přímočará implementace má v haldě všechny hrany, které vedou z aktuálního stromu. Vybírá nejlacinější hranu z haldy, buď ji zahodí, pokud vede uvnitř stromu, nebo pomocí ní připojí další vrchol. Po připojení vrcholu přidá do haldy hrany z tohoto vrcholu. Vede na složitost $O(m \log n)$. (Což je totéž jako $O(m \log \underline{m})$ pro celkovou složitost operací na haldě max. velikosti m.)

Použitá implementace si pamatuje aktuální nejlepší možné napojení nepoužitých vrcholů na budovaný strom. Nejlehčí hrana se vybírá s využitím haldy Q, která obsahuje vrcholy mimo strom. Maximální velikost Q je n.

- časová složitost: $n = |V|$, $m = |E|$; Po n iteracích cyklu (6) se alg. zastaví a vydá kostru vstupního grafu.

- pro binární haldu: $O(m \log n)$ zahrnuje

1. postavení haldy $O(n)$, nebo postupně v $O(n \log n)$
2. OdeberMin $n.O(\log n)$
3. SníženíKlíče (A) $m.O(\log n)$

- zlepšení: Fibonacciho haldy $O(m + n \log n)$ kvůli

3. SníženíKlíče (A) $m.O(1)$ amortizovaně

- reprezentace v poli: $\Theta(n^2)$ zahrnuje

1. Inicializace pole $\Theta(n)$
2. OdeberMin $n.\Theta(n)$
3. SníženíKlíče (A) $m.O(1)$

- Pozn.: struktura operací je stejná jako u Dijkstrova algoritmu, ale klíče jsou jiné

Pozorování: Pro min. kostry platí princip optimality podobně jako pro min. cesty: každá podkostra min. kostry je minimální. (Pokud najdeme menší podkostru, můžeme jí část původní kostry nahradit. Po nahrazení máme zase kostru a ta má menší cenu. Spor s minimalitou původní kostry.)
Korektnost. Idea vybírání hran: Postupně přidáváme hrany do množiny $E(T)$ tak, že $E(T)$ je v každém okamžiku podmnožinou nějaké minimální kostry. To zaručuje věta o řezu, která je formulována obecně a nevyužívá konkrétní strategii (tj. konkrétní pořadí) výběru hran.

Df: Rozklad množiny vrcholů na dvě části $(S, V \setminus S)$ se nazývá řez. Hrana $(u, v) \in E$ *kříží* řez $(S, V \setminus S)$, pokud $|\{u, v\} \cap S| = 1$. Řez respektuje množinu hran A, pokud žádná hrana z A nekříží daný řez. Hrana křížící řez se nazývá *lehká hrana* (pro daný řez), pokud její váha je minimální ze všech hran křížících řez.

Df: Nech je množina hran A podmnožinou nějaké minimální kostry. Hrana $e \in E$ se nazývá *bezpečná* pro A, pokud také $A \cup \{e\}$ je podmnožinou nějaké minimální kostry.

Věta (o řezu): Nech $G = (V, E)$ je souvislý neorientovaný graf s váhovou funkcí $w : E \to R$, nech $A \subseteq E$ je podmnožinou nějaké minimální kostry a nech $(S, V \setminus S)$ je libovolný řez, který respektuje A. Potom pokud je hrana $(u, v) \in E$ lehká pro řez $(S, V \setminus S)$, tak je bezpečná pro A.

Dk: Nech T je min. kostra, která obsahuje A a neobsahuje lehkou (u, v). Zkonstruujme min. kostru T', která obsahuje $A \cup \{(u, v)\}$.

Hrana (u, v) v T uzavírá cyklus. Vrcholy u a v jsou v opačných stranách řezu $(S, V \setminus S)$, proto aspoň jedna hrana v T kříží řez, označme ji (x, y). Platí $(x, y) \notin A$, protože řez respektuje A. Odstranění (x, y) z T kostru rozdělí na dvě komponenty a přidání (u, v) ji opět spojí a vytvoří $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$.

Protože (u, v) je lehká pro $(S, V \setminus S)$ a (x, y) kříží tento řez, platí $w(u, v) \leq w(x, y)$, proto $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$, odkud T' je min. kostra. Protože $A \cup \{(u, v)\} \subseteq T'$, je (u, v) bezpečná pro A.

Důsl: Nech $G = (V, E)$ je souvislý orientovaný graf s váhovou funkcí $w : E \to R$, nech $A \subseteq E$ je podmnožinou nějaké minimální kostry a nech C je souvislá komponenta (strom) podgrafu zadaného množinou A. Pokud je $(u, v) \in E$ hrana s minimální váhou spojující C s jinými komponentami grafu G_A zadaného množinou A, potom (u, v) je bezpečná pro A.

Dk: Řez $(C, V \setminus C)$ respektuje A a (u, v) je lehká hrana pro tento řez.

- pokud jsou ceny hran různé, je jedna minimální kostra.
- Zdůvodnění: každý řez má jednoznačně určenou hranu, která je součásti kostry.
- V důkazu výše nemůžeme odebranou hranu nahradit stejně ohodnocenou hranou, proto jiná min. kostra neexistuje.

- první alg.: Otakar Borůvka 1926, pro různé ceny hran:

Algoritmus: Borůvka 1926

Vstup: souvislý graf G s různými vahami

Výstup: minimální kostra T
1 $T \leftarrow (V(G), \emptyset)$
2 Dokud T není souvislý
3 Rozložíme T na komponenty $T_1, \ldots, T_k$
4 Pro každý strom T_i najdeme nejlehčí hranu e_i mezi T_i a zbytkem grafu
5 Přidáme do T hrany $e_1, \ldots, e_k$

V: Algoritmus se zastaví po $\lfloor \log_2 n \rfloor$ iteracích a vydá minimální kostru.

Dk: Po k iteracích mají stromy velikost aspoň 2^k. Na začátku ($k = 0$) jsou stromy jednovrcholové ($1 = 2^0$), v každé iteraci se spojí stromy aspoň do dvojic. Nejpozději po $\lfloor \log_2 n \rfloor$ iteracích strom obsahuje všechny vrcholy, je souvislý a alg. se zastaví.
Každá přidaná hrana e_i je nejlehčí hrana řezu mezi T_i a zbytkem grafu. Všechny vybrané hrany patří do jediné min. kostry. Na konci je strom souvislý, proto máme celou kostru.

- pro neunikátní váhy hran může sestrojený T obsahovat cyklus.
- graf s neunikátními váhami můžeme zunikátnit, když k hranám přidáme druhotné kriterium (pro lexikografické uspořádání), např. čísla vrcholů

- Implementace a složitost:
- počet iterací je $O(\log n)$
- v jedné iteraci rozklad na komponenty zvládneme pomocí DFS v $O(m)$, potom každou hranu ($O(m)$) přidáme do průběžného minima obou komponent ($O(1)$, když vrchol určuje

svojí komponentu). Nakonec vybrané hrany přidáme do kostry ($O(n)$).
- celkem: $O(m \log n)$

- Pozn.: - Hledání minimální kostry je příklad hladového algoritmu.
- Hladový alg.: Lokálně optimální rozhodnutí (zde volba bezpečné hrany) vede ke globálnímu optimu. Mnoho problémů *nemá* tuto vlastnost a hladový algoritmus nelze použít pro nalezení glob. optima. (Ale někdy se volba lok. optima používá jako *hladová heuristika*.)
- obecnější metoda než hladový alg. je dynamické programování (platí v něm princip optimality: optimální řešení se skládá pouze z optimálních podřešení)
- tato vlastnost neplatí pro problémy obecně, speciálně NP-těžké problémy (budou v ADS2): řešení prohledáváním všech možností, typicky (chytře, ale) backtrackingem
- (řešení pomocí backtrackingu – bez využití hladového výběru, pro srovnání: najdeme (rekurzivně) optimální řešení se zvolenou hranou/vrcholem a bez ní a z těchto dvou řešení vybereme lepší; zkoumaný vrchol stačí napojit pouze minimální hranou (díky principu optimality), ale musíme zkoušet všechny vrcholy v daném stavu)
- jiné příklady hlad. alg.: stavba stromu pro Huffmanovo kódování, rozvrhnutí max. počtu úloh na 1 stroj, problém batohu při dělitelných předmětech ...
- v problému nejdelší cesty (nebo batohu, tj. součtu podmnožiny) lze použít dynamické programování, ale na expo-

nenciálně (tj. nepolynomiálně) větším prostoru stavů
- teorie pro hladové algoritmy: (vážené) matroidy

(2020 Nepřednášeno: Kruskalův alg. a dat. struktura Union-Find)

Algoritmus: Kruskal 1956

1. uspořádej hrany z E tak, aby $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
2. $E(T) \leftarrow \emptyset; i \leftarrow 1$
3. `while` $E(T) < |V| - 1$ `do`
4. `if` $E(T) \cup \{e_i\}$ neobsahuje kružnici
5 `then` přidej e_i do $E(T)$ `fi`
6 `i++`
7 `od`

Korektnost. Idea vybírání hran: Postupně přidáváme hrany do množiny $E(T)$ tak, že $E(T)$ je v každém okamžiku podmnožinou nějaké minimální kostry. S využitím řezového lematu.

- časová složitost: $(n = |V|, m = |E|)$
- třídění hran v $\Theta(m \log m)$ $(= \Theta(m \log n))$
- zpracování hrany při vhodné reprezentaci $O(\log m)$, pomocí Union-Find struktury, která reprezentuje faktorovou množinu komponent

Datová struktura Union-Find: Ke každé komponentě si pamatujeme reprezentanta (vrchol) – pro všechny vrcholy komponenty vracíme ten samý
- implementace, dvě možnosti:
1. v poli velikosti $|V|$: každý vrchol ukazuje přímo na repre-

zentanta.

Najdi(u) v $O(1)$, $O(m)$-krát

Sjednocení(u,v) v $O(n)$, $O(n)$-krát

celkem: $O(n^2)$

2. pomocí pointrů (v pointrové struktuře nebo poli): Reprezentanti jsou kořeny stromů, pointer vede směrem ke kořeni.
Operace: Sjednocení(u,v): při přidávaní hrany spojujeme komponenty, vykonává se $n-1$-krát, změna d.s. v $O(1)$
implementace Sjednocení:

`r`$_u$ `<- najdi(u)`

`r`$_v$ `<- najdi(v)`

`reprezentant(r`$_u$ `) <- r`$_v$ (1)

operace Najdi(u): projde vrcholy od u nahoru
- při testování hrany zjišťujeme, zda reprezentanti koncových vrcholů hrany jsou stejní (pokud ano, konce jsou ve stejné komponentě a hrana by tvořila cyklus, tj. nepřidáme ji)

- počet volání: $2\times$ pro každou testovanou hranu, tj. $O(m)$

- složitost operace: rovna hloubce stromu reprezentantů

- pokud v (1) připojujeme menší komponentu k větší, dostaneme hloubku $O(\log k)$, kde k ($\leq n$) je velikost komponenty; široký strom nevadí

$\rightarrow$ celková složitost $O(m \log n)$

- (impl. trik: Nepamatujeme si konkrétní velikost komponent, ale pouze rank r: odpovídá (původní největší) hloubce stromu a zaručuje aspoň 2^r vrcholů, rank zvýšíme o 1, pokud spojujeme komponenty stejného ranku, jinak menší přivěsíme k větší. !Pořadí spojování si nevybíráme, ale směr

pointru lze zvolit.)

- (impl: Zkracování cest (při Najdi, několik variant). Vede na "skoro lineární"algoritmus $\Theta(m \cdot f(n))$, kde $f(n)$ je velmi pomalu rostoucí funkce.)

Dynamické množiny

dynamické - mění se v čase (obsah, velikost, ...)

prvek dynamické množiny je přístupný přes ukazatel (pointer) a obsahuje

1. klíč - typicky z nějaké lineárně uspořádané množiny
2. ukazatel(e) na další prvky (nebo části reprezentující struktury)
3. další (uživatelská) data (!)

Rozlišujeme dat. strukturu Slovník, kde ke klíči máme data (3), a jednodušší d.s. Množina, kde máme pouze klíče (bez 3).

Operace na dynamické množině: Nechť S je dynamická množina prvků, k hodnota klíče a x ukazatel na prvek:

- $Find(S, k)$ - vrací ukazatel na prvek s klíčem k v množině S anebo NIL
- $Insert(S, x)$ - do S vloží prvek, na který ukazuje x
- $Delete(S, x)$ - z S odstraní prvek, na který ukazuje x
- $Min(S)$ - vrací ukazatel na prvek v S s min. klíčem
- $Max(S)$ - vrací ukazatel na prvek v S s max. klíčem
- $Succ(S, x)$ - vrací ukazatel na prvek v S bezprostředně následující (v lineárním uspořádání klíčů) po prvku, na který ukazuje x, anebo vrací NIL
- $Predec(S, x)$ - analogicky pro bezprostředně předcházející prvek k x

Binární vyhledávací stromy (BVS)

Dynamická datová struktura, která podporuje všechny operace na dynamické množině

Binární strom: každý vrchol reprezentuje jeden prvek množiny a obsahuje klíč a 3 pointry na levého syna (levý), pravého syna (pravý) a rodiče (rodič)

- strukturu binárního stromu můžeme použít pro jiné datové struktury, např. pro binární haldu (pro heapsort). V ní je klíč vrcholu menší než oba potomci.

Pro binární vyhledávací strom platí:
pro každý prvek x platí: všechny prvky v levém podstromě prvku x mají menší klíč než x (připouštíme rovnost) a všechny prvky v pravém podstromě prvku x mají vetší klíč než x.

```
Find(x,k) % x je ukazatel na kořen BVS obsahující S
 1 while (x<>NIL) and (k<>klíč(x)) do  % k je pod x
 2   if k < klíč(x)   % zúžení výběru
 3     then x <- levý(x)  % posun doleva
 4     else x <- pravý(x) % posun doprava
 5 return x       % x=NIL or k=klíč(x)
```

Složitost je $O(h)$, kde h je výška BVS. Na každé úrovni stromu spotřebujeme konstantní čas, tj. $O(1)$. Stejná složitost platí i pro ostatní operace.

- další operace (vyhledávací i modifikující):

Min(S): Od kořene doleva, dokud to jde. Prvek, který nemá levého syna, je nejmenší.

Max(S): symetricky.

Succ(S,x): Do pravého syna, pokud existuje, pak opakovaně doleva. Jinak opakovaně přecházíme do rodiče a vracíme prvního rodiče, kde opouštěný syn je vlevo, anebo NIL, pokud dojdeme do kořene.
Lokálně, v pravém podstromě hledáme Min(), ale tato operace je zavedena jen pro celou strukturu S.
Pozn. Musíme ošetřit všechny možné případy.

Predec(S,x): symetricky

Insert(S,x): Klesáme podle porovnání doleva nebo doprava. Prvek přidáme místo NIL jako list.

Delete(S,x): Pokud má vypouštěný prvek x 0 nebo 1 syna, vypustíme přímo. Jinak najdeme $s = Succ(S, x)$, kterým nahradíme x (!přelinkováním, ne kopírováním) a vypouštíme prvek z pozice s.

BVS s n vrcholy:

- nejlepší případ: vyvážený (úplný) strom: výška $\Theta(\log n)$; ale: udržování globálního (výškového nebo vrcholového) vyvážení je drahé
- nejhorší případ: lineární seznam n vrcholů: výška $\Theta(n)$
- náhodně postavený strom (průměrný případ): výška $\Theta(\log n)$

Chceme zlepšit nejhorší případ: Červeno-černé stromy a AVL stromy pomocí lokálních invariantů a lokálních vyvažovacích operací zaručí (globální) výšku $\Theta(\log n)$ v nejhorším případě.

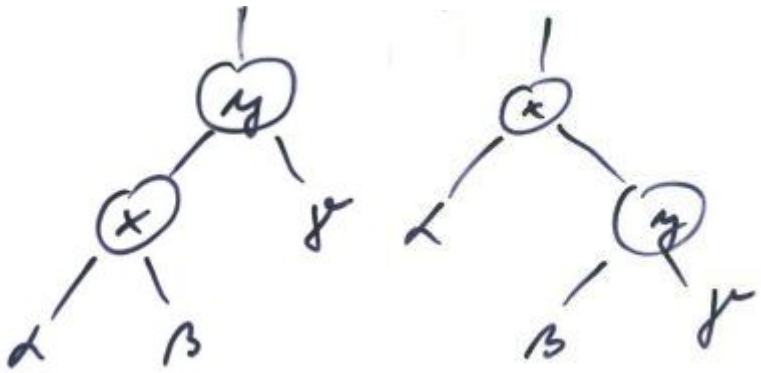
Rotace

Pro odstraňování poruchy se používají rotace, které situaci lokálně vyřeší a/nebo přesunou poruchu výš. Mějme situace:

1. $((\alpha, X, \beta), Y, \gamma)$
2. $(\alpha, X, (\beta, Y, \gamma))$

PraváRotace(Y) převede 1. na 2., LeváRotace(X) převede 2. na 1.

Argument rotace je kořen podstromu, který se mění.



Obrázek 1: Pravá rotace: strom před a po

Pozorování: uspořádání klíčů zůstalo při obou rotacích zachováno.

Pro jednotlivé transformace budeme kontrolovat zachování pořadí klíčů a zachování (lokální) hloubky podstromů.

Hackerská školka: (Neoptimální) implementace rotace (v OOP): Zapamatujeme si nejdřív vše, co budeme potřebovat (pro pravou rotaci: rodič(y), y, x, kořen(β)), do pomocných proměnných a potom změníme příslušné položky/ukazatele (6x, pro 3 hrany).

- pozn: Jsou programovací jazyky (Prolog, Haskell), které dovolují zapsat vlastní rotaci (jen výkonná část, bez testů):

```
RotDoprava( t(t(TA,X,TB),Y,TG), t(TA,X,t(TB,Y,TG)) ).
RotDoprava (T (T ta x tb) y tg) = T ta x (T tb y tg)
```

AVL stromy

Df (Adelson-Velskij, Landis): Binární vyhledávací strom je AVL strom (vyvážený AVL), právě když pro každý vrchol x ve stromě platí
$|h(levy(x)) - h(pravy(x))| \leq 1$,
kde $h(x)$ je výška (pod)stromu (počítáme úrovně hran).

