Algoritmy a datové struktury*

RNDr. Jan Hric

Obsah

*sepsal a vypracoval Petr Hošek

1 Asymptotická notace

Porovnávání algoritmů

- časová složitost
- prostorová složitost

Závisí na velikosti vstupních dat.

Odstranění závislosti na konkrétních datech

- v nejhorším případě
- v průměrném případě (vzhledem k rozložení)
- v nejlepším případě

Měření velikosti vstupních dat Obvykle počet bitů nutných k napsání dat.

Příklad Vstupem je posloupnost $a_1, a_2, \dots a_n \in \mathbb{N}$, velikost dat D v binárním zápisu je $\sum \lceil log_2 a_i \rceil$.

1.1 Časová složitost algoritmu

Funkce $f : \mathbb{N} \to \mathbb{N}$ taková, že $f(|D|)$ udává počet kroků algoritmu, který je spuštěn na datech D.

Co je krok algoritmu?

teoreticky operace daného abstraktního stroje (Turingův stroj, stroj RAM)

prakticky (zjednodušeně) operace proveditelné v konstantním čase

pro nás aritmetické operace ($\pm$, $\times$, $\div$, ...), porovnávání 2 čísel (hodnot), přiřazení pro jednoduché typy

1.2 Asymptotická složitost

Zkoumá chování algoritmu na velkých datech. Zařazuje algoritmy do kategorií. Zanedbání multiplikativní a aditivní konstanty.

Značení

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

Asymptotická složitost dovoluje konečný počet výjimek. Typicky

$$\text{největší vyjímka } +1 = n_0.$$

Tato technika je obecná, používá se používá tedy na časovou i paměťovou složitost.

Amortizovaná složitost Nepočítáme dobu jedné operace, ale počítáme celkovou dobu a tu amortizujeme počtem operací.

2 Binární vyhledávací stromy

Vrcholy obsahují klíč, ukazatel na levého, pravého následníka, rodiče.

Vlastnost binárních stromů (invariant) x je vrchol v binárním stromě. Pokud y je vrchol v levém podstromě x, potom klíč(y) $\leq$ klíč(x). Pokud y je vrchol v pravém podstromě x, potom klíč(y) $\geq$ klíč(x).

Operace

- $find/search(S, k)$
- $min(S)$
- $max(S)$
- $preceddessor(S, x)$
- $successor(S, x)$
- $insert(S, x)$
- $delete(S, x)$

Složitost operací stromu s n vrcholy Časová složitost operací je v nejhorším případě úměrná výšce stromu.

nejlepší případ vyvážený (úplný) strom, složitost $\Theta(\log n)$

nejhorší případ lineární seznam n vrcholů, složitost $\Theta(n)$

náhodně postavený strom výška $\Omega(\log n)$, složitost $\Omega(\log n)$

Typicky máme náhodně postavený strom, vyvážený strom stavíme většinou v případě kdy víme, že se již nebude měnit.

3 Červeno-černé stromy

Jsou to binární vyhledávací stromy které mají navíc obarvené vrcholy, barva *červená* nebo *černá* (implementačně 1 bit informace).

Podmínka obarvení Délka cest z kořene do listu se liší nejvíce 2×, stromy jsou přibližně vyvážené.

Binární vyhledávací strom je *červeno-černý strom*, pokud splňuje následující vlastnosti

1. každý vrchol je červený nebo černý
2. každý list (jako listy bereme v tomto případě ukazatele na *nil*) je černý
3. pokud je vrchol červený, jsou oba potomci černí
4. každá cesta z vrcholu do podřízeného listu obsahuje stejný počet černých vrcholů

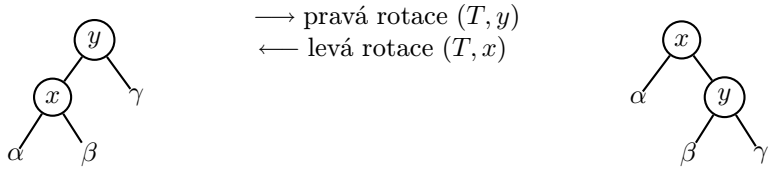
Lemma Červeno-černý strom s n vnitřními vrcholy má výšku h nejvíce $2\log(n+1)$.

Důkaz Indukcí podle hloubky podstromů. Podstrom ve vrcholu x má aspoň $2^{b(x)}-1$ vnitřních vrcholů, kde $b(x)$ je černá výška x (počet černých vrcholů na cestě z x do listů, kromě x). Použijeme pro kořen: $n \geq 2^{h/2}-1 \Rightarrow h \leq 2\log(n+1)$.

3.1 Operace

3.1.1 Rotace

Potřebné pro obnovení vlastností při operacích *insert* a *delete*.



Platí v obou stromech

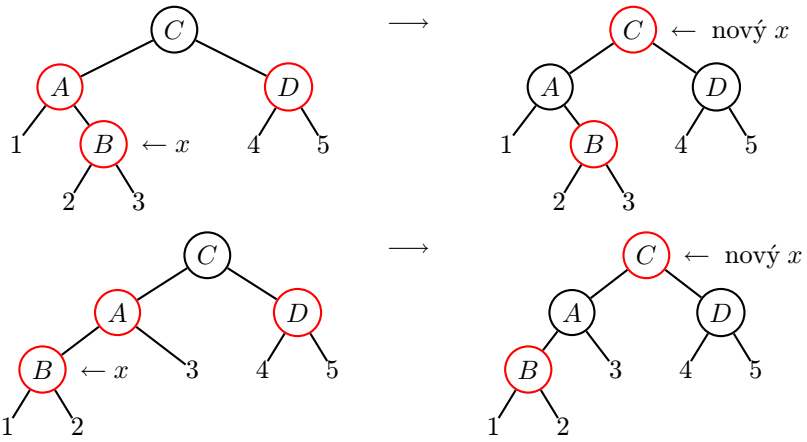
- $\forall z \in \alpha \quad \text{klíč}(z) \leq \text{klíč}(x)$
- $\forall z \in \beta \quad \text{klíč}(x) \leq \text{klíč}(z) \leq \text{klíč}(y)$
- $\forall z \in \gamma \quad \text{klíč}(y) \leq \text{klíč}(z)$

3.1.2 Vkládání

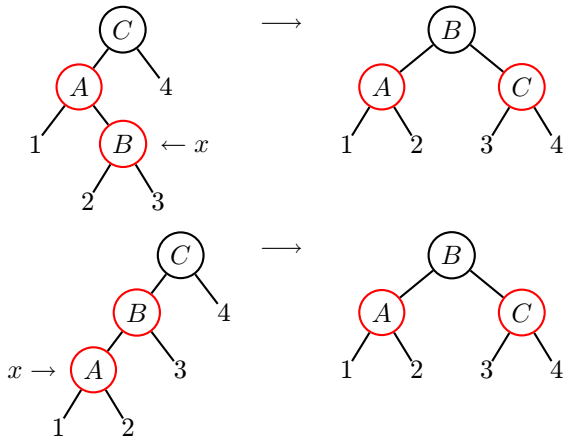
Najdeme správné místo pro x