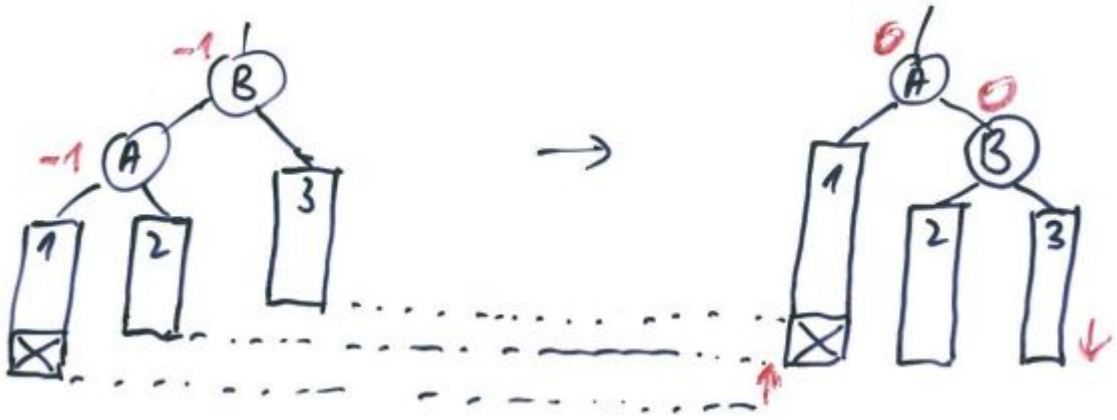
- pro efektivitu operací si pamatujeme explicitně (v položce vrcholu) aktuální vyvážení: v z množiny $\{-1, 0, +1\}$, kde $v = h(pravy) - h(levy)$

Věta: Výška AVL stromu s n vrcholy je $O(\log n)$.
Idea Dk: Konstruujeme nejnevyváženější strom, s nejméně vrcholy při dané výšce: označme p_n počet vrcholů takového stromu T_n s hloubkou n.
$T_0 : p_0 = 1$
$T_1 : p_1 = 2$
$T_n : p_n = p_{n-1} + p_{n-2} + 1 = fibonacci_{n+3} - 1$, (pro $fib_3 = 2$)
Pozn: $fibonacci_n \approx \varphi^n/\sqrt{5} \approx \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$, kde $\varphi \approx 1.618$ je zlatý řez

Důsledek: Všechny dotazovací operace pro BVS (Find, Min, Max, Succ, Predec) fungují na AVL stromě stejně a mají složitost $O(\log n)$

Zůstávají modifikující operace Insert, Delete.
Operace pracují stejně, po změně upravujeme zdola vyvážení, s případnou propagací změny nahoru (pokud to stačí) a případně použijeme rotace (jednoduchou, dvojitou)
- rotace je časově náročnější než úprava vyvážení, ale pořád $O(1)$

Jednoduchá rotace:



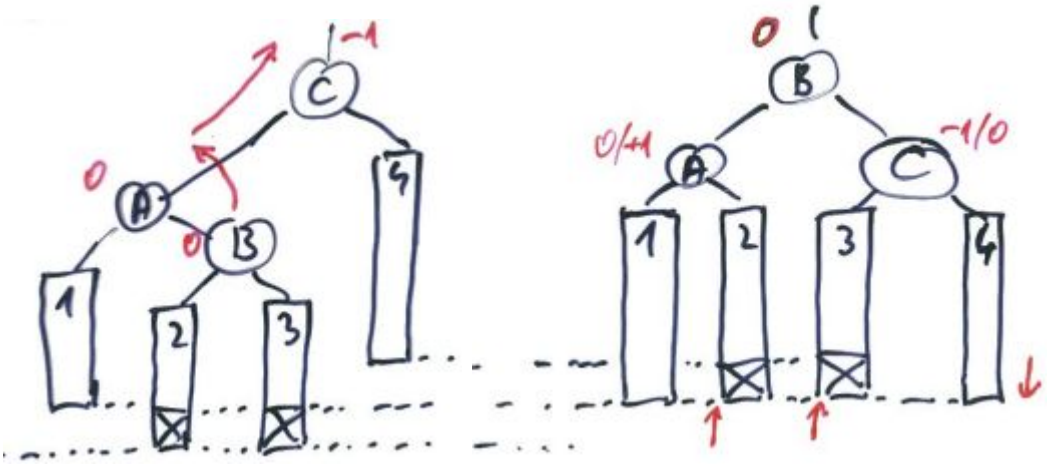
Obrázek 2: AVL: Přidáváme X do vnějšího podstromu, před a po

((T1,A/-1,T2),B/-1,T3) → (T1,A/0,(T2,B/0,T3))
- přidáváme prvek X do stromu T1 (přesněji: hloubka T1 rostla)
- po rotaci se do předků nového A nešíří změna, protože původní hloubka zůstala zachována (T3 měl "rezervu")
- uspořádání zůstalo: $T1 < A < T2 < B < T3$

Neoptimální implementace rotace: zapamatujeme si ukazatele na "objekty"včetně rodiče B a potom nastavíme nové hodnoty pro jednotlivé změněné položky

Dvojitá rotace: ((T1,A/0,(T2,B/0,T3)),C/-1,T4) → ((T1,A,T2),B/0,(T3,C,T4))
- přidáváme prvek X do T2 nebo T3, podle toho určíme vyvážení v A a C po rotaci
- po rotaci se do předků B opět nešíří změna
- uspořádání zůstalo

- dvojitou rotaci bereme jako jeden celek a invarianty popisujeme před a po provedení, ne v mezistavu. Implementačně: lze použít dvě jednoduché rotace



Obrázek 3: AVL: Přidáváme (jedno z) X do vnitřního podstromu, před a po

- analogicky symetrické případy

Rozbor případů vyvážení při Insert: (BÚNO, přidávalo se do levého podstromu, tj. zvyšovala se hloubka vlevo)

1) +1 na 0, konec (využila se rezerva vlevo)

2) 0 na -1, propagace zvýšení hloubky nahoru

3) -1, přidáváme do levého levého podstromu: jednoduchá rotace, konec

4) -1, přidáváme do pravého levého podstromu: dvojitá rotace, konec

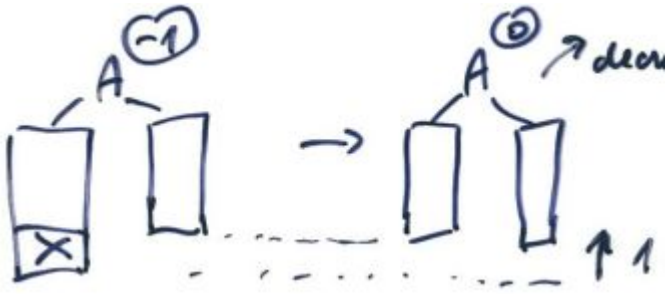
V případech 3) a 4) je nový strom stejně hluboký jako původní, proto nemusíme propagovat změnu nahoru.

Rezerva pro vkládání je vhodná nevyváženost v kořeni.

Operace Delete, rozbor případů pro předky fyzicky vypouštěného vrcholu:

- ubíráme vrchol, tj. snižujeme výšku, BÚNO vlevo. Při propagaci nahoru se snížila výška celého stromu a musíme upravovat na cestě ke kořeni.

1) -1 to 0, propagace snižování

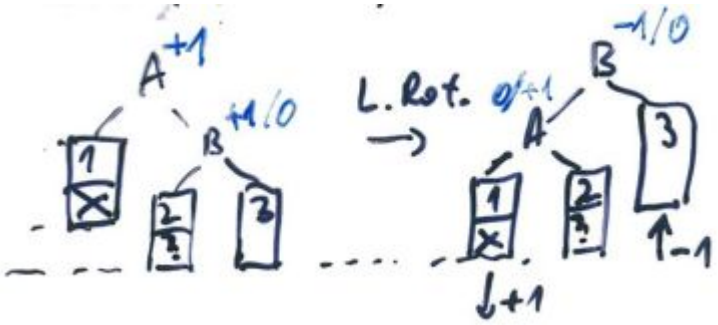


Obrázek 4: AVL: Delete 1: vypouštíme X, propagace nahoru, před a po

2) 0 to +1, konec

Obrázek 5: AVL: Delete 2: vypouštíme X, lokální oprava, před a po

3) +1 v otci, +1 v bratrovi: jednoduchá L-rotace, propagace nahoru (vrchol '?' neexistuje)

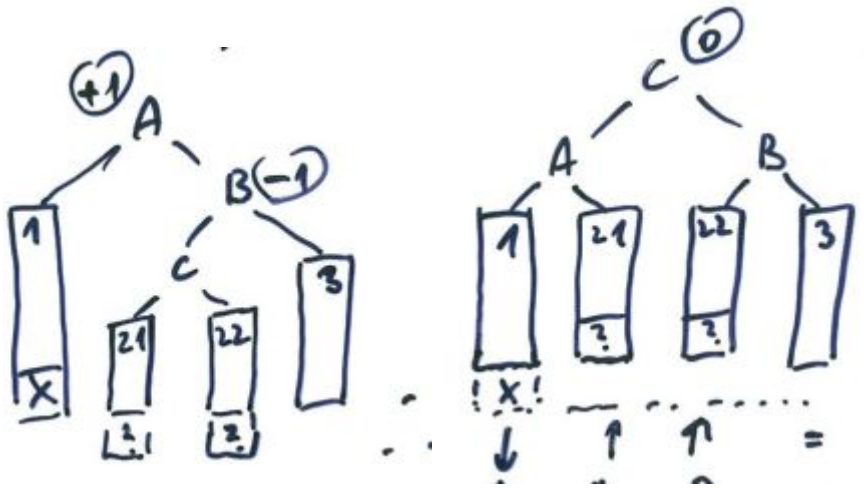


Obrázek 6: AVL: Delete 3,4: vypouštíme X, před a po

4) +1 v otci, 0 v bratrovi: jednoduchá L-rotace, bez propagace (vrchol '?' existuje)

5) +1 v otci, -1 v bratrovi: dvojitá rotace, propagace nahoru (aspoň jeden vrchol '?' existuje)

- Při Delete nezaručíme O(1) rotací jako v Č-Č stromech.



Obrázek 7: AVL: Delete 5: vypouštíme X, před a po

- Možná implementace: líné operace (bude na pokročilejších přednáškách): vyvažování a propagaci nemusíme vykonat hned, potřebné informace pro změny si zapamatujeme a např. samostatné vlákno (s nízkou prioritou) vykoná opravy a rotace, až bude čas.

Červeno-černé stromy

- Jsou to binární vyhledávací stromy, které mají navíc obarvené vrcholy: barva je červená nebo černá. (Implementačně: 1 bit)

- Z podmínek na obarvení plyne, že délka cest z kořene do listů se liší nejvíc 2x. Stromy jsou vyvážené v tom smyslu, že nejdelší cesta je logaritmická vůči počtu vrcholů.

Df: BVS je *červeno-černý strom*, pokud splňuje následující vlastnosti:

1. Každý vrchol je červený nebo černý
2. Každý list (Nil, tj. externí vrchol) je černý
3. Pokud je vrchol červený, potom obě děti jsou černé
4. Každá cesta z vrcholu do podřízeného listu obsahuje stejný počet černých vrcholů

Vrcholy, které nejsou externí, jsou vnitřní, tj. interní. Červené vrcholy mají (případně externí) černé syny. Z 3. plyne, že nejsou dva červené vrcholy bezprostředně pod sebou. Následně ze 4. plyne speciálně pro kořen, že při zvolené černé výšce je nejkratší možná cesta z kořene do listu pouze černá a nejdelší jen 2x delší (střídavě červené a černé vrcholy).

Značení: $bh(x)$ je černá výška (black height) - počet černých vrcholů na cestě z x do listů, kromě x. (Podle 4. na volbě listu nezáleží.)

Lemma: Č-č strom s n vnitřními vrcholy má výšku h nejvíc $2\log(n+1)$.

Dk: Indukcí podle výšky podstromů lze dokázat, že podstrom ve vrcholu x má aspoň $2^{bh(x)} - 1$ vnitřních vrcholů.

Použijeme pro kořen:
$n \geq 2^{h/2} - 1$, odtud $h \leq 2\log(n+1)$

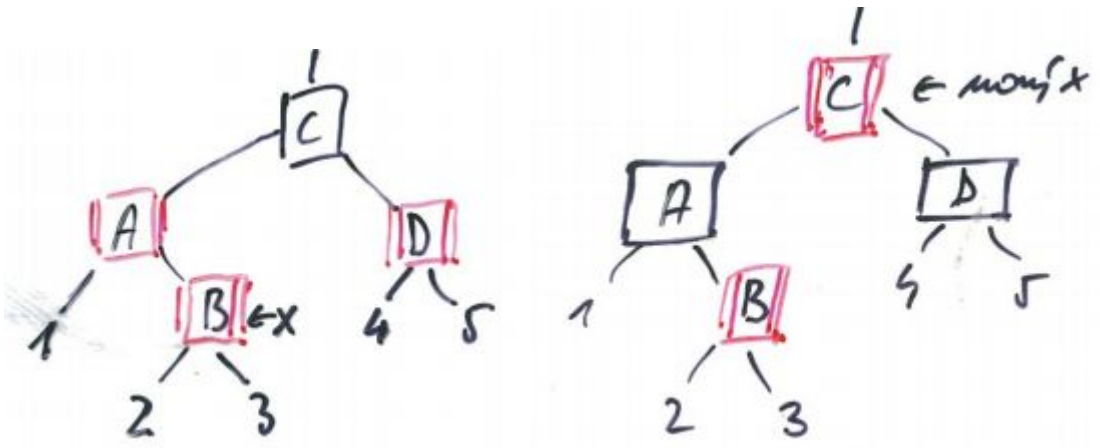
Vkládání: viz Obrázky

Červený kořen můžeme přebarvit na černý bez porušení vlastností Č-Č stromu. Budeme tuto vlastnost udržovat.

Fyzické vložení uzlu: jako v BVS, vložený uzel x obarvíme na červeno.

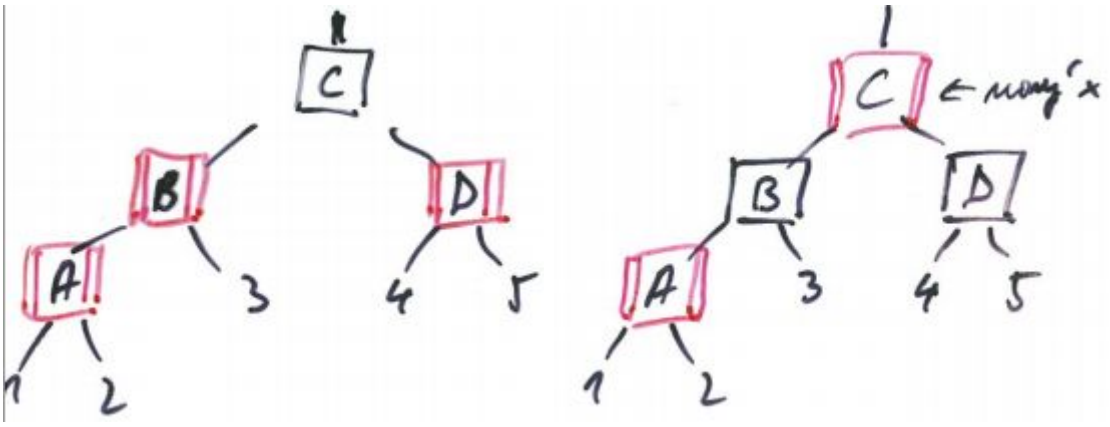
Pak může být porušena pouze vlastnost 3, když vložený uzel x i jeho otec y jsou červené. Pokud tato porucha nastane, y má otce z , který je černý. Jinak končíme, strom je správně utvořený. Bratra y označíme $y2$.

Rozeberme 3 možné případy.



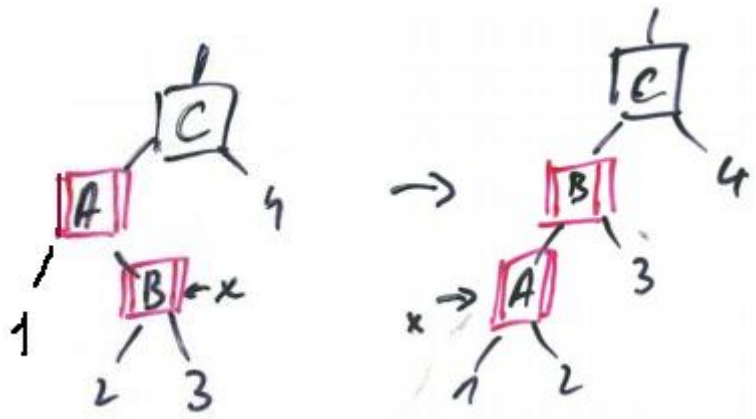
Obrázek 8: Č–Č případ 1a, $x = B$: Jeho strýc D je červený, před a po

1. Zahrnuje 1a, resp. 1b. Bratr $y2 = D$ uzlu $y = A$, resp. $y = B$, tj. strýc x, je červený. Pak y a $y2$ přebarvíme na černo, $z = C$ na červeno. Pokud je z kořen, přebarvíme ho na černo a končíme, pokud má z černého otce, končíme,



Obrázek 9: Č–Č případ 1b, $x = A$: Jeho strýc D je červený, před a po

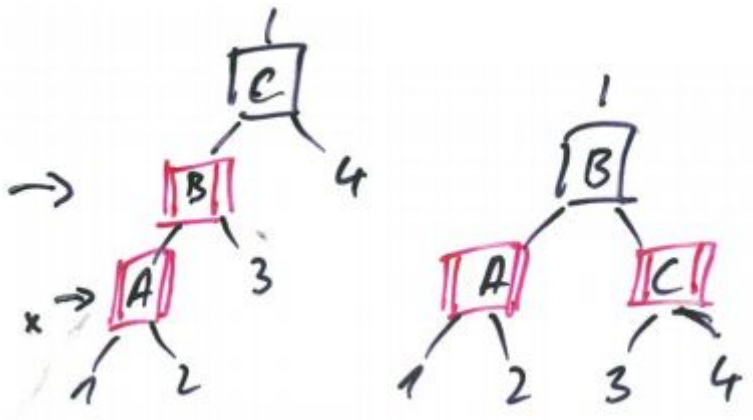
jinak má z červeného otce, porucha se přesunula výš a iterujeme (rozborem 3 možností). Uzel z je polohou "nový" x.



Obrázek 10: Č–Č případ 2: Strýc 4 je černý, $x = B$ je vnitřní (před a po)

2. Bratr $y2$ uzlu y je černý a x je opačným synem y než y synem z. Pokud je x pravým synem y, tak LeváRotace(y), jinak PraváRotace(y). (Bez přebarvování.) Tím je situace převedena na případ 3.

3. Bratr $y2$ uzlu y je černý a x je stejným synem y než y synem z. Pokud je x levým synem y a y levým synem z, tak PraváRotace(z) a přebarvení y na černo a z na



Obrázek 11: Č–Č případ 3: Strýc 4 je černý, $x = A$ je vnější (před a po)

červeno. Tím jsou vlastnosti Č-Č stromu splněny. Opačný případ je symetrický.

Pozn: V případě vkládání dva černé uzly pod sebou tvoří "rezervu", kterou využijeme lokálně a poruchu nemusíme propagovat nahoru.

Složitost vkládání je $O(\log n)$.
- vlastní vkládání do BVS $O(\log n)$
- složitost 1. je $O(1)$ (test+rotace+přebarvení), provede se nejvýše $O(\log n)$-krát
- složitost 2. a 3. je $O(1)$, provede se každá nejvýše jednou

Vypouštění:

Prvek odstraníme standardním způsobem z BVS.

Pro vypouštění, červený uzel v lokálním okolí je rezerva, která umožní změnu nepropagovat nahoru.

Skutečně odstraňovaný prvek y má nejvýše jednoho interního syna, nechť je to x. Pokud nemá interní syny, označíme x jednoho z externích synů. (Uzel x se dostane na místo y.)

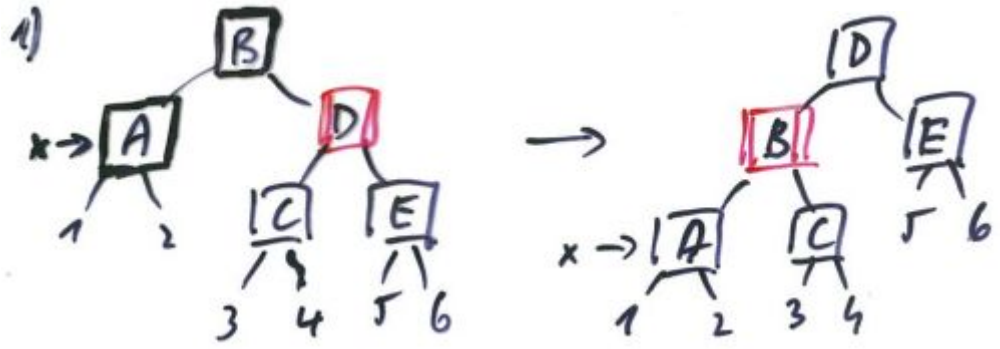
Pokud je y červený, po jeho vypuštění je strom v pořádku. Pokud je y černý, je po jeho vypuštění porušena vlastnost 4 (kromě případu, když je y kořen), protože cesty vedoucí přes y ztratily 1 ze své černé výšky.

Pokud je x červený , přebarvíme ho na černo a strom je v pořádku.

Pokud je x černý, označíme ho jako dvojitě černý a tuto (jedinou) poruchu budeme odstraňovat, buď na místě nebo posunem vzhůru.

Pokud je x kořen stromu, tak černou barvu navíc odstraníme a černá výška všech vrcholů klesne o 1. Pokud x není kořen, tak rodič(x), označme ho z, musí mít druhého syna interního, označme ho w, jinak by cesty k listům nesplňovaly podmínky na černou výšku. (Externí uzel má černou výšku 1, tedy menší než x.)

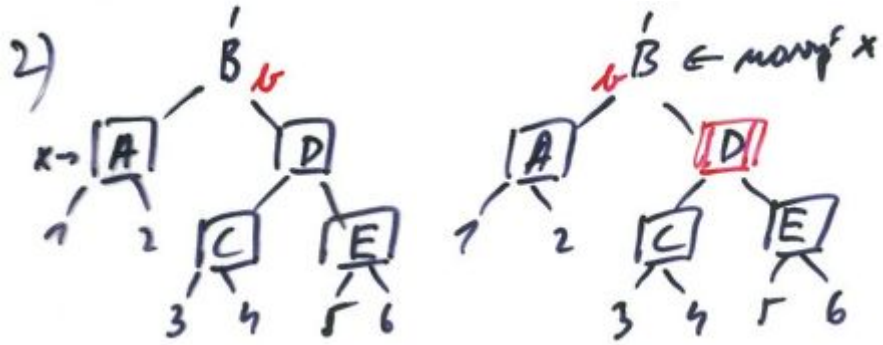
Budeme předpokládat, že x je levým synem rodiče z (opačný případ je symetrický) a rozlišíme čtyři případy podle barvy w a jeho synů.



Obrázek 12: Č–Č případ 1: Bratr D je červený, před a po

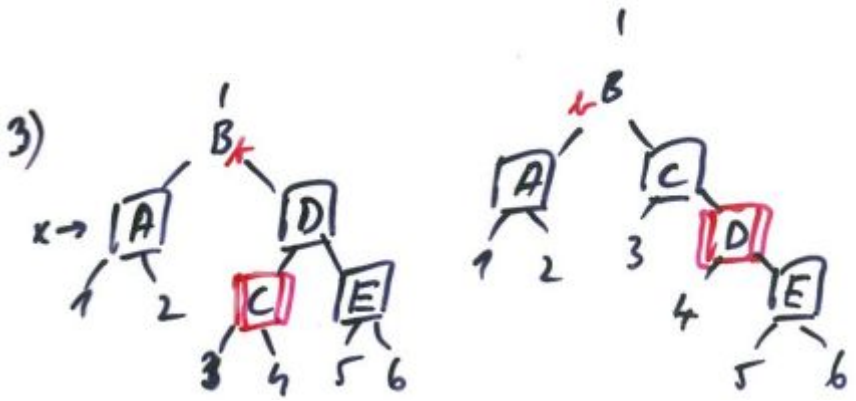
1. uzel w je červený (a tedy má černé syny). Přebarvíme w na černo a z (=rodič(x)=rodič(w)) na červeno a prove-

deme LeváRotace(z). Tím se situace převede na jednu z dalších 3 možností, tj. x má černého bratra.



Obrázek 13: Č–Č případ 2: Bratr D, C a E jsou černé, před a po

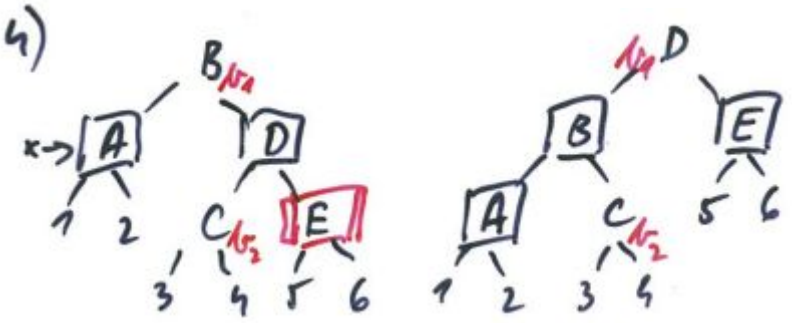
2. uzel w je černý a má dva černé syny. Odstraníme jednu černou z x a přebarvíme w na červeno. Jejich otci z přidáme jednu černou, tj. pokud je červený, přebarvíme na černo a končíme, a pokud je z černý, změníme ho na dvojitě černý. Tím jsme poruchu posunuli ke kořeni a iterujeme. Tento postup dvojitě černé vzhůru se zastaví nejpozději v kořeni, kde jednu černou můžeme odebrat.



Obrázek 14: Č–Č případ 3: Bratr D a vnější syn E černý, vnitřní syn C červený, před a po

3. uzel w je černý, jeho levý ("vnitřní") syn je červený a pravý syn černý. Vyměníme barvy w a jeho levého syna

a provedeme PraváRotace(w). Tím převedeme situaci na případ 4.



Obrázek 15: Č–Č případ 4: Bratr D je černý, jeho vnější syn E červený, před a po

4. uzel w je černý a jeho pravý syn je červený. Pravého syna uzlu w přebarvíme na černo a odebereme černou z uzlu x. Pokud byl z, tj. rodič x červený, tak ho přebarvíme na černo a w na červeno. Provedeme LeváRotace(z).

Pozn: V případě vypouštění červené uzly (případy 3 a 4) tvoří rezervu, kterou můžeme využít lokálně. Pokud máme "v okolí" černé uzly (případ 2), musíme posouvat poruchu výš.

Složitost vypouštění je $O(\log n)$. Vlastní vypouštění (včetně přesunu listu) je $O(\log n)$. Při odstraňování poruchy všechny testy a akce trvají $O(1)$ a případy 1, 3 a 4 se vykonají jednou a případ 2 nejvýš $O(\log n)$-krát.

Pozn.: Popsané změny vykonáváme najednou. Při implementaci můžete samostatně rotovat a přebarvovat, ale pro účely vysvětlování a důkazu používáme pouze zobrazené stavy.

B-stromy

- B-stromy jsou vyvážené vyhledávací stromy

- jsou s proměnlivým počtem následníků, větším než 2: vrchol x s $n(x)$ klíči má $n(x)+1$ dětí

- aplikace: data na disku, indexové soubory databází

$\rightarrow$ přístup na disk je časově náročnější a diskové stránky se načítají celé, proto chceme menší hloubku za cenu většího počtu dětí a nevyužívání paměti (použitelné i pro bloky keše v operační paměti, obecně paměťovou hierarchii)

B-strom má následující vlastnosti:

1. klíče jsou uloženy v neklesající posloupnosti (zleva doprava)
2. pokud je x vnitřní vrchol s $n(x)$ klíči, pak obsahuje $n(x)+1$ pointrů na děti
3. klíče ve vnitřním vrcholu rozdělují intervaly klíčů v podstromech
4. každý list je ve stejné hloubce
5. pro nějaké pevné t platí, $t \geq 2$ (tzv. minimální stupeň)
   (a) každý vrchol kromě kořene má aspoň $t-1$ klíčů. Každý vnitřní vrchol kromě kořene má aspoň t dětí. Pokud je strom neprázdný, má kořen aspoň 1 klíč.
   (b) každý vrchol má nejvíc $2t-1$ klíčů, tedy nejvíc $2t$ dětí.

- pozn. k literatuře: Jsou různé varianty B-stromů (resp. a-b stromů). My používáme: s hodnotami ve vnitřních vrcho-

lech (vs. pouze v listech), s přípravným štěpením a slučováním: tj. počtem dětí p: $t \leq p \leq 2t$ (vs. $t-1 \leq p \leq 2t-1$)

Tvrzení: Pro $n \geq 1$ a každý B-strom T s n klíči, výškou h a minimálním stupněm $t \geq 2$ platí:
$h \leq \log_t \frac{n+1}{2}$
Dk: Pro strom dané výšky h je počet vrcholů minimální, když kořen obsahuje 1 klíč a ostatní vrcholy $t-1$ klíčů. Pak jsou 2 vrcholy v hloubce 1, $2t$ vrcholů v hloubce 2, $2t^2$ vrcholů v hloubce 3 atd., až $2t^{h-1}$ vrcholů v hloubce h.

Počet klíčů n splňuje:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$= 1 + 2(t-1)(\frac{t^h-1}{t-1})$
$= 2t^h - 1$
a odtud plyne tvrzení.

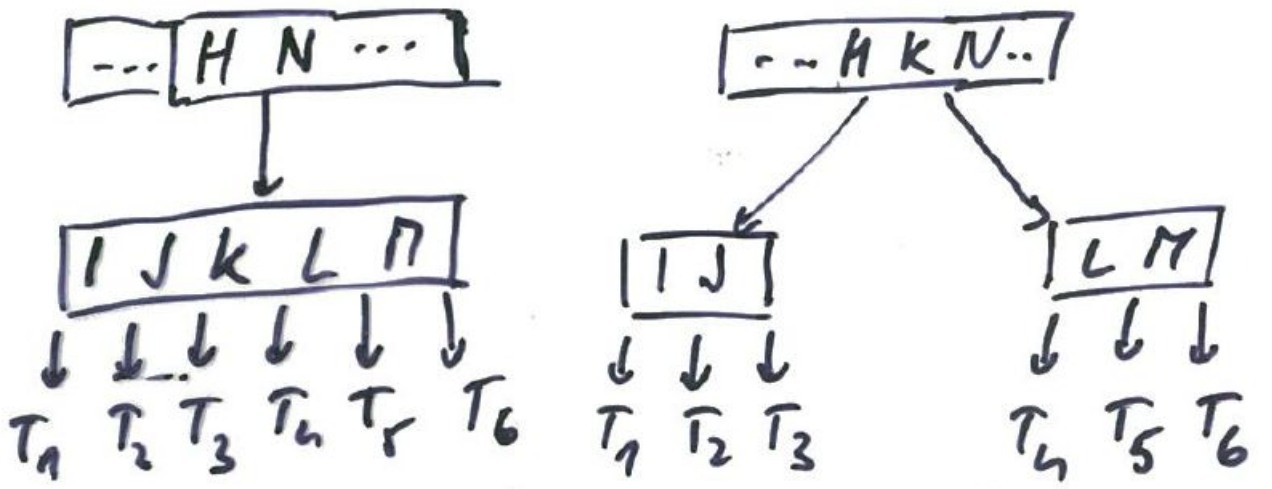
Operace na B-stromě:

- Create - vytvoření stromu
- Search - nalezení prvku
- Insert - vložení prvku do stromu
- Delete - vypuštění prvku ze stromu

- pomocná operace: rozdělení plného vrcholu: štěpení a slévání

Obrázek 16.

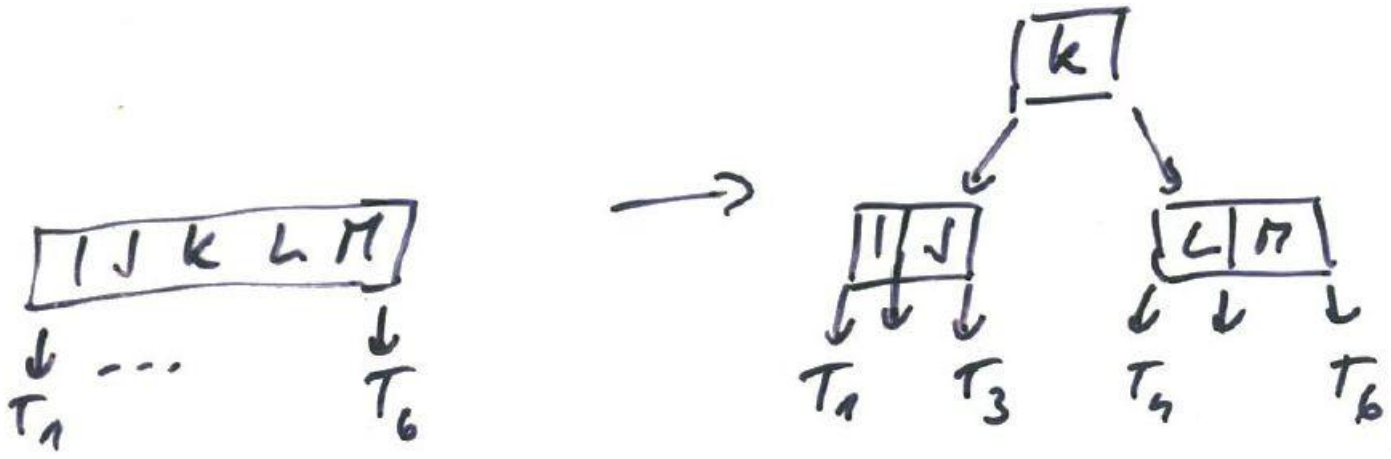
- vkládání do B-stromu výšky h: $O(h)$
- při průchodu dolů (preventivně) štěpíme plné vrcholy



Obrázek 16: B-stromy: štěpení, $t = 3$: před a po

- varianta: štěpíme při navracení, pak si musíme pamatovat (a zamknout) cestu
- speciálně: dělení kořene způsobí zvýšení výšky o 1

Obrázek 17.



Obrázek 17: B-stromy: štěpení v kořeni, $t = 3$: před a po

- invariant: vkládáme do neplného vrcholu

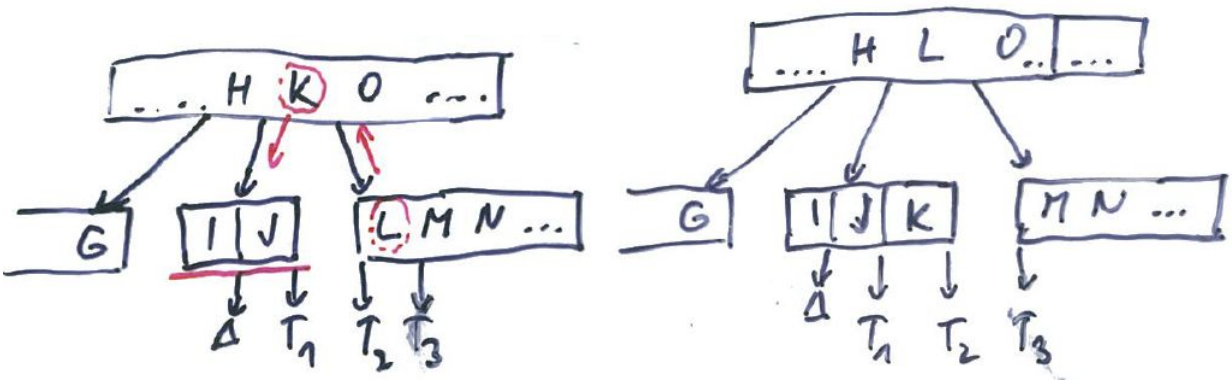
Vypouštění z B-stromu

- (rekurzivně) od kořene procházíme stromem, kontrolujeme a vynucujeme invariant: počet klíčů ve vrcholu je aspoň $t \rightarrow$ nejsme na minimu $t - 1$ klíčů

- zabezpečení invariantu: klíč z aktuálního vrcholu se přesune do syna a nahradí se klíčem ze souseda syna
- speciální případ: kořen (v neprázdném stromě): Pokud kořen nemá žádné klíče a pouze 1 syna, snížíme výšku stromu.

Rozbor případů vypouštění:

1. vypouštíme klíč k z listu x: přímo
2. vypouštíme klíč k z vnitřního vrcholu x
   (a) pokud syn y, který předchází k v x, má aspoň t klíčů: najdi předchůdce k' ke klíči k ve stromě y. Vypusť k' a nahraď k klíčem k' v x. (Nalezení a vypuštění k' v jednom průchodu)
   (b) symetricky, pro následníka z klíče k
   (c) jinak, synové y a z mají $t-1$ klíčů. Slej k a obsah z do y, z vrcholu x vypusť klíč k a ukazatel na z. Syn y má $2t-1$ klíčů a vypustíme k ze syna y.
3. klíč k není ve zkoumaném vrcholu. Určíme odpovídající kořen c_i podstromu s klíčem k. Pokud c_i má pouze $t-1$ klíčů, uprav c_i podle 3a) nebo 3b), aby obsahoval aspoň t klíčů. Pak vypouštěj rekurzivně v odpovídajícím synovi.
   (a) Pokud c_i má $t-1$ klíčů a má souseda s t klíči, přesuň klíč z x do c_i, přesuň klíč z (bezprostředního) souseda do x a přesuň odpovídajícího syna ze souseda do c_i
   (b) Pokud oba sousedé mají $t-1$ klíčů, slej c_i s jedním ze sousedů. Přitom se přesune 1 klíč z vrcholu x do nově vytvářeného vrcholu (jako medián)



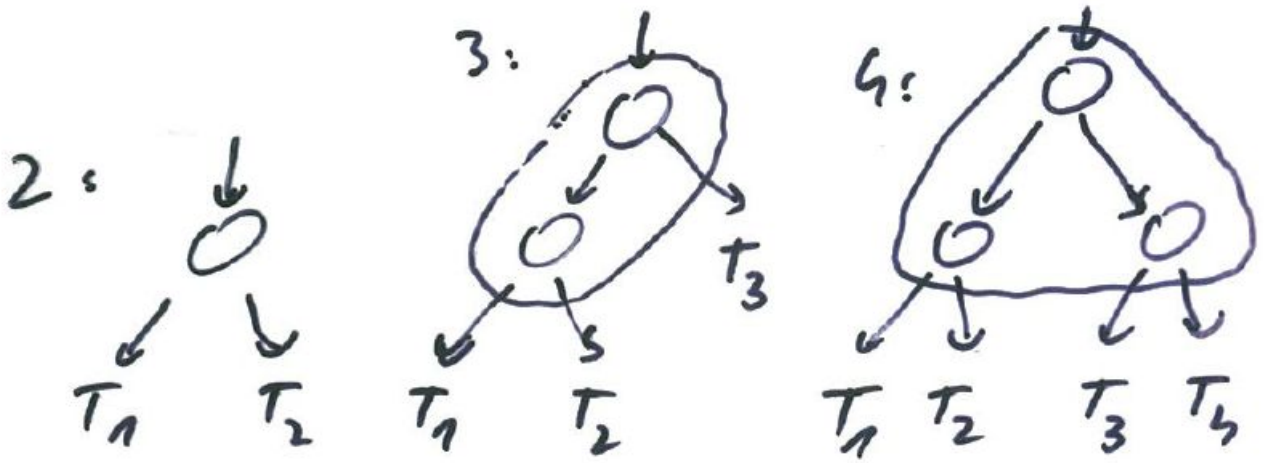
Obrázek 18: B-stromy: vypouštění – přesun: před a po

- složitost vypouštění ze stromu výšky h: máme $O(1)$ diskových přístupů na 1 hladině, proto je $O(h)$ diskových operací celkem

- přesun podrobně, 3a: (obrázek 18)
- uspořádání je zachováno
- hloubka se nemění, tj. neporuší
- nový vrchol IJK má t klíčů, tzn. splňuje invariant
- nový vrchol MN má aspoň $t - 1$ klíčů (po úpravě), tzn. splňuje podm. B-stromu

Souvislosti:

- varianta: všechna data v listech (tzv. externí reprezentace): výhody: víc B-stromů (indexů) nad stejnými daty, větší štěpení (šetřím pointry na data ve vnitřních vrcholech)
- souvislost s červeno-černými stromy: Pro $t = 2$ má vrchol B-stromu $1-3$ klíče a $2 - 4$ děti, nazývá se $2 - 4$ strom. Vrcholu B-stromu odpovídá černý vrchol s případnými (0 až 2) červenými syny. Při průchodu dolů se neprovádí předběžné štěpení, ale strom se upravuje až při průchodu zdola směrem vzhůru (pro Insert i Delete)
- varianta: počet p klíčů je: $t \leq p \leq 2t$, operace štěpení a slévání se



Obrázek 19: B-stromy a Č–Č stromy

vykonávají po úpravě podstromů zdola směrem vzhůru, pro $t = 1$ dostaneme $2 - 3$ stromy,

- zobecnění: $a - b$ stromy: máme explicitně určenou i dolní hranici a počtu synů, počet synů je mezi a a b kromě kořene, myšlenka úprav je stejná

- impl: provázané stromy - mají pointry na sousedy; slévání 3 vrcholů na 2; binární hledání pro pevnou délku klíčů; klíče proměnné délky (řetězce)

- statický strom (např. data na CD): vrcholy jsou naplněny, pamětí se neplýtvá

- prakticky: pro 32bitové pointry a 32bitové klíče potřebujeme 8B na položku; volíme t=255 resp. (256,511) pro B-stromy resp. (a,b)-stromy, aby se 1 vrchol vešel do diskové stránky velikosti 4KB; podobně volíme t=3 nebo (4,7), aby se 1 vrchol vešel do bloku keše velikosti 64B. V obou případech musíme zajistit zarovnání vrcholů na hranice stránek nebo bloků.

Hašování

- ideově vychází z přímo adresovatelných tabulek, které mají malé univerzum klíčů a prvky nemají stejné klíče.
- operace Insert, Search, Delete v čase $O(1)$
- implementace: pole; hodnoty pole obsahují reprezentované prvky (nebo odkazy na ně) anebo NIL
  - Ale přímo adresovatelné tabulky nevyhovují vždy:
- univerzum klíčů U je velké vzhledem k množině klíčů K, které jsou aktuálně uloženy ve struktuře
- prvky mají stejné klíče
  - idea: adresu budeme z klíče počítat
- hašovací funkce $h : U \to \{0, 1, \ldots, m-1\}$ mapuje univerzum klíčů U do položek hašovací tabulky $T[0..m-1]$
  $\to$ redukce paměti
- problém: vznikají kolize: dva klíče se hašují na stejnou hodnotu
- pozorování: kolizím se nevyhneme, pokud $|U| > m$

Řešení kolizí:
- zřetězením prvků
- otevřená adresace

Pozn: Hašovací funkce mají i jiné aplikace (s jinými požadavky) než pro datové struktury.
- kryptografické hašovací funkce: otisk MD5 (Message Digest 5) nebo SHA2: v kryptografii (např.) pro ověřování neporušenosti zpráv; pro adresaci/identifikaci objektů/souborů v distribuovaných systémech
- hašování pro výrobu signatury: předfiltrování stejných (částí)

klíčů (alg. Rabin-Karp: inkr. změny; Bloomův filtr; "texty"a bioinformatika); databáze: operace join (nerovnoměrné rozdělení dat); transpoziční tabulky v hrách (kolize řešíme přepisováním)

Analýza hašování se zřetězením

Df: Faktor naplnění $\alpha = \frac{n}{m}$ pro tabulku T velikosti m, ve které je uloženo n prvků.
V tabulce se zřetězením může platit $n > m$, teda $\alpha > 1$.

Předpoklady
- jednoduché uniformní hašování: Každý prvek se hašuje do m položek tabulky se stejnou pravděpodobností, nezávisle na jiných prvcích
- hodnota hašovací funkce $h(k)$ se počítá v čase $O(1)$

Analýza hledání prvku:
- úspěšné nalezení
- neúspěšné hledání

Věta: V hašovací tabulce s řešením kolizí pomocí zřetězení neúspěšné vyhledávání trvá průměrně $\Theta(1+\alpha)$, za předpokladu jednoduchého uniformního hašování
Dk: Podle předpokladu se klíč k hašuje se stejnou pravděpodobností do každé z m položek. Neúspěšné hledání klíče k je průměrný čas prohledání jednoho ze seznamů do konce. Průměrná délka seznamů je α. Proto je očekávaný počet navštívených prvků α a celkový čas (včetně výpočtu hašovací funkce $h(k)$) je $\Theta(1 + \alpha)$.

Úspěšné vyhledávání

Věta: V hašovací tabulce s řešením kolizí pomocí zřetězení úspěšné vyhledávání trvá průměrně $\Theta(1+\alpha)$ za předpokladu jednoduchého uniformního hašování.

Dk: Předpokládáme, že vyhledáváme každý z n uložených klíčů se stejnou pravděpodobností. Předpokládáme, že nové prvky se ukládají na konec seznamu.

Očekávaný počet navštívených vrcholů při úspěšném vyhledávání je o 1 větší než při vkládání tohoto prvku. Počítáme průměr přes n prvků v tabulce z 1+ očekávaná délka seznamu, do kterého se přidává i-tý prvek. Očekávaná délka tohoto seznamu je $(i-1)/m$. Dostaneme:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) \\
&= 1+\frac{1}{nm}(\frac{(n-1)n}{2}) \\
&= 1+\frac{\alpha}{2}-\frac{1}{2m} \\
&= \Theta(1+\alpha)
\end{aligned}$$

Závěr: Pokud $n = O(m)$, pak $\alpha = n/m = O(m)/m = O(1)$

Po vyhledání, vlastní operace pro přidání, resp. vypuštění prvku v čase $O(1)$.

- pozn: ukládání na konec vs. na začátek (frekventované klíče vs. princip lokality)

Volba Hašovacích funkcí

1) dělením
2) násobením
3) univerzální hašování (samostatně)

Dobrá hašovací funkce splňuje (přibližně) předpoklady jednoduchého uniformního hašování
Pro rozložení pravděpodobností P zvolení klíče k z univerza U chceme:

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{pro } j = 0, 1..m-1$$

ale: obvykle rozložení pravděpodobností neznáme

- interpretujeme klíče jako (přirozená) čísla, aby se daly používat aritmetické operace

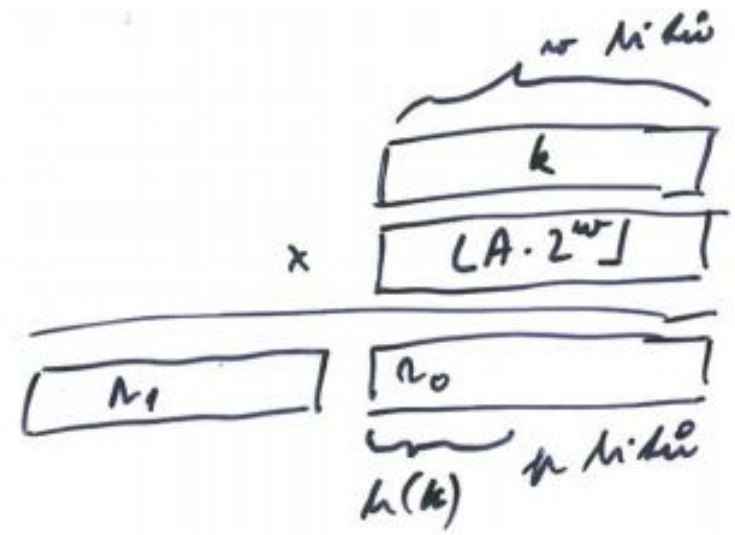
1) Dělení

$h(k) = k \bmod m$ zbytek po dělení
nevhodné pro $m = 2^p, 10^p, 2^p - 1$
vhodné: prvočísla "vzdálené"od mocnin dvojky

2) Násobení (obr. 20)

$h(k) = \lfloor m(kA \bmod 1) \rfloor$, pro $k \in [0, 1)$
obvykle pro $m = 2^p$
Knuth doporučuje $A \approx (\sqrt{5} - 1)/2 = 0,6180339887$ ... pro zlatý řez φ: $A = \varphi - 1 = 1/\varphi$
Pozorování: změna jednoho bitu se kvůli vynásobení A propaguje na mnoho míst vypočítaného klíče

"Paradox"narozenin (a vztah k hašování): Mezi 23 (a více) lidmi jsou s pravděpodobností větší než 50% aspoň dva se stejnými narozeninami. (První kolize vznikají brzo.)



Obrázek 20: Hašování násobením

Otevřené adresování

- všechny prvky jsou uloženy v tabulce, proto $\alpha < 1$
- pro řešení kolizí nepotřebujeme pointry, ale počítáme adresy navštívených pozic

$\rightarrow$ ve stejné paměti máme větší tabulku než při zřetězení
- posloupnost zkoušených pozic závisí na klíči a pořadí pokusu: $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- prohledávání pozic v posl. $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$

- operace Search, Insert, Delete (pouze někdy); místo Delete používáme Pseudodelete: zneplatnění prvku

Předpoklad uniformního hašování: každá permutace pozic $\{0..m-1\}$ je pro každý klíč vybrána stejně pravděpodobně jako posloupnost zkoušených pozic
- je těžké implementovat, používají se metody, které to nesplňují

Otevřené adresování - metody:

1. lineární zkoušení

2. kvadratické zkoušení

3. dvojité hašování

1) Lineární zkoušení, pro hašovací funkci $h' : U \to 0..m-1$ bude
$h(k,i) = (h'(k) + i) \bmod m$
- pouze m různých posloupností zkoušených pozic (a nezáleží na přírůstku)
- problém: primární klastrování: vznikají dlouhé úseky obsazených pozic
Pravděpodobnost obsazení pozice při vkládání závisí na obsazenosti předchozích pozic: pokud je obsazeno i pozic před, je pravděpodobnost obsazení $\frac{i+1}{m}$: speciálně pro $i = 0$, tj. předcházející prázdnou pozici, je pravděpodobnost $1/m$.

Příklad: $\alpha = 0.5$, porovnejte
a) obsazeny sudé pozice
b) obsazena první polovina tabulky

- DC: lze implementovat Delete: následující prvky posuneme, pokud patří na/před uvolňované místo

2) Kvadratické zkoušení:
$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, $c_1 \neq 0$, $c_2 \neq 0$
- aby se prohledala celá tabulka, hodnoty c_1, c_2 a m musí být vhodně zvoleny
- pro stejnou počáteční pozici klíčů x a y (tj. $h(x,0) = h(y,0)$) následuje stejná posloupnost zkoušených pozic

$\rightarrow$ problém: druhotné klastrování
- pouze m různých posloupností pozic

3) Dvojité hašování
$h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$, h_1 a h_2 jsou pomocné hašovací funkce
- $h_2(k)$ nesoudělné s m, aby se prošla celá tabulka
možné volby:
a) $m = 2^p$ a $h_2(k)$ je liché
b) m prvočíslo, $0 < h_2(k) < m$

Příklad:
$h_1(k) = k \bmod m$
$h_2(k) = 1 + (k \bmod m')$, kde m' je o trochu menší než m
- volíme například m a $m' = m - 2$ prvočísla
- máme $\Theta(m^2)$ posloupností zkoušených pozic
- místo Delete používáme Pseudodelete: vypouštěný prvek zneplatníme, odstraníme ho při přehašování tabulky

Analýza hašování s otevřeným adresováním

- "obvyklé"předpoklady: uniformní hašování: Pro každý klíč k je posloupnost zkoušených pozic libovolná permutace $\{0..m-1\}$ se stejnou pravděpodobností.

Věta: V tabulce s otevřeným adresováním s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet zkoušených pozic při neúspěšném vyhledávání nejvíce $1/(1-\alpha)$, za předpokladu uniformního hašování.

Dk: Všechny zkoušky pozice kromě poslední našly obsazenou pozici.
Definujme $p_i = Pr\{\text{právě } i \text{ zkoušek našlo obsazenou pozici}\}$

Očekávaný počet zkoušek je $1 + \sum_{i=0}^{\infty} i \cdot p_i$ (1)

Definujme $q_i = Pr\{\text{aspoň } i \text{ zkoušek našlo obsazenou pozici}\}$

Platí $1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=1}^{\infty} q_i$, chceme spočítat q_i (!$\sum$ od "1")

První zkouška narazí na obsazenou pozici s pravděpodobností $n/m = q_1$.

Obecně $q_i = (\frac{n}{m})(\frac{n-1}{m-1})...(\frac{n-i+1}{m-i+1}) \leq (\frac{n}{m})^i = \alpha^i$

Spočítáme (1): $1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}$,

QED

Důsledek: vkládání prvku vyžaduje nejvýš $1/(1-\alpha)$ zkoušek, za stejných předpokladů.

Věta: V tabulce s otevřeným adresováním s faktorem naplnění $\alpha = n/m < 1$ je očekávaný počet zkoušek při úspěšném vyhledávání nejvíc

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

za předpokladu uniformního hašování a pokud vyhledáváme každý klíč v tabulce se stejnou pravděpodobností.

Dk: Vyhledávání klíče k zkouší stejnou posloupnost pozic jako při vkládání klíče k. Podle minulého důsledku je vkládání $(i+1)$-ního klíče vykonáno na $1/(1-i/m)$ zkoušek. Platí $1/(1-i/m) = m/(m-i)$.

Průměr přes všechny klíče v tabulce je hledaný počet zkoušek: $\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n})$ kde $H_i = \sum_{j=1}^{i} \frac{1}{j}$ je

i-té harmonické číslo. Platí $\ln i \leq H_i \leq \ln i + 1$, odtud

$$\begin{aligned}\frac{1}{\alpha}(H_m - H_{m-n}) &\leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n)) \\ &\leq \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha} \\ &\leq \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}\end{aligned}$$

QED

Univerzální hašování

Idea: zvolíme hašovací funkce náhodně a nezávisle na klíčích (za běhu, z vhodné množiny funkcí)
- použití randomizace

Df. Nechť H je konečná množina hašovacích funkcí z univerza klíčů U do $\{0..m-1\}$. Množinu H nazveme *univerzální* (resp. *c-univerzální*), pokud pro každé dva různé klíče $x, y \in U$ je počet hašovacích funkcí $h \in H$, pro které $h(x) = h(y)$, roven $|H|/m$ (resp. $c.|H|/m$).
- tj. pro náhodně zvolenou funkci h je pravděpodobnost kolize pro x a y, kde $x \neq y$, právě $1/m$. To je stejná pravděpodobnost, jako když jsou hodnoty $h(x)$ a $h(y)$ zvoleny náhodně z množiny $\{0..m-1\}$
- implementace: hašovací funkce závisí na parametrech, které se zvolí za běhu, tj. při *vytvoření* tabulky.
$H = \{h_a(x) : U \to \{0..m-1\} | a \in A\}$, kde $h_a(x) = h(a, x)$, a je parametr

Důsledky randomizace:
- žádný konkrétní vstup (konkrétních n klíčů) není apriori

špatný
- opakované použití na stejný vstup volá (skoro jistě) různé hašovací funkce

$\rightarrow$ "průměrný případ" nastane pro libovolné rozložení vstupních dat (!); průměr zde uvažujeme přes možné haš. fce

Věta: Nechť h je náhodně vybraná hašovací funkce z univerzální množiny hašovacích funkcí a nechť je použita k hašování n klíčů do tabulky velikosti m, kde $n \leq m$. Potom očekávaný počet kolizí, kterých se účastní náhodně vybraný konkrétní klíč x, je menší než 1.

Dk: Pro každý pár různých klíčů y a z označme c_{yz} náhodnou proměnnou, která nabývá hodnotu 1, pokud $h(y) = h(z)$ a 0 jinak.
Z definice, konkrétní pár klíčů koliduje s pravděpodobností $1/m$, proto očekávaná hodnota $E[c_{yz}] = 1/m$.
Označme C_x celkový počet kolizí klíče x v hašovací tabulce T velikosti m obsahující n klíčů. Pro očekávaný počet kolizí máme:

$$E[C_x] = \sum_{y \in T, y \neq x} E[c_{xy}] = \frac{n-1}{m}$$

Protože $n \leq m$, platí $E[C_x] < 1$.

Pozn: předpoklad $n \leq m$ implikuje, že průměrná délka seznamu klíčů nahašovaných do stejné adresy je menší než 1.

Konstrukce univerzální množiny hašovacích funkcí (jedna z možností): založena na skalárním součinu.

Zvolíme prvočíslo m jako velikost tabulky. Každý klíč x rozdělíme na $r+1$ částí (např. znaků, hodnota r závisí na velikosti klíčů). Píšeme $x = \langle x_0, x_1 ... x_r \rangle$. Zvolené r splňuje podmínku, že každá část x_i je (ostře) menší než m. Zvolme $a = \langle a_0, a_1 ... a_r \rangle$ posloupnost $(r+1)$ čísel náhodně a nezávisle z množiny $\{0, 1..m-1\}$. Definujeme $h_a \in H$: $h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$ a $H = \bigcup_a \{h_a\}$
Platí $|H| = m^{r+1}$, tj. počtu různých vektorů a.

Věta: H je univerzální množina hašovacích funkcí.
Dk: Uvažujme různé klíče x a y. Bez újmy na obecnosti, nech $x_0 \neq y_0$. Pro pevné hodnoty $a_1, a_2 ... a_r$ je právě jedna hodnota a_0, která splňuje $h(x) = h(y)$. Je to řešení rovnice

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \bmod m$$

Protože m je prvočíslo, nenulová hodnota $x_0 - y_0$ má jediný inverzní prvek modulo m a teda existuje jediné řešení pro a_0 modulo m. Následně, každý pár klíčů x a y koliduje pro právě m^r hodnot vektoru a, protože kolidují právě jednou pro každou volbu $\langle a_1, a_2 ... a_r \rangle$, tj. pro jedno a_0. Protože je m^{r+1} možností pro a, klíče kolidují s pravděpodobností $m^r/m^{r+1} = 1/m$. Teda H je univerzální. QED

- pozn: a_i vybíráme včetně 0

Příklad randomizace: Zobristovo hašování pro reprezentaci pozic her (v transpozičních tabulkách), používá náhodná čísla pro (jednobitové) rysy/fíčury, používá xor (místo plus a mod), typicky v 32 bitech.

- Pro zjištění úspěchu nalezení se neporovnávají celé klíče, tj. pozice, ale hodnota sekundární hašovací funkce.

Hašování a rostoucí tabulky (dynamizace h.t.)

Chceme odstranit nevýhody hašování:
- pevná velikost tabulky
- nedokonalá implementace Delete (u některých metod)
při zachování asymptotické ceny operací (tj. průměrně $O(1)$ pro pevné α)

Idea: Periodická reorganizace datové struktury (a využití amortizované složitosti)
- při růstu: při dosažení naplnění α zvětšíme tabulku d-krát (např. $d = 2$, z $m = 2^i$ na $m = 2^{i+1}$; pro prvočísla m si je předpočítáme) a prvky přehašujeme (s novou haš. fcí.)

Dále uvažujme $d = 2$: aspoň $2^{i-1}.\alpha$ operací Insert (od posledního přehašování) zaplatí:

- 1x: přidání $2^{i-1}.\alpha$ prvků do staré tabulky
- 2x: přehašování $2.2^{i-1}.\alpha$ prvků

DC: Kredit operace obecně $\leq \frac{2d-1}{d-1}$

- pozn: vhodné α lze spočítat (z pohledu efektivity budoucích operací s a bez přehašování)

- operace přehašování je amortizována minulými operacemi, které vytvoří dostatečný kredit $\rightarrow$ amortizovaná slož. $O(1)$

Obecné řešení při růstu i zmenšování tabulky: po přehašování je naplnění tabulky $\alpha/4$ až $\alpha/2$, označme $n^* = \alpha.m$.
- tabulku zvětšíme, pokud máme n^* prvků (proběhlo aspoň

$n^*/2$ operací Insert)
- tabulku zmenšíme, pokud máme $n^*/8$ prvků (proběhlo aspoň $n^*/8$ operací Delete)
- velikost tabulky nezměníme, máme aspoň $n^*/2$ pseudo-vypuštěných prvků, vytvoření pseudovolného místa potřebuje (aspoň) 2 operace

Kvůli mezeře mezi $\alpha/4$ a $\alpha/2$ se nemůže stát, že tabulku zvětšíme a hned ji potřebujeme zmenšovat (nebo naopak).

Aktuální počet prvků si pamatujeme; pseudovolná místa se při přehašování uvolní.

Pozn: (DC) Přehašování dávkové (popsané výš) vs. postupné: máme dvě aktivní tabulky a s každou operací několik prvků přemístíme do nové tabulky. Po přemístění všech prvků starou tabulku zahodíme. (Jen to chceme stihnout dřív, než začne další přehašování.)

Jsou i jiné metody dynamizace dat. struktur.

Metoda Rozděl a panuj (Divide et impera)

- metoda pro návrh (rekurzivních) algoritmů

- Malé (nedělitelné) zadání vyřešíme přímo, jinak
- Úlohu rozdělíme na několik podúloh stejného typu, ale menšího rozsahu (*)
- Vyřešíme podúlohy, rekuzivně
- Sloučíme získaná řešení na řešení původní úlohy

příklady:
- mergesort, binární vyhledávání v utříděném poli
- (*) někdy je potřeba původní úlohu zobecnit: hledání mediánu na hledání k-tého z n prvků (cv.)

Analýza složitosti

$T(n)$ Čas zpracování úlohy velikosti n, pro $n < c$ předpokládáme $T(n) = \Theta(1)$

$D(n)$ čas na rozdělení úlohy velikosti n na a podúloh (stejné) velikosti n/c plus čas na sloučení řešení podúloh

$\rightarrow$ rekurentní rovnice

$T(n) = a.T(n/c) + D(n) \quad pro\ n \geq c$

$T(n) = \Theta(1) \quad pro\ n < c$

příklad: mergesort

```
procedure ms(a[1..n]) % mergesort
(a1[1..n/2],a2[1..n/2]):=rozdel(a[1..n]);
return(merge(ms(a1[1..n/2]),
             ms(a2[1..n/2]) )).
```

- rekurze: dvě ($a = 2$) podúlohy poloviční ($c = 2$) velikosti
- dělení: sudá-lichá $\rightarrow O(n)$, první/druhá polovina (v poli) $\rightarrow O(1)$
- sloučení (merge): $O(n)$, celkem $D(n) = O(n)$, tj. $d = 1$

Vychází rovnice: $T(n) = 2.T(n/2) + O(n)$

Pozn.: "Malé" podproblémy řešíme algoritmem s malou réžií (např. v Quicksortu zatřiďováním). Tato změna neovlivní asymptotickou časovou složitost.

Metody řešení

1. substituční metoda
2. Master Theorem (pomocí "kuchařky")

Master Theorem taky ukazuje, která část řešení je "kritická"

Používáme zjednodušení

- předpoklad $T(n) = \Theta(1)$ pro malá n nepíšeme do rovnice
- zanedbáváme celočíselnost, píšeme pouze $n/2$ místo $\lfloor n/2 \rfloor$ a $\lceil n/2 \rceil$
- řešení nás zajímají pouze asymptoticky $\rightarrow$ používáme asymptotickou notaci už v zápisu rekurentní rovnice

Substituční metoda

- uhádneme asymptoticky správné řešení

- dokážeme, typicky indukcí, správnost odhadu (zvlášť pro dolní a horní odhad)

- - !častá chyba: odhady v indukci musí vyjít se stejnou asymptotickou konstantou jako v ind. předpokladu :-) , nestačí asymptotický odhad s jinou konstantou. (Tak lze chybně indukcí "dokázat" $n^2 \in O(n)$ s využitím $(n+1)^2 = n^2+2n+1$.)

- příklad: mergesort

Pro $T(n) = 2.T(n/2)+b.n$ uhádneme $T(n) = \Theta(n \log n)$. Pro zjednodušení předpokládejme, že funkce $T(n)$ je neklesající.

Dokážeme indukcí horní odhad $T(n) = O(n \log n)$ (pro $n > n_0$). Chceme teda pro vhodnou konstantu c ukázat, že $T(n) \leq cn \log n$, volme $c \geq b$ a tak, aby platili okrajové podmínky pro indukci (tj. $T(n) \leq cn \log n$ pro $n_0/2 \leq n \leq n_0$). Předpokládejme, že tvrzení platí pro $k = n/2$. Potom platí

$T(n) = 2T(n/2) + bn$ (rekurzivní vztah)
$\leq 2c(n/2)\log(n/2) + bn$ (substituce)
$= cn(\log n - \log 2) + bn$ (krácení 2, log)
$= cn(\log n - 1) + bn$ (log)
$= cn \log n - cn + bn$ (distributivita)
$\leq cn \log n$ (volba c), QED

Dolní odhad analogicky.

Ve výše uvedeném příkladě byla náročnost "rekurze" a "režie"

vyrovnaná. Pro režii $O(n^\alpha)$ je vhodný odhad $O(n^\alpha \log n)$.

- příklad: rychlé násobení dlouhých čísel

Pro $T(n) = 3.T(n/2) + b.n$ uhádneme $T(n) = O(n^{\log_2 3})$. Pro zjednodušení předpokládejme, že funkce $T(n)$ je neklesající.

Dokážeme indukcí horní odhad $T(n) = O(n^{\log_2 3})$ (pro $n > n_0$). Chceme teda pro vhodné konstanty c, d ukázat, že $T(n) \leq cn^{\log_2 3} - dn$ (trik volby), volme $c \geq d$ a tak, aby platili okrajové podmínky pro indukci (tj. $T(n) \leq cn^{\log_2 3} - dn$ pro $n_0/2 \leq n \leq n_0$). Předpokládejme, že tvrzení platí pro $k = n/2$. Hledáme vhodné d a c v závislosti na b (místo uhádnutí v minulém příkladě). Platí

$T(n) = 3T(n/2) + bn$ (rekurzivní vztah)
$\leq 3c(n/2)^{\log_2 3} - 3d(n/2) + bn$ (substituce, ind. předp.)
$= 3cn^{\log_2 3}.(1/2)^{\log_2 3} - (3/2)dn + bn$ (aritmetika)
$= cn^{\log_2 3} + n(-(3/2)d + b)$ (log, vykrácení, upr.)
$? \leq ? cn^{\log_2 3} - dn$ zbývá dokázat

Stačí ukázat $-(3/2)d + b \leq -d$, protože n je kladné.

$-(3/2)d + b \leq -d$ hypotéza
$b \leq -d + (3/2)d = 1/2d$ převedení d, úprava
$2b \leq d$ osamostatnění d
(úpravy byly ekvivalentní, vyjádření d směrem dolu, důkaz (pro zvolené d) směrem nahoru)

Pro tuto volbu $d \geq 2b$ (a následně volbu c) platí poslední nerovnost výše, QED.

Triková volba $cn^\alpha - dn$ umožní dokázat tesnější odhad ($T(n) \in O(n^{\log_2 3})$ místo $T(n) \in O(n^{\log_2 3 + \epsilon})$), protože první

člen vyjde z rekurze přesně a druhý člen je rezerva, do kterého se "schová" režie.

V tomto případě rekurze (tj. # koncových případů) je dominantní, proto vhodná volba odhadu je $T(n) \in O(n^\alpha)$.

Třetí případ je, pokud je dominantní režie. Potom je odhad celkové složitosti roven složitosti režie.

př: hledání mediánu: $T(n) = T(n/5)+T(7n/10)+O(n) \rightarrow T(n) = O(n)$ substituční metodou

Pro $T(n) = T(n/5)+T(7n/10)+O(n)$ uhádneme $T(n) = \Theta(n)$. Pro zjednodušení předpokládejme, že funkce $T(n)$ je neklesající.

Dokážeme indukcí horní odhad $T(n) = O(n)$ (pro $n > n_0$). Chceme teda pro vhodnou konstantu c ukázat, že $T(n) \leq cn$. Zvolíme $c \geq c'$, kde c' je vhodná konstanta, pro kterou platí okrajové podmínky pro indukci (tj. $T(n) \leq cn$ pro $n \leq n_0$). Předpokládejme, že tvrzení platí pro $k < n$. Hledáme vhodné c v závislosti na b. Platí

$T(n) = T(n/5) + T(7n/10) + bn$ (rekurzivní vztah)
$\leq c(n/5) + c(7n/10) + bn$ (substituce, ind. předp.)
$= (9/10)cn + bn$ (aritmetika)
$? \leq ? cn$ zbývá dokázat

Stačí ukázat $9/10c + b \leq c$, protože n je kladné.

$9/10c + b \leq c$ hypotéza
$b \leq c - 9/10c = 1/10c$ převedení c, úprava
$10b \leq c$ osamostatnění c
(úpravy byly ekvivalentní, vyjádření c směrem dolu, důkaz (pro zvolené c) směrem nahoru)

Pro volbu $c \geq \max(10b, c')$ platí poslední nerovnost v hlavním důkazu výše, QED.

(Hledání k-tého prvku a mediánu)

Master theorem

Nech $a \geq 1, c > 1, d \geq 0$ jsou reálna čísla a nech $T : N \to N$ je neklesající funkce taková, že pro všechna n ve tvaru c^k, pro $k \in N$, platí

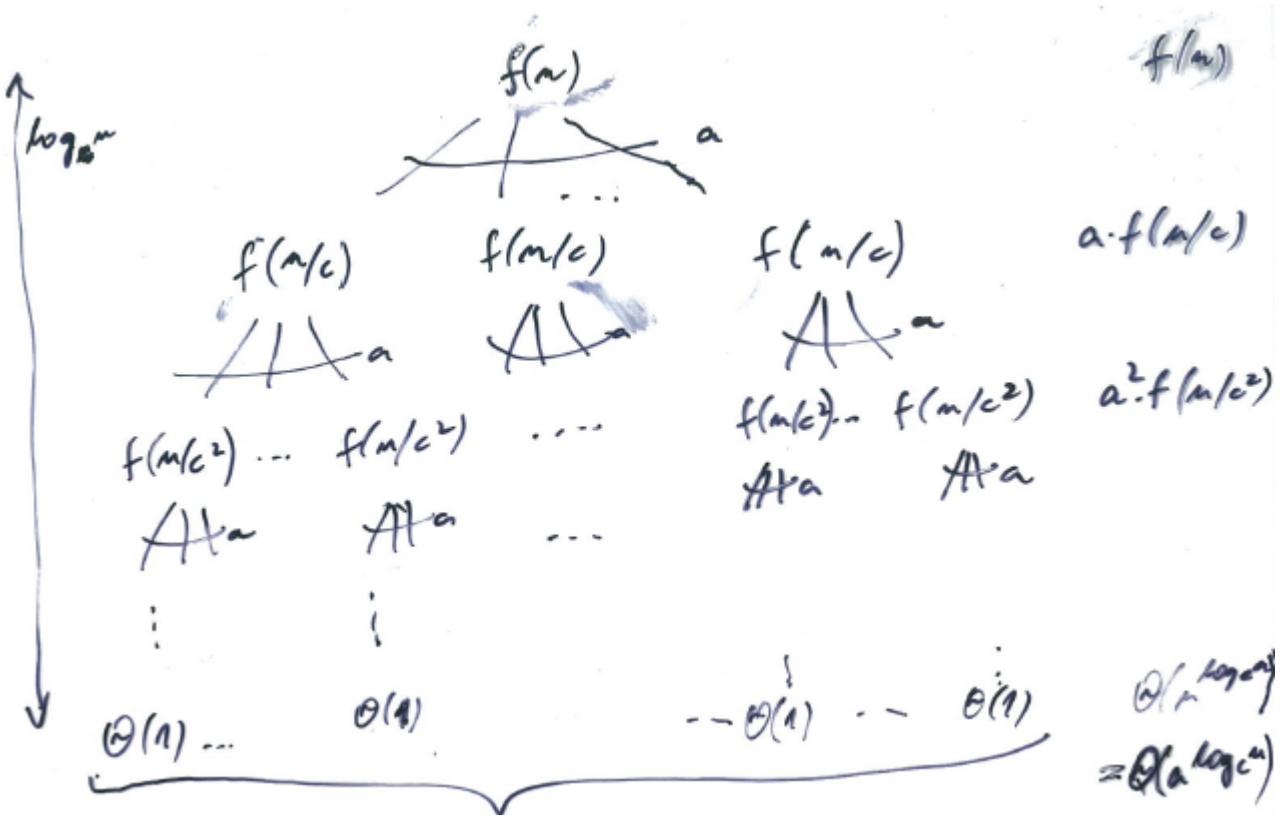
$T(n) = a.T(n/c) + F(n)$

kde pro funkci $F : N \to N$ platí $F(n) = O(n^d)$. Označme $x = \log_c a$. Potom

a) je-li $a < c^d$, tj. $x < d$, potom $T(n) = O(n^d)$

b) je-li $a = c^d$, tj. $x = d$, potom $T(n) = O(n^d \log_c n) = O(n^x \log_c n)$

c) je-li $a > c^d$, tj. $x > d$, potom $T(n) = O(n^x)$



Obrázek 21: Rekurzivní rozdělování

Dk. Protože $F(n) = O(n^d)$, existují n_0 a e taková, že pro každé $n \geq n_0$ platí $F(n) \leq e \cdot n^d$. Zvolme m, tž. $c^m \geq n_0$,

pak pro každé $k \geq m$ platí $F(c^k) \leq e \cdot (c^k)^d$ (tj. platí už bez výjimek).

Zvolme $b = \max\{T(c^m), e \cdot (c^m)^d\}$,

potom pro $k \geq 0$ a $n = c^{m+k} = c^m \cdot c^k$ platí

$$T(n) \leq a \cdot T(n/c) + b \cdot (c^k)^d$$

Indukcí dle k ukážeme, že pro $n = c^{m+k}$ platí (první sčítanec odpovídá počtu koncových případů, druhý režii)

$$T(n) \leq b \cdot \left(a^k + c^{kd} \sum_{i=0}^{k-1} (a/c^d)^i \right)$$

Pro $k = 0$ tvrzení platí.

Předpokládejme, že tvrzení platí pro k, dokažme pro $n = c^{m+k+1}$.

$T(n) \leq a \cdot T(c^{m+k}) + b \cdot (c^{k+1})^d$ z rozpisu

$\leq ab \left(a^k + c^{kd} \sum_{i=0}^{k-1} (a/c^d)^i \right) + b \cdot (c^{k+1})^d$ subst. z i.p.

$= b \left(a^{k+1} + (c^{k+1})^d ((a/c^d) \sum_{i=0}^{k-1} (a/c^d)^i + 1) \right)$

$= b \left(a^{k+1} + (c^{k+1})^d \sum_{i=0}^{k} (a/c^d)^i \right)$, tím je ind. krok dokončen.

Označme s_k součet prvních k členů geometrické posloupnosti $\{(a/c^d)^i, i = 0, 1, \ldots\}$, t.j.

$$s_k = \frac{(a/c^d)^k - 1}{a/c^d - 1}$$

Rozbor případů:

a) $a < c^d$: geometrická řada s kvocientem a/c^d konverguje a pro libovolné k platí $s_k < s = \frac{c^d}{c^d - a} = konst.$

Odtud (po úpravách) $T(n) \leq T(c^{m+k}) \leq b \cdot (a^k + c^{kd} s_k) < b \cdot (c^{dk} + c^{kd} s_k) \leq konst \cdot c^{kd} = O(n^d)$

Neformálně: Dominující člen je "režie" na jednotlivých úrovních rekurze.

b) $a = c^d$: platí $s_k = k$, protože členy řady jsou rovny 1.

Odtud $T(n) \leq konst \cdot c^{kd} \cdot k = O(n^d \log n)$, protože $k = \Theta(\log n)$

c) $a > c^d$: platí $s_k < \frac{(\frac{a}{c^d})^k}{\frac{a}{c^d} - 1} = (\frac{a}{c^d})^k \cdot t$ (Idea: od největšího členu směrem "dolů" je to geometrická řada s kvocientem menším než 1.)

Odtud $T(n) \leq T(c^{m+k}) \leq b \cdot (a^k + c^{kd} (\frac{a}{c^d})^k \cdot t) = b \cdot (a^k + a^k \cdot t) = konst \cdot a^k = konst \cdot c^{(k \cdot \log_c a)} = O(n^{\log_c a})$

Neformálně: Dominující člen je počet (a složitost) koncových případů rekurze.

Příklady a konstanty z Master teorému

mergesort: $T(n) = 2.T(n/2) + O(n) \rightarrow T(n) = O(n \log n)$; $a = 2, c = 2, d = 1$

binární vyhl. v utříděném poli: $T(n) = 1.T(n/2) + O(1) \rightarrow T(n) = O(\log n)$; $a = 1, c = 2, d = 0$

násobení čísel - klasické: $T(n) = 4.T(n/2) + O(n) \rightarrow T(n) = O(n^2)$; $a = 4, c = 2, d = 1$

násobení čísel - rychlé: $T(n) = 3.T(n/2)+O(n) \to T(n) = O(n^{\log 3})$; $a = 3, c = 2, d = 1$

násobení matic - klasické: $T(n) = 8.T(n/2) + O(n^2) \to T(n) = O(n^3)$; $a = 8, c = 2, d = 2$

hledání mediánu (k-tého prvku): $T(n) = T(n/5)+T(7n/10)+O(n) \to T(n) = O(n)$ substituční metodou

(Příklady odjinud:)

kreslení fraktální křivky: $T(n) = 4.T(n/3) + O(1) \to T(n) = O(n^{\log_3 4})$ (Místo prostřední třetiny úsečky přidáme ostatní dvě strany rovnostranného trojúhelníku, opakovaně.) ; $a = 4, c = 3, d = 0, x = \log_3 4$

Cantorovo diskontinuum (pokud jste měli na Analýze; taky fraktál): $T(n) = 2.T(n/3) + O(1) \to T(n) = O(n^{\log_3 2})$; $a = 2, c = 3, d = 0, x = \log_3 2$ (Z intervalu $\langle 0, 1\rangle$ odebíráme opakovaně prostřední třetinu. Zbude C.d., tj. čísla, která v trojkovém zápisu nemají jedničku. Hodnota $x = \log_3 2$ je (neceločíselná) dimenze fraktálu.)

Násobení čtvercových matic

Úloha: pro dané dvě matice A, B řádu $n \times n$ spočítat $C = A \otimes B$, řádu $n \times n$.

Klasický algoritmus má složitost $O(n^3)$, počítáme n^2 skalárních součinů délky n.

Předpokládejme, že n je mocnina čísla 2, tj. $\exists k, n = 2^k$. Potom vstupní matice můžeme dělit na 4 matice poloviční řádu (až do matic 1 x 1).

Použijeme "rozděl a panuj"(na 4 podmatice)

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$C_{11} = (A_{11} \otimes B_{11}) \oplus (A_{12} \otimes B_{21})$
$C_{12} = (A_{11} \otimes B_{12}) \oplus (A_{12} \otimes B_{22})$
$C_{21} = (A_{21} \otimes B_{11}) \oplus (A_{22} \otimes B_{21})$
$C_{22} = (A_{21} \otimes B_{12}) \oplus (A_{22} \otimes B_{22})$

- počet maticových operací na maticích řádu $n/2$: 8 násobení $\otimes$ a 4 sčítání $\oplus$ (a pomocné operace)

- počet sčítání reálných čísel v maticovém sčítání: $4(n/2)^2 = n^2$

Vyšla rovnice: $T(n) = 8T(n/2) + O(n^2)$

Master theorem: $a = 8$, $c = 2$, $\log_c a = 3$, $d = 2$, platí $T(n) = O(n^3)$, tj. asymptoticky stejné jako klasický algoritmus

- ke snížení složitosti je potřeba snížit $a = 8$ a zachovat (nebo mírně zvýšit) $d = 2$.

Strassenův alg. násobení matic (1969)
(vzorce nezkouším)

Používá pouze 7 násobení matic řádu $n/2$
$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$
$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$
$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$
$M_4 = (A_{11} \oplus A_{12}) \otimes B_{22}$
$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$
$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$
$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$
- spočítáme výsledné submatice
$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$
$C_{12} = M_4 \oplus M_5$
$C_{21} = M_6 \oplus M_7$
$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$
- počet operací nad maticemi řádu $n/2$: 7 násobení $\otimes$ a celkem 18 sčítání $\oplus$ a odčítání $\ominus$
- složitost: $T(n) = 7T(n/2) + O(n^2)$
- Master theorem: $a = 7$, $c = 2$, $\log_c a = \log_2 7 = x$, $d = 2$, teda $T(n) = O(n^x) \doteq O(n^{2.81})$
- praktické použití: husté matice řádu $n > 45$ větší asymptotická konstanta než u klasického násobení
- pozn.: Strassenův algoritmus používá odčítání, tj. inverzní prvky vzhledem ke sčítání $\rightarrow$ pracuje nad (maticí nad) okruhem (s op. plus a krat). Proto nejde použít pro počítání minimálních cest (s min a plus) ani pro boolovské matice (s operacemi or a and). Ale lze ho použít na (vstupní)

$0-1$ matice (místo boolovských) pro výpočet tranzitivního uzávěru grafu. Číslo 0 reprezentuje false, kladné číslo true a počítáme prvky výsledné matice $c_{ij} = \sum_k (a_{ik} b_{kj})$ místo $c_{ij} = \bigvee_k (a_{ik} \wedge b_{kj})$.

Strassenův alg. v obrázcích (A_{ij} po řádcích, B_{ij} po sloupcích):

$$C_{11} = \begin{array}{c} \\ A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{array} \begin{array}{c} \begin{array}{cccc} B_{11} & B_{21} & B_{12} & B_{22} \end{array} \\ \begin{pmatrix} + & . & . & . \\ . & + & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \end{array}$$

$$C_{12} = \begin{pmatrix} . & . & + & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \quad C_{21} = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & + & . & . \end{pmatrix} \quad C_{22} = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & + & . \\ . & . & . & + \end{pmatrix}$$

Matice M:

$$M_1 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix} \quad M_2 = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix} \quad M_3 = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$M_4 = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad M_5 = \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad M_6 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}$$

$$M_7 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

Součty v závorkách:

$C_{11} = M_1 \oplus (M_2 \ominus M_4 \oplus M_6)$; $C_{22} = (M_2 \oplus M_5 \ominus M_7) \ominus M_3$:

$$M_2 \ominus M_4 \oplus M_6 = \begin{pmatrix} + & \cdot & \cdot & \pm \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \pm & + & \cdot & + \end{pmatrix} \quad (M_2 \oplus M_5 \ominus M_7) =$$

$$\begin{pmatrix} + & \cdot & + & \pm \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \pm & \cdot & \cdot & + \end{pmatrix}$$

Násobení dlouhých čísel

Úloha: pro dané dvě přirozené čísla x a y o n bitech spočítat součin $s = x * y$. (Analogicky lze počítat v soustavě s jiným základem než 2)

Klasický algoritmus má složitost $O(n^2)$, násobíme každý bit s každým, resp. sčítáme n čísel dlouhých $O(n)$ bitů.

Předpokládejme, že n je mocnina čísla 2. (Případně doplníme nulami zleva.) Označme $m = n \div 2$.

Násobení s rozdělením na poloviny: x a y jsou dvouciferná čísla v soustavě se základem 2^m.

$x = x_1 * 2^m + x_2$

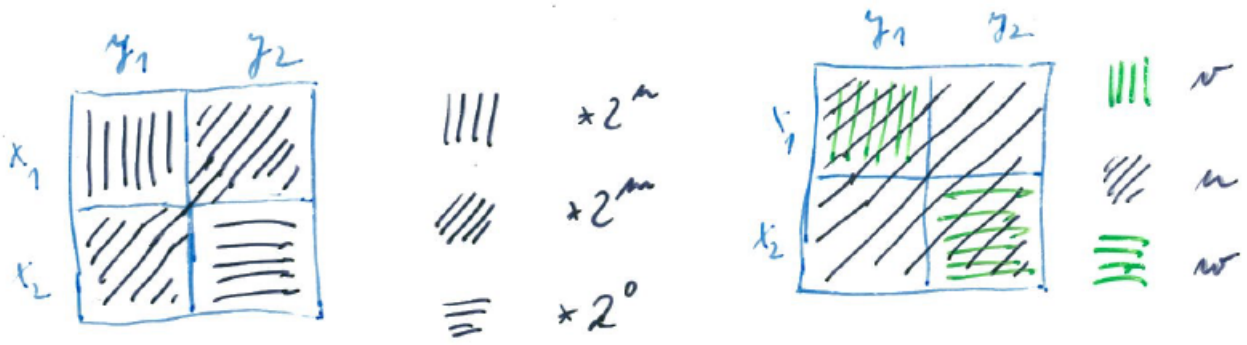
$y = y_1 * 2^m + y_2$

$x * y = x_1 * y_1 * 2^{2m} + (x_1 * y_2 + x_2 * y_1) * 2^m + x_2 * y_2$

- 4 násobení (násobení čísly 2^k jsou shifty), vychází vztah: $T(n) = 4 \cdot T(n/2) + 5n$

odtud: $T(n) = O(n^{\log_2 4}) = O(n^2)$, pro $a = 4, c = 2, d = 1$ (tj. $a > c^d$)

Idea zlepšení (asymptotické složitosti): zmenšit a, tj. ušetřit násobení.



Obrázek 22: Násobení čísel: vlevo klasicky, vpravo lépe

Spočítáme:
$u = (x_1 + x_2) \underline{*} (y_1 + y_2)$
$v = x_1 \underline{*} y_1$
$w = x_2 \underline{*} y_2$
$x * y = v * 2^{2m} + (u - v - w) * 2^m + w$
- 3 násobení: $T(n) = 3 \cdot T(n/2) + O(n)$
odtud: $T(n) = O(n^{\log_2 3}) \doteq O(n^{1.59})$, pro $a = 3, c = 2, d = 1$ (tj. $a > c^d$)

- obecně: Při počítání u mohou být $(x_1 + x_2)$ a $(y_1 + y_2)$ o 1 bit delší než m. To lze ošetřit rozborem případů za cenu dalších (režijních, tj. lineárních) operací: $u = (x_b 2^m + x') * (y_b 2^m + y') = x_b y_b 2^{2m} + x_b y' 2^m + y_b x' 2^m + x' * y'$, kde $x_b, y_b \in \{0, 1\}$ a $x' < 2^m, y' < 2^m$. Rekurzivní volání je pouze poslední sčítanec.

Jiná možnost je využít varianty algoritmu, která počítá $u' = (x_1 - x_2) * (y_1 - y_2)$ a samostatně ošetřuje znaménko u'.

DC: Popište, jak lze vynásobit dvě komplexní čísla pomocí 3 reálných násobení.

Dynamické programování

DP je *metoda* pro řešení úloh. Podobně jako metoda rozděl a panuj anebo hladový algoritmus.

DP se používá (podobně jako rozděl a panuj) při úlohách, které můžeme rozdělit na podproblémy, ale ty jsou závislé: sdílejí podproblémy.

DP se používá typicky na optimalizační problémy. Řešení jsou ohodnocena a ze všech řešení chceme vybrat řešení s optimální (minimální nebo maximální) cenou.

Vývoj algoritmu založeného na DP se typicky skládá z:

- Rekurzivní definice ceny optimálního řešení. Ta dává přímočarý rekurzivní algoritmus, který je ale pomalý.
- Nalezení opakovaných výpočtů a charakterizace struktury *stavového prostoru*.
- Zavedení tabulky pro stavový prostor: *tabelace*, taky kešování nebo memoizace. Počítáme pořád rekurzivně, ale výsledky si pamatujeme.
- Převod rekurze na iteraci, dostaneme typicky rychlejší a jednodušší algoritmus. Spočítání ceny optima zdola-nahoru, ve vhodném pořadí. (A)
- Zrekonstruování optimálního řešení ze spočítaných informací, pokud potřebujeme i řešení (nestačí nám cena).

(A) Vhodné pořadí respektuje závislosti ve stavovém prostoru. Potřebujeme topologické uspořádání stavů.

Pozn. Jednotlivé úlohy můžou mít lepší jednoúčelové algoritmy, než jaké získáme z DP (Fib. číslo $f(n)$ v $O(\log n)$, optimální vyhledávací strom v $O(n^2)$ místo $O(n^3)$, ...). V principu to znamená, že nevyplňujeme celou tabulku stavového prostoru.

Podobně, pokud to dovoluje struktura stavového prostoru a nepoužíváme tabulku k rekonstrukci, můžeme šetřit paměť a část informací zahazovat nebo přepisovat. (alg. Floyd-Warshall pro všechny cesty v grafu.)

Př. Fibonacciho čísla.

$f(n) = f(n-1) + f(n-2)$ pro $n > 2$ a $f(n) = 1$ pro $n \in \{1, 2\}$.

- Stavy jsou hodnoty argumentu z $\{1, ..., n\}$.
- Tabulka je pole, indexované stavy.
- Převedeme rekurzi na cyklus, počítáme zdola.
- Zde (neplatí obecně): jediná hodnota je optimální
- Zde: nepotřebujeme si pamatovat celý stavový prostor, stačí poslední dva stavy $\rightarrow$ šetříme pamětí
- Zde: DP vede na alg. v $O(n)$. Ex. alg. v $O(\log n)$: $(f_{n+1}, f_n) = (1, 1|0, 1)^{n-1} \cdot (f_2, f_1)$ s rychlým umocňováním matic

Př. Floydův-Warshallův alg. pro všechny min. cesty v grafu.

Opakování:

Invariant: $\delta_k(i, j)$ je délka nejkratší cesty z i do j, jejíž všechny vnitřní vrcholy jsou v množině $\{1, 2 \dots k\}$ (pro lib. pevné očíslování vrcholů)

$$\begin{aligned} \delta_k(i,j) &= c(i,j) \quad \text{pro } k = 0 \text{ (pouze přímé hrany)} \\ &= \min\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\} \text{ pro } k > 0 \end{aligned}$$

Pro graf s n vrcholy je stavový prostor $n \times n \times n$, tj. přímočaře paměť $O(n^3)$.
Hodnoty pro k závisí pouze na hodnotách pro $k-1$, proto můžeme počítat po vrstvách. Následně, ze struktury stavového prostoru stačí dvě vrstvy, tj. paměť $O(n^2)$. (Z chování alg. pak víme, že stačí jedna vrstva přepisovaná "na místě")
Algoritmus (popsaný u min. cest.) počítá zdola podle k, rekurzi jsme použili pouze při popisu vztahů.
Rekonstrukce optima: použili jsme speciální datovou strukturu pro rekonstrukci cesty odzadu, jako obvykle. Nebo podle rovnosti cen: předchozí vrchol leží na opt. cestě.

- možná/už znáte z programování: (optimální) násobení obdélníkových matic, (optimální) dělení mnohoúhelníku na trojúhelníky

Nutné vlastnosti úloh pro použití DP:
- optimální podstruktura
- překrývající se podproblémy

Obecná charakterizace úloh pro DP není známá.

Problém má optimální podstrukturu , pokud optimální řešení problému obsahuje pouze optimální řešení podproblémů. (Tuto vlastnost splňují i úlohy vhodné pro hladové algoritmy) (Bellmanův Princip optimality.)

Bellmanova rovnice popisuje, jak získáme optimální řešení z řešení podproblémů.

Optimální podstruktura často "přirozeně" určí prostor podproblémů. Z řešení podproblémů budeme sestavovat řešení větších problému. Chceme co nejmenší prostor podproblémů.

Důsledek: pro využití v celkovém výsledku si stačí pamatovat hodnotu nejlepšího řešení podstruktury (a případně jedno řešení pro rekonstrukci)

Překrývající se podproblémy. Pokud chceme aplikovat DP, počet různých podproblémů musí být malý, typicky polynomiální. (Chceme si pamatovat spočítaná řešení.) Pokud rekurzivní řešení počítá stejné problémy opakovaně, potom optimizační problém má *překrývající se podproblémy.*

Pro porovnání, úlohy řešené pomocí metody rozděl a panuj typicky generují stále nové podproblémy (např. používají různé části vstupu).

Na rozdíl od hladových algoritmů musíme prohledávat, protože nelze zaručit, že lokálně nejlepší volba je součásti globálního optima (tj. že lok. opt. řešení lze doplnit na optimální). Opt. řešení nelze převést na jiné opt.ř. lokálními úpravami, protože mezi opt. řešeními jsou (v obecnosti) "bariéry".

Protipř: *Nejdelší cesta v grafu.* Pokud si pamatuji jen koncové body, potom neplatí princip optimality: složením nejdelších cest nedostanu nejdelší *cestu.* Pokud si pamatuji i vnitřní body, mám nepolynomiálně mnoho podproblémů (a DP lze použít, ale má exponenciálně velký stavový prostor). Jak uvidíme další semestr, pro tuto úlohu není známý polynomiální algoritmus.

Techniky použité v DP:

Tabelace (memoization). Použijeme rekurzivní algoritmus, ale pamatujeme si výsledky spočítaných podproblémů

v tabulce. (Potřebujeme poznat všechny možné parametry podproblémů a určit jejich zobrazení do tabulky. Jinak lze použít hašování.) Tento přístup má výhodu, pokud nebudeme v rekurzi procházet celý prostor podproblémů a dokážeme ušetřit čas nebo paměť.

Výpočet zdola nahoru. Naplánujeme počítání hodnot v tabulce tak, že všechny potřebné podproblémy jsou vždy už vyřešené. Pokud každý podproblém budeme řešit aspoň jednou, výpočet zdola nahoru je obvykle efektivnější o multiplikativní konstantu, protože má menší režii.

Rekonstrukce řešení. Obvykle chceme znát řešení, proto si zapamatujeme optimální lokální rozhodnutí.

Při některých problémech (v závislosti na struktuře stavového prostoru) můžeme řešení průběžně zapomínat (pokud chceme znát pouze cenu), protože je už nebudeme potřebovat. Pokud si pamatujeme hodnoty řešení pro podstruktury, můžeme zrekonstruovat optimální řešení (trade-off čas/paměť) pomocí principu optimality.

Problém: **Editační vzdálenost.** (2022)

Definice: *Editační operace* na řetězci je přidání, vypuštění nebo změna jednoho znaku. *Editační vzdálenost* dvou řetězců $X = \langle x_1, x_2, ..., x_m \rangle$ a $Y = \langle y_1, y_2, ..., y_n \rangle$ je nejmenší počet editačních operací, které upraví řetězec X na Y. Označíme ji $L(X, Y)$.

Pozorování: v optimální posloupnosti operací se každého znaku týka nejvýš jedna operace. Proto operace můžeme uspořádat "zleva doprava", podle rostoucího indexu.

Značení: X_i je podposloupnost $X_i = \langle x_1, x_2, ..., x_i \rangle$ posloupnosti X. Řetězce X_i jsou *prefixy* X .

Tvrzení. Charakterizace struktury optimálního řešení.
1. Pro $x_i = y_j$ můžeme poslední znak skopírovat: $L(X_i, Y_j) = L(X_{i-1}, Y_{j-1})$.
2. Znak x_i změníme na y_j: $L(X_i, Y_j) = 1 + L(X_{i-1}, Y_{j-1})$.
3. Znak x_i smažeme: $L(X_i, Y_j) = 1 + L(X_{i-1}, Y_j)$.
4. Znak y_j vložíme: $L(X_i, Y_j) = 1 + L(X_i, Y_{j-1})$.

Hodnota $L(X, Y)$ pro nějaké řetězce závisí na editační vzdálenosti pro nějaké prefixy řetězců X a Y. Hodnota $L(X, Y)$ je nejlepší možnost z popsaných 4 případů. Hodnoty $L(X_i, Y_j)$ pro prefixy spočítáme rekurzivně. Poslední editační operace, resp. v případě 1 poslední kopírování, je nezávislé na předchozích operacích a proto stačí počítat optimální hodnoty pro prefixy. (Lepší hodnota $L(X, Y)$ by vynutila lepší hodnotu editační vzdálenosti pro některý prefix.) Proto má problém optimální podstrukturu.
Pokud je jeden z řetězců prázdný, potom editační vzdálenost je délka druhého. Tj. koncové podmínky jsou $L(X_0, Y_j) = j$ a $L(X_i, Y_0) = i$.

Pro vstupní řetězce X a Y délky m, resp. n voláme výpočet $L(X_m, Y_n)$. Přímočaré rekurzivní počítání může běžet exponenciálně dlouho. Ale stavový prostor je určen délkami prefixů, tj. máme $(n + 1)(m + 1)$ možností volání a s tabelací spotřebujeme čas $\Theta(mn)$, protože vlastní výpočet jedné hodnoty L je v konstantním čase.

Pozn.: Min. hodnota $L(X, Y)$ je jednoznačná, konkrétních

editačních posloupností může být víc, např. pro "ab"a "bc".

Iterativní řešení, tabulku T s hodnotami $T[i,j] = L(X_i, Y_j)$ vyplňujeme po řádcích:
Vstup: řetězce $X = \langle x_1, x_2, ..., x_m \rangle$ a $Y = \langle y_1, y_2, ..., y_n \rangle$
1. Pro i=0..m přiřadíme T[i,0] := i // init
2. Pro j=0..n přiřadíme T[0,j] := j // init
3. Pro i=1..m:
4. Pro j=1..n:
5. if $x_i = y_j$ then $\delta := 0$ else $\delta := 1$
6. T[i,j]:=min(δ+T[i-1,j-1], 1+T[i-1,j], 1+T[i,j-1])
Výstup: $L(X, Y) = T[m, n]$

Kromě hodnoty můžeme uschovávat (v pomocné tab. b[1..m,1..n]) operaci (případně směr), kterou jsme pro minimální hodnotu použili, . "Kopíruj"a "Změň"pro případy 1 a 2, "Smaž", resp. "Vlož"pro případ 3, resp. 4. To odpovídá členům v min pořadě: první s $\delta = 0$, první s $\delta = 1$, druhý, resp. třetí.

Rekonstrukce řešení: Podle popisů v tabulce b[]. Pokud tabulku b nemáme, rekonstruujeme řešení odzadu, podle směru, ve kterém platí shoda (v Bellmanově rovnici). Tento princip rekonstrukce je použitelný obecně.

V tomto problému: $T[i, j]$ závisí pouze na třech sousedních prvcích ve dvou řádcích. Pokud chceme pouze hodnotu, při výpočtu zdola nahoru si stačí pamatovat dva řádky. (DC: ještě lépe, jeden kratší řádek a dva prvky.) V tom případě ale pro (rychlou) rekonstrukci editační posloupnosti nemáme dost informací (trade-off).

Optimální binární vyhledávací strom. (2022)

Dosud jsme se snažili o vyvážené BVS, tj. (všechny) vrcholy jsou v malé hloubce ← neznáme pravděpodobnostní rozdělení, příp. frekvence, tj. předpokládáme rovnoměrné rozdělení.

Pokud známe frekvence (nebo pravděpodobnosti přístupu, $p_i = w_i / \sum_i w_i$), můžeme *spočítat* optimální strom.

Formálně: Pro n vstupních prvků $x_1 < x_2 < ... < x_n$ známe jejich váhy (frekvence) w_i. Konkrétní strom má prvek x_i v hloubce h_i (počítáno od 1) a jeho *cena* je $C = \sum_i w_i \cdot h_i$. Chceme najít *optimální vyhledávací strom*, tj. strom s minimální cenou.

Rozbor: Pokud si vybereme prvek x_i do kořene, tak jsou určeny prvky v levém a v pravém podstromě. Oba tyto podstromy musí být optimální, jinak je můžeme nezávisle vyměnit a tím celý strom zlepšit. Tedy máme vlastnost optimální podstruktury.

Pokud si vybereme prvky x_i a x_j, $i < j$, do kořene a jeho bezprostředního následníka, tak při obou možnostech potřebujeme pro spočítání ceny podstromy s prvky x_1 až x_{i-1}, x_{i+1} až x_{j-1} a x_{j+1} až x_n, případně prázdnými. Tedy máme opakující se podvýpočty. Povšimněme si, že zpracovávané úseky klíčů jsou spojité. Proto stavový prostor je určen krajními indexy i a j, pro $1 \leq i \leq j \leq n$ a jeho velikost je $O(n^2)$. Optimální hodnotu nechť vrací funkce OptStrom(i,j).

Pro optimum projdeme všechny možnosti kořene k a vy-

bereme minimální cenu. V OptStrom(i,j) procházíme k od i do j.

Výpočet OptStrom(i,j): pokud jsme už rekurzivně spočítali optimální cenu levého podstromu c_l a cenu pravého c_p, pak celková cena je $c_l + c_p + (w_i + \ldots\ldots + w_j)$. Levý i pravý podstrom klesl o 1, kořen je nově v hloubce 1, proto v závorce připočítáváme váhy všech prvků stromu jednou. Koncová podmínka je $i = j$ s optimální hodnotou w_i. (A)

Bellmanova rovnice:
Označme $W_{ij} = w_i + \cdots + w_j$, $C_{ij} = \text{OptStrom}(i, j)$, pak
$C_{ij} = \min_{i \le k \le j}(C_{i(k-1)} + C_{(k+1)j} + W_{ij})$ pro $i < j$,
$C_{ii} = W_{ii} = w_i$ a
$C_{ij} = 0$ pro $i > j$.

Rekurzivní řešení přímo z Bellmanovy rovnice. Výsledek je C_{1n}. Potřebujeme spočítat $O(n^2)$ hodnot C_{ij}, každou stihneme v $O(n)$, celkem tedy $O(n^3)$.

Hodnoty W_{ij} počítáme v $O(1)$, protože $W_{ij} = W_{i(j-1)} + w_j$ pro $i < j$.

Rekonstrukce řešení: rekurzivně. Pro každý podstrom si stačí pamatovat index optimálního kořene k. V nerekurzivním algoritmu si pamatujeme v poli K.

Pozn.: Závislosti v B. rovnici jsou "dlouhé" a současně využívají víc prvků st. prostoru. Hodnoty v tabulce nelze přepisovat, jak to šlo u "krátkých" závislostí.

Nerekurzivní řešení:

Vnější cyklus podle délek úseků.

alg. OptStrom2

Vstup: Klíče $x_1, x_2, ..., x_n$ s váhami $w_1, w_2, ..., w_n$.

01 Pro $i = 1, ..., n+1$: $T[i, i-1] := 0$ // zarážka
02 Pro $l = 1, ..., n$: // podle délky
03 Pro $i = 1, ..., n-l+1$: // zač. úseku
04 $j := i + l - 1$ // konec úseku
05 $W := w_i + ... + w_j$ // váha úseku
06 $T[i, j] := +\infty$ // init lok. ceny
07 Pro $k = i, ..., j$: // možné kořeny
08 $C := T[i, k-1] + T[k+1, j] + W$ // cena stromu
09 Pokud $C < T[i, j]$: // nové minimum?
10 $T[i, j] := C$ // nová cena
11 $K[i, j] := k$ // nový kořen

Výstup: Cena $T[1, n]$ optimálního stromu, pole K s optimálními kořeny

Složitost: Vnitřní cyklus 04-11 běží v čase $O(n)$ a spouští se pro $O(n^2)$ kombinací hodnot (a položek T). Celkem $O(n^3)$.

DC: Upravte hledání optimálního vyhledávacího stromu pro případ, kdy máte dány nejen váhy klíčů x_i, ale i váhy pro neúspěšné dotazy jako váhy externích listů, které odpovídají intervalům (x_i, x_{i+1}) mezi klíči.

Pozn. Pro tuto úlohu existuje speciální rychlý algoritmus v $O(n^2)$. (PLA 12.4, cv. 7)

Jiné úlohy DP:

Klasické:

Bitonická cesta v problému obchodního cestujícího Průchod doprava a doleva. Neposkytuje globálně optimální řešení, ale optimum v určité třídě řešení. (Bohužel, z toho nejde odvodit to globální optimum.)

Vyrovnaný tisk Rozdělit odstavec na řádky zarovnané "do bloku", tj. zarovnané vlevo i vpravo. Optimalizujeme druhé mocniny chyb, kromě posledního řádku.

Optimální uzávorkování násobení matic Násobíme posloupnost matic. Díky asociativitě můžeme různě uzávorkovat. Hledáme uzávorkování s min. cenou. Prostor podproblémů je určen začátkem a koncem spojitého úseku. Pamatujeme si pozici posledního násobení, tj. optimální rozdělení na dva úseky.

I méně klasické (z našeho pohledu):

Hry dvou hráčů s úplnou informací při acyklickém grafu tahů. Určujeme, zda je pozice (z našeho pohledu) vyhraná nebo prohraná (tzv. minimax). Ve vyhrané pozici si pamatujeme vyhrávající (optimální) tah. Když nemáme vyhrávající tah, je pozice prohraná.

Existence odvození v bezkontextových gramatikách St. prostor: Z neterminálu N existuje odvození úseku vstupu od i do j. Boolovské hodnoty: "Existuje"je lepší než "neexistuje". Rekurze podle pravidel gramatiky.

Součet podmnožiny (problém batohu) - přesný i co nejtěsnější. Například: St. prostor jsou možné součty

(cena), pamatujeme si existenci řešení pro danou cenu, řešení je podmnožina prvků, nemá polynomiální stav. prostor.

Nejdelší cesta. St. prostor: počáteční a koncový vrchol cesty a podmnožina mezilehlých vrcholů; není polynomiální. Cena je cena cesty. Úloha má optimální podstrukturu i opakující se podvýpočty. DP zlepší složitost z $O(n!)$ na exponenciální.

Hledání opt. strategie rozhodování v diskrétní optimalizaci pro konečný (i nekonečný) počet kroků. Např. optimalizace investic do SW nástrojů s různou cenou a návratností, které se navíc ovlivňují.

Zobecnění:

Nedeterministické/stochastické (vs. deterministické) modely.

Zpětnovazebné učení (v umělé inteligenci, angl. reinforcement learning) využívá Bellmanovu rovnici.

Spojitý čas (vs. diskrétní kroky).

Pozn. Tabelace nemusí být úplná. V hrách se používají pro stavy (hašovací) transpoziční tabulky, které řeší konflikty přepsáním; podle nějaké strategie (např. zůstává lepší; větší/pracnější; novější).

Při optimalizačních problémech si můžeme pamatovat min. a max. odhad řešení a postupně upřesňovat. Např. $a_{ij} = min_k(a_{ik} + a_{kj})$ (Lazy vyhodnocování bool. problémů je speciální případ. Např. $a_{ij} = \bigvee_k (a_{ik} \wedge a_{kj})$)

Problém: **Nejdelší společná podposloupnost - NSP.** (nepřednášeno 2022)

Definice. Pokud máme posloupnost $X = \langle x_1, x_2, ..., x_n \rangle$, její *podposloupnost* vznikne vypuštěním některých prvků.

Posloupnost Z je společná podposloupnost X a Y, pokud je Z podposloupnost X a zároveň Z je podposloupnost Y. Z je *nejdelší společná podposloupnost* X a Y, pokud je Z společná podposloupnost X a Y a neexistuje žádna delší společná podposloupnost. (NSP Z není určena jednoznačně, její délka je určena jednoznačně.)

Př.: X=ACBA, Y=BDAB, pak Z^1=AB, Z^2=BA, k=2.

Problém nejdelší posloupnosti je dán dvěma posloupnostmi $X = \langle x_1, x_2, ..., x_m \rangle$ a $Y = \langle y_1, y_2, ..., y_n \rangle$ a cílem je najít (jednu) nejdelší společnou podposloupnost $Z = \langle z_1, z_2, ..., z_k \rangle$ posloupností X a Y.

Značení: X_i je podposloupnost $X = \langle x_1, x_2, ..., x_i \rangle$ posloupnosti X.

Řešení hrubou silou: posloupnost délky m má 2^m podposloupností.

Věta. Charakterizace struktury optimálního řešení.
1. Pokud $x_m = y_n$, potom $z_k = x_m = y_n$ a Z_{k-1} je NSP X_{m-1} a Y_{n-1}.
2. Pokud $x_m \neq y_n$, potom $z_k \neq x_m$ znamená, že Z je NSP X_{m-1} a Y.
3. Pokud $x_m \neq y_n$, potom $z_k \neq y_n$ znamená, že Z je NSP X a Y_{n-1}.

Dk.

1. Pokud $z_k \neq x_m$, potom Z můžeme prodloužit, spor. Pokud Z_{k-1} není nejdelší, můžeme ji v Z nahradit delší, spor.
2. Pokud $z_k \neq x_m$, potom Z je společná podposloupnost X_{m-1} a Y. Pokud existuje W – delší společná podposloupnost X_{m-1} a Y, je taky společnou podposloupností X a Y a je delší než Z, spor.
3. Symetricky k 2.

Důsl. Problém má optimální podstrukturu.

Jiný prostor podproblémů (větší :-( ). Podproblémy byly výše charakterizovány koncem X_i a koncem Y_j, tj. dvojicí (i, j). Nejdelší podposloupnost můžeme počítat a rekonstruovat, pokud známe nejdelší společné podposloupnosti $X_{i'..i}$ a $Y_{j'..j}$. Podprostor podproblémů je charakterizován čtveřicemi (i', i, j', j).

Rekurzivní řešení.
Nechť c[i,j] je délka optimální podposloupnosti X_i a Y_j. Potom
1. $c[i, j] = 0$, pokud $i = 0$ nebo $j = 0$.
2. $c[i, j] = c[i - 1, j - 1] + 1$, pokud $i, j > 0$ a $x_i = y_j$.
3. $c[i, j] = max(c[i, j - 1], c[i - 1, j]$, pokud $i, j > 0$ a $x_i \neq y_j$.

Je pouze $\Theta(mn)$ různých podproblémů, hodnoty můžeme počítat zdola nahoru (od menších problémů k větším), např. po řádcích c[]. Výpočet jedné hodnoty potřebuje čas $O(1)$.

Kromě hodnoty můžeme uschovávat (v pomocné tab. b[0..i,0..j]) směr, odkud jsme maximální hodnotu použili. "Diag"pro případ 2, "Sloupec", resp. "Řádek"pro případ 3 - první, resp.

druhý člen v max.

Rekonstrukce řešení: Podle popisů v tabulce $b[]$. Pokud tabulku b nemáme, rekonstruujeme řešení odzadu, podle směru, ve kterém platí shoda. Tento princip rekonstrukce je použitelný obecně.

V tomto problému: $c[i, j]$ závisí pouze na třech sousedních prvcích ve dvou řádcích. Pokud chceme pouze hodnotu, pri výpočtu zdola nahoru si stačí pamatovat dva řádky. (DC: ještě lépe, jeden kratší řádek a dva prvky.) Pro (rychlou) rekonstrukci posloupnosti nemáme dost informací.

Podobný problém: *Hledání společné nadposloupnosti.*

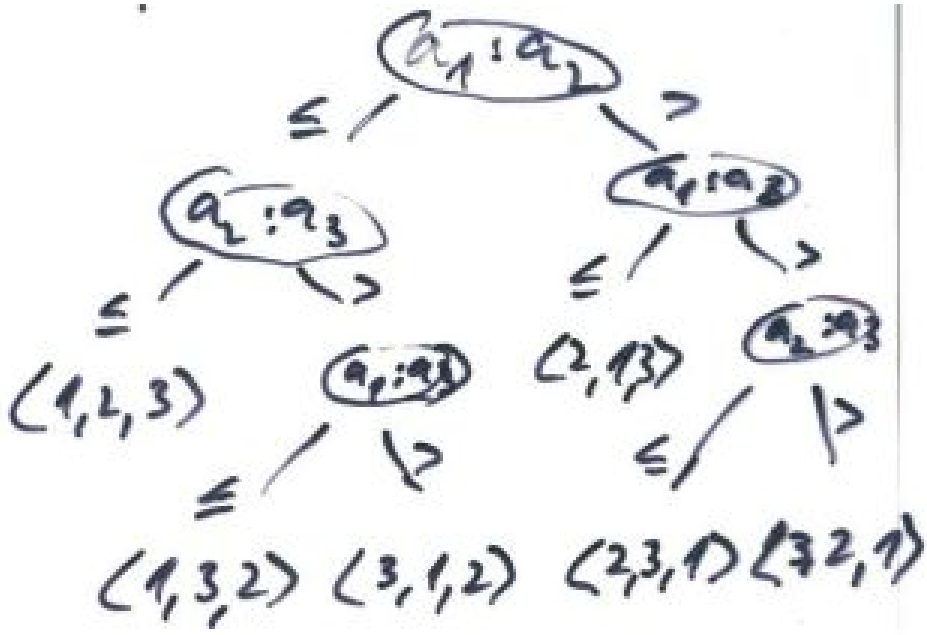
Pro dvě dané posloupnosti X a Y (nebo víc) chceme najít nejkratší nadposloupnost Z, která X i Y obsahuje jako podposloupnosti.

Stručně: stejný stavový prostor jako výše, stejná struktura závislostí, jiná Bellmanova rovnice, tj. jiný konkrétní výpočet položek.

Dolní odhad složitosti třídění založeného na porovnávání prvků

- rozhodovací strom: reprezentuje porovnávaní vykonaná při běhu třídícího algoritmu (na vstupu $a_1, a_2, \ldots, a_n$).

- vnitřní uzly jsou označené $a_i : a_j$ a odpovídají testu $a_i \leq a_j$, pro nějaké $1 \leq i, j \leq n$. Vnitřní uzly mají dva podstromy pro dva různé výsledky porovnání. (Hrany označené $\leq$ a $>$)



Obrázek 23: Rozhodovací strom (pro třídicí algoritmus)

- listy stromu jsou označené permutací $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$, která odpovídá výsledku $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$

- Běh algoritmu odpovídá cestě z kořene do listu. Složitost algoritmu v nejhorším případě je hloubka stromu.

Pozorování: V listech se musí objevit všech $n!$ permutací, tj. #listů$\geq n!$. Pokud se nějaká permutace neobjeví, jsme schopni na vstup předložit inverzní permutaci a algoritmus ji nedokáže utřídit, tedy *není korektní.*

V: Libovolný rozhodovací strom pro n prvků má výšku

$\Omega(n \log n)$.

Dk. Pro strom hloubky h platí $n! \leq 2^h$, protože 2^h je max. počet listů. Odtud zlogaritmováním $\log(n!) \leq h$.

Odhadneme $n!$: $n! \geq n \cdot (n-1) \cdot \ldots \cdot 2 \geq n \cdot (n-1) \cdot \ldots \cdot (\frac{n}{2}) \geq \frac{n}{2}^{\frac{n}{2}}$, odtud $h \geq \log(n!) \geq \log(\frac{n}{2})^{\frac{n}{2}} \geq \frac{n}{2} \log \frac{n}{2} = \frac{1}{2} n(\log n - 1) \in \Omega(n \log n)$

Důsl. Heapsort, Mergesort (a Quicksort v průměrném případě) jsou optimální třídící algoritmy.

Analýza quicksortu

Procedure Quicksort(M)
begin
if $|M| > 1$ then
  pivot := nějaký prvek z M
  $M_1 := \{m | m < pivot\}$
  $M_2 := \{m | m \geq pivot\} \backslash \{pivot\}$
  Quicksort(M_1) – na místě
  Quicksort(M_2) – na místě
end

Analýza složitosti: $T(n)$ je čas zpracování n prvků
T(0) = T(1) = 0
T(n)= 1 +n+T(n-k)+T(k-1), pivot je k-tý

nejlepší případ $k \doteq \frac{n}{2}$
$T(n) = 2 \cdot T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$

nejhorší případ $k = 1$ nebo $k = n$
$T(n) = 1 + n + T(n-1) \Rightarrow T(n) = O(n^2)$

Potřebujeme předpoklady o pravděpodobnostním rozložení:
- na vstupu jsou permutace čísel $1..n$, všechny se stejnou

pravděpodobností
- proto: $pivot = k$, pro $\forall k$ se stejnou pravděpodobností
- pro vytvořené posloupnosti M_1 a M_2 potřebujeme zaručit (pro rekurzi), že jsou náhodné permutace: vhodnou volbou algoritmu

Očekávaná doba výpočtu $ET(n)$:
$ET(0) = ET(1) = 0$
$ET(n) = \sum_{k=1}^{n} \frac{1}{n}(n+1+ET(k-1)+ET(n-k))$ pro $n \geq 2$, sloučení sum
$ET(n) = n+1+\frac{2}{n}\sum_{k=0}^{n-1}(ET(k))$, $(\cdot n)$
$n \cdot ET(n) = n(n+1)+2\sum_{k=0}^{n-1}(ET(k))$, (1)
$(n+1)\cdot ET(n+1) = (n+2)(n+1)+2\sum_{k=0}^{n}(ET(k))$,
dosadíme $n \leftarrow n+1$ v (1); (2)
Spočteme: (2)-(1):
$(n+1)\cdot ET(n+1) - n\cdot ET(n) = 2(n+1)+2ET(n)$,
...
$ET(n+1) = 2+\frac{(n+2)}{(n+1)}ET(n)$, ...
$ET(n) = \sum_{i=2}^{n} 2\cdot\frac{(n+1)}{(i+1)} = 2(n+1)(H_{n+1}-3/2)$
$\leq 2(n+1)\cdot H_{n+1} \approx 2(n+1)\cdot\log(n+1)$
tedy $ET(n) = O(n\log n)$, když použijeme fakt, že n-té harmonické číslo H_i je pro velké n přibližně rovno $\log n$.
$H_n = \sum_{i=1}^{n}\frac{1}{i}$

- pivot nemusí být prvek vstupní posloupnosti
- Quicksort jako *schéma* algoritmů podle strategie volby pivota
- př.: heuristika z praxe, vybereme pivota jako medián z

prvků x_1, $x_{\lfloor \frac{n}{2} \rfloor}$ a x_n. Nerovnoměrné rozdělení pivotů.
(- DC: Při tomto výběru pivota:
1/ Jaká je pravděpodobnost, že pivot bude v prostřední třetině?
2/ Jaká je pravděpodobnost, že pivot bude v prostředních dvou čtvrtinách?
3/ Jaká je střední hodnota pořadí pivota, pokud pivot bude vybrán z dolní třetiny vstupů?
4/ Jaká je pravděpodobnost, že pivot bude medián?)
- pro libovolný pevný způsob výběru pivota existuje (tj. inteligentní nepřítel dokáže zkonstruovat) posloupnost délky n s časem třídění $O(n^2)$
- očekávaná průměrná hloubka náhodného bin. stromu: $O(\log n)$, vztah k možným průběhům Quicksortu: vybraný pivot je první prvek z příslušného intervalu, určeném minulými pivoty. Vkládaný vrchol x je vložen na hloubce, která odpovídá počtu porovnání x s minulými pivoty, tj. vrcholy v BVS nad ním. Počet porovnání v BVS a Quicksortu je stejný, jen jsou jinak strukturována. Celkový počet porovnání v Quicksortu je průměrně $O(n \log n)$, proto hloubka stromu je v průměru $O(\log n)$.

Randomizovaný Quicksort

- randomizovaný: znáhodněný, (pravděpodobnostní) - problém pevného výběru pivota: určité vstupní posloupnosti se třídí (vždy) v čase $O(n^2)$. Pokud se nevhodné posloupnosti vyskytují na vstupu s větší pravděpodobností, pak očekávaný čas se může blížit $O(n^2)$.

- řešení: volíme pivota náhodně

proto: Pro každou vstupní posloupnost má algoritmus očekávanou složitost $O(n \log n)$. (Počítáme jako průměr časů dosažených při všech volbách dělících bodů.)

Závěr: Pro randomizovaný Quicksort neexistují špatné vstupy, ale pro konkrétní vstup můžeme zvolit špatné pivoty (špatná volba vždy existuje), kdy doba výpočtu je $O(n^2)$. (Randomizovaný Quicksort nevyžaduje rovnoměrné rozdělení vstupů jako deterministický Quicksort.)

Poznámky: (jiný pohled)

Když je pivot vždy vybrán tak, že rozdělí posloupnost v poměru 99 : 1 (nebo lepším)

$T(n) = T(99/100n) + T(n/100) + (n - 1)$

Řešení (subst. metodou): $T(n) = \Theta(n \log n)$

Stejnou složitost dostaneme pro každý konstantní poměr dělení, tj. poměr nezávislý na n (Stačí, když se tento poměr dosáhne aspoň v jedné ze dvou (obecně z pevného počtu k) úrovní.)

Neformálně, pivota jako pseudomedián mezi 1/4 a 3/4 délky posloupnosti dostaneme s pravděpodobností 1/2. Průměrně se nám taková volba podaří v každé druhé úrovni. (Víme z hašování.) Délka delší z posloupností klesá geometrickou řadou s kvocientem 3/4.

Hledání k-tého prvku

- neutříděný vstup, chceme pouze k-tý minimální prvek
- zahrnuje hledání mediánu, tj. $k = \lceil \frac{n}{2} \rceil$

Ukážeme si:

• *Quickselect*: hledání v průměrně lineárním čase

• *Linearselect*: hledání v (deterministicky) lineárním čase

Přímočaré alg.: (pro malé nebo pevné k)

• alg. založený na insertsortu: udržuju si k nejmenších prvků, $O(n.k)$

• alg. založený na heapsortu: z haldy vyberu k nejmenších prvků, $O(n + k \log n)$

Alg. Quickselect:

- zal. na Rozděl a panuj,
- pro jednoduchost: na vstupu jsou různé prvky
- myšlenka: podle pivota rozdělíme vstup na menší prvky vlevo a větší vpravo, části značíme L a P, potom rekurzivně zpracujeme část, která obsahuje k-tý prvek

Alg. Quickselect (najde k-tý nejmenší prvek)

Vstup: posl. $X = x_1, ..., x_n$ prvků a k, kde $1 \leq k \leq n$

1 Pokud $n = 1$: vrať $y := x_1$ a skonči.

2 $p := x_i$, některý prvek z X; volba pivota

3 $L :=$ prvky z X menší než p

4 $P :=$ prvky z X větší než p

5 Pokud $k \leq |L|$: $y := Quickselect(L, k)$.

6 Jinak, je-li $k = |L| + 1$: $y := p$.

7 Jinak: $y := Quickselect(P, k - |L| - 1)$.

Výstup: y je k-tý nejmenší prvek v X

Rozbor:
Při náhodné volbě pivota (nebo pevné volbě s náhodným vstupem, tj. dobře zamíchaným) je polovina prvků pseudomediány. (Podobně jako u Quicksortu:) pseudomedián volíme průměrně v každém druhém pokusu a on způsobí odstranění aspoň 1/4 vstupu. Proto délka vstupů v rekurzi klesá geometrickou řadou a celková složitost je v průměru $\Theta(n)$.

Alg. Linearselect
- myšlenka: jako Quickselect, ale najdeme deterministicky lineárně dobrý odhad pivota
- trik: vstup rozdělíme na pětice, poslední bude neúplná (případně doplníme velkými prvky), z pětic spočítáme mediány a z nich spočítáme medián. Ten použijeme jako pivota a pokračujeme jako v alg. Quickselect.
(- proč pětice?: protože s nimi to funguje)

Alg. Linearselect (najde k-tý nejmenší prvek v lin. čase)
Vstup: posl. $X = x_1, ..., x_n$ prvků a k, kde $1 \leq k \leq n$
1 Pokud $n \leq 5$: vyřešíme triviálně.
2 Prvky rozdělíme na pětice $P_1, ..., P_{\lceil n/5 \rceil}$
3 Spočítáme mediány pětic: $m_i := P_i$.
4 Najdeme pivota $p := Linearselect(m_1, ..., m_{\lceil n/5 \rceil}; \lceil n/10 \rceil)$.
5 $L, P :=$ prvky z X menší než p, větší než p
6 Pokud $k \leq |L|$: $y := Linearselect(L, k)$.
7 Jinak, je-li $k = |L| + 1$: $y := p$.
8 Jinak: $y := Linearselect(P, k - |L| - 1)$.
Výstup: y je k-tý nejmenší prvek v X

Vysvětlení: (obr.)
Vybraný prvek p je blízko mediánu: aspoň 3/10 prvků jsou menší a aspoň 3/10 jsou větší. Menší jsou 3 nejmenší prvky z pětic, které mají mediány pod p, a dva prvky v pětici s p. Větší prvky symetricky. Následně, části L i P mají nejvýš $7/10.n$ prvků.

Složitost:
- rozdělení na pětice a počítání mediánů: $\Theta(n)$
- dělení na části podle p a rozhodování, do které části se vydat: $\Theta(n)$
- volání Linearselect na ř. 4 na $n/5$ prvků
- volání Linearselect na ř. 6 nebo 8 na levou nebo pravou část s max. $7/10.n$ prvků

Z toho dostanene pro nejhorší případ rekurentní rovnici (pro vhodně velkou jednotku času)

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/5) + T(7/10.n) + n. \end{aligned}$$

Master Teorem nepomůže, ale substituční metoda je použitelná a tento konkrétní příklad jsme tam spočítali. Uhádneme řešení $T(n) = cn$ a ověříme:

$$\begin{aligned} cn &= 1/5.cn + 7/10.cn + n \\ &= 9/10.cn + n. \end{aligned}$$

Rovnost platí pro $c = 10$ a alg. Linearselect je opravdu lineární.

Pozn.:
Lineární hledání mediánu umožní v *Quicksortu* spočítat me-

dián tímto způsobem a potom celé třídění běží v $O(n \log n)$ i v nejhorším případě. Ale konstanty v složitosti jsou velké a proto se tímto způsobem nepoužívá.

(Nevhodné) dělení na *trojice*: medián hledáme rekurzivně pro $n/3$ prvků. Můžeme odstranit $2/6.n = n/3$ prvků a pro rekurzivní volání zbude $2/3.n$ prvků. Vyjde rekurentní rovnice

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/3) + T(2/3.n) + n, \end{aligned}$$

která nemá lineární řešení (ale $O(n \log n)$).

Lineární algoritmy třídění (nepřednášeno 2022)

Radix sort - přihrádkové třídění (původní motivace: třídění děrných štítků)

- třídíme podle jedné cifry (1 sloupce) do p přihrádek ($p = 10$ pro decimální cifry) a skupiny poskládáme za sebe.

- Pozorování: pokud byly přihrádky předem utříděny podle méně významných cifer a použité třídění bylo stabilní, máme utříděnou posloupnost.

pozn: Stabilní třídění zachovává pořadí prvků se stejným klíčem ze vstupu na výstupu.
$\langle a_1, b_1, a_2, c_1, b_2, c_2\rangle \rightsquigarrow \langle a_1, a_2, b_1, b_2, c_1, c_2\rangle$

Alg.: pro d-místná čísla (1. cifra nejnižší, d-tá cifra nejvyšší řád)

```
for i=1 to d do
    utřiď stabilním tříděním podle i-té cifry
```

Složitost: $O(d(n + p))$; d se předpokládá konstantní.

Aplikace: třídění řetězců/slov a strukturovaných klíčů (datum = (rok, měsíc, den)) lexikograficky.

Doplnění/zarovnání:

- Při třídění slov různé délky doplňujeme mezerami zprava
- Při třídění čísel doplňujeme nulami zleva

Counting sort

- omezení: na vstupu I[1..n] čísla v rozsahu 1-k, přirozená.
- pomocná paměť: pole C[1..k]; pole výsledků O[1..n]

Idea alg.:

`for i:=1 to k do C[i]:=0` 0. průchod C: init C

`for i:=1 to n do C[I[i]]++` 1. průchod I: do pole C nasčítáme počet výskytů vstupních čísel z I

`for i:=2 to k do C[i]+=C[i-1]` 2. průchod C: sečtení zdola: C[x] obsahuje počet výskytů menších nebo rovných čísel než x, tj. (konec) umístění výsledků v O

`for i:=n to 1 do O[C[I[i]]]:=I[i]; C[I[i]]--` 3. průchod I, odzadu: uložení vstupního čísla x na C[x]-té místo O, posun pozice dolu

Složitost čas: $O(k+n)$, paměť: $O(k+n)$, předpokládáme $k = O(n)$

Doplňková vlastnost: stabilita třídění, protože ve 3. průchodu jdeme odzadu a indexy `C[x]` po 2.druhém průchodu ukazují na konec úseku s hodnotami `x`.

- pozn.: ve 3. průchodu nestačí vygenerovat příslušný počet indexů `x` do výstupu `O` podle hodnot v `C[x]`, protože nás zajímají i přidružená data v `I`.

DC: Upravte alg. tak, aby 3. průchod byl zepředu (např. streamovaná data z komprese nebo serializace, z mag. pásky `:-)`) a dostali jste stabilní výstup.

LUP dekompozice

Df: Matice je dvourozměrné pole prvků. Matice $A = (a_{ij})$ o rozměrech $m \times n$ má m řádků a n sloupců. Pokud jsou prvky z množiny S, potom množinu matic označujeme $S^{m \times n}$.

příklad matice A: $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

- Transponovaná matice A^T vznikne výměnou sloupců a řádků matice A, tj. matice $A^T = (a_{ji})$.

- Nulová matice má všechny prvky 0. Rozměry matice lze obvykle určit z kontextu.

- Vektory jsou sloupcové matice $n \times 1$. (Řádkové získáme transpozicí.)

- Jednotkový vektor e_i je vektor, který má i-tý prvek 1 a všechny ostatní 0.

- Čtvercová matice $n \times n$ se objevuje často. Speciální prípady čtvercových matic jsou:

- Diagonální matice má $a_{ij} = 0$ pro $i \neq j$.

- Jednotková matice I_n rozměrů $n \times n$ je diagonální matice s jedničkami na uhlopříčce. Sloupce jsou jednotkové vektory e_i. Píšeme I bez indexu, pokud lze rozměry odvodit z kontextu.

- Horní trojuhelníková matice U má $u_{ij} = 0$ pro $i > j$ (hodnoty pod diagonálou jsou 0). Jednotková horní trojúhelníková matice má navíc na diagonále pouze jedničky.

- Dolní trojúhelníková matice L má $l_{ij} = 0$ pro $i < j$ (hodnoty nad diagonálou jsou 0). Jednotková dolní trojúhelníková matice L má navíc na diagonále pouze 1.

- Permutační matice P má právě jednu 1 v každém řádku

a sloupci a 0 jinde. Název permutační pochází z toho, že násobení vektoru x permutační maticí permutuje (přeháže) prvky x.

- Symetrická matice A splňuje $A = A^T$.

- Inverzní matice k $n \times n$ matici A je matice rozměrů $n \times n$, označovaná A^{-1} (pokud existuje), tž. platí $AA^{-1} = I_n = A^{-1}A$. Matice, která nemá inverzní matici, se nazývá singulární (nebo neinvertovatelná), jinak se nazývá nesingulární (nebo invertovatelná). Inverze matice, pokud existuje, je jednoznačná (DC).

Řešení soustav lineárních rovnic, pomocí LUP dekompozice.

Máme soustavu rovnic $Ax = b$, tj. pro $A = (a_{ij})$, $x = (x_j)$ a $b = (b_i)$

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

Pro dané A a b hledáme řešení x soustavy. Řešení může být i několik (málo určená soustava) nebo žádné (přeurčená soustava).

Pokud je A nesingulární, existuje A^{-1} a $x = A^{-1}b$, protože $x = I_n x = A^{-1}Ax = A^{-1}b$. Řešení x je potom jediné (DC).

Možná metoda řešení: spočítáme A^{-1} a následně x. Ale tento postup je numericky nestabilní, tj. zaokrouhlovací chyby se kumulují při práci s počítačovou *reprezentací* reálných čísel.

(Problémy:
1. Numerická nestabilita. V LUP omezujeme volbou (absolutní hodnotou) velkého pivota s použitím matice P,
2. Špatně podmíněný (ill posed) problém : malá změna vstupních dat (např. zaokrouhlení) způsobí velkou změnu výsledků - "efekt motýlích křídel". (opak je dobře podmíněný, well posed),
3. Testování reálných čísel na 0 (LUP dekompozice: ř. 10). Vlivem zaokrouhlovacích chyb vyjde nenulová hodnota tam, kde měla vyjít 0. Následně: lineárně závislé vektory se stanou lineárně nezávislé, s malou "odchylkou".)

Metoda LUP: pro A najdeme tři matice L, U, P rozměrů $n \times n$, tzv. *LUP dekompozici*, tž. $PA = LU$, kde

- L je jednotková dolní trojúhelníková matice
- U je horní trojúhelníková matice
- P je permutační matice

Soustava $PAx = Pb$ odpovídá přehození rovnic. Použitím dekompozice máme $LUx = Pb$ a řešíme trojúhelníkové soustavy. Označme $y = Ux$. Řešíme $Ly = Pb$ pro neznámý vektor y metodou dopředné substituce a potom pro známé y řešíme $Ux = y$ pro x metodou zpětné substituce. Vektor x je hledané řešení, protože P je invertovatelná a $Ax = P^{-1}LUx = P^{-1}Pb = b$.

Dopředná substituce řeší dolní trojúhelníkovou soustavu v čase $\Theta(n^2)$ pro dané L, P, b.

Označme $c = Pb$ permutaci vektoru b, $c_i = b_{\pi(i)}$. Řešená soustava $Ly = Pb$ je soustava rovnic

$$\begin{aligned}
y_1 &= c_1 \\
l_{21}y_1 + y_2 &= c_2 \\
l_{31}y_1 + l_{32}y_2 + y_3 &= c_3 \\
&\vdots \\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots + y_n &= c_n
\end{aligned}$$

Hodnotu y_1 známe z první rovnice a můžeme ji dosadit do druhé. Dostáváme

$y_2 = c_2 - l_{21}y_1$.

Obecně, dosadíme $y_1, y_2, \ldots, y_{i-1}$ "dopředu"do i-té rovnice a dostaneme y_i:

$$y_i = c_i - \sum_{j=1}^{i-1} l_{ij} y_j$$

Zpětná substituce je podobná dopředné substituci a řeší horní trojúhelníkovou soustavu v čase $\Theta(n^2)$ pro dané U a y. Soustava má tvar

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 \\ u_{22}x_2 + \ldots + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 \\ &\vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \\ u_{nn}x_n &= y_n \end{aligned}$$

Řešíme postupně pro $x_n, x_{n-1}, \ldots, x_1$ takto:

$x_n = y_n/u_{nn}$,

$x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}$,

obecně

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii} \ .$$

Program LUP-solve: přepisem vzorců. Permutační matice P je reprezentována polem $\pi[1..n]$, kde $\pi[i] = j$ znamená, že i-tý řádek P obsahuje 1 v j-tém sloupci.

Složitost LUP-solve: $\Theta(n^2)$ celkem, pro dopřednou i pro zpětnou substituci. V obou případech vnější cyklus probíhá proměnné a vnitřní cyklus počítá sumu, která prochází část řádku.

Výpočet LU dekompozice. Nejprve jednodušší prípad, když matice P chybí (tj. $P = I_n$).

Idea metody: Gaussova eliminace, při které vhodné násobky prvního řádku přičítáme k dalším řádkům tak, abychom odstranili x_1 z dalších rovnic (koeficienty u x_1 v prvním sloupci budou nulové). Potom pokračujeme (rekurzivně) v dalších sloupcích, až vznikne horní trojúhelníková matice, tj. U. Matice L vzniká z koeficientů, kterými jsme násobili řádky.

Z matice A oddělíme první řádek a sloupec, potom matici rozložíme na součin. Matice A' je $(n-1) \times (n-1)$ matice, v sloupcový vektor a w^T řádkový vektor a součin vw^T je také $(n-1) \times (n-1)$ matice.

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix}$$

Podmatice $A' - vw^T/a_{11}$ rozměrů $(n-1) \times (n-1)$ se nazývá *Schurův komplement* A vzhledem k a_{11}.

Rekurzivně najdeme LU rozklad Schurova komplementu, nech je roven $L'U'$.

S využitím maticových operací odvodíme

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU \end{aligned}$$

Matice L a U jsou jednotková dolní trojúhelníková matice a horní trojúhelníková matice, protože L' a U' jsou požadovaného tvaru.

Program LU dekompozice - přepisem vzorců. (Převádí tail-rekurzivní strukturu na iteraci-cyklus.)

Složitost: $\Theta(n^3)$, protože počítáme n-krát Schurův komplement, který má $\Theta(k^2)$ prvků pro $k = 0..n-1$. Výpočet jedné úrovně rekurze (tj. hlavního cyklu) trvá $\Theta(k^2)$ a celkový čas lze odhadnout $\sum_{k=0}^{n-1} k^2 = \Theta(n^3)$ (DC).

Pokud $a_{11} = 0$, metoda nefunguje, protože se dělí nulou. Prvky, kterými dělíme, nazýváme *pivoty* a jsou na diagonále U. Zavedení matice P nám umožňuje se vyhnout dělení nulou (nebo malými čísly – kvůli zaokrouhlovacím chybám) a vybrat si ve sloupci nenulový prvek. Takový musí existovat, pokud je matice nesingulární.

Implementační poznámky - optimalizace: a) stačí počítat nenulové prvky, b) obě matice můžeme uložit "na místě", pokud ukládáme pouze významné prvky, tj. nenulové a diagonálu U.

Program LUP-dekompozice

```
LUP-dekompozice(A)
 1 n <- rows[A]          % počet řádků
 2 for i <- 1 to n       % P inicializujeme
 3   do pi[i] <- i       %  jako diagonální I_n
 4 for k <- 1 to n-1     % hlavní cyklus
 5   do p <- 0           % nulování pivota
 6      for i <- k to n           % výběr pivota
 7        do if |a[ik]| > p       % test vel. pivota
 8           then p <- |a[ik]|    % bereme maximum
 9                k' <- i         % pozice pivota
10      if p = 0              % test pivota
11        then error "singular matrix" k  % chyba
12      exchange pi[k] <-> pi[k'] % změna P tj. pi[]
13      for i <- 1 to n           % výměna řádků
14        do exchange a[ki] <-> a[k'i]  % matice A
15      for i <- k+1 to n             % přes řádky
16        do a[ik] <- a[ik]/a[kk]  % k-tý sloupec L
17           for j <- k+1 to n     % v řádku i
18             do a[ij]<-a[ij]-a[ik]*a[kj] %změna U
```

Složitost: tři vnořené cykly (18) $\rightarrow \Theta(n^3)$

Počítání inverze pomocí LUP-dekompozice.

Pokud máme LUP rozklad matice A, dokážeme spočítat pro dané b řešení $Ax = b$ v čase $\Theta(n^2)$. LUP rozklad totiž nezávisí na b.

Rovnici $AX = I_n$ můžeme považovat za n různých soustav tvaru $Ax = b$ pro $b = e_i$ a $x = X_i$, kde X_i znamená i-tý sloupec X. Řešení každé soustavy nám dá sloupec matice $X = A^{-1}$.

Složitost: řešíme n soustav rovnic, každou v čase $\Theta(n^2)$. Výpočet LUP dekompozice spotřebuje čas $\Theta(n^3)$, teda celkem inverzi A^{-1} matice A spočítáme v čase $\Theta(n^3)$.

Souvislosti:

Lze ukázat: $T_{inverze}(n) = \Theta(T_{nasobeni}(n))$

Tj. složitost počítání inverze je stejná jako násobení matic.

(Převod maticového násobení na inverzi)

(Redukce inverze na násobení)

...

Pořadí témat 2019, dotace předn.:poř. předn:

- Prostředky pro popis složitosti 0,5:1
- Základní grafové alg. 2:2-3
- Min. cesty 2:4-5
- Min. kostry 1,5: 6-
- Stromové struktury 1,5:7
- Rozděl a panuj 2:8-9
- Třídění 1,5:10+
- Hašování 1:11+
- Lin. Alg. 1:12+
- - Celkem: 13P:...
-
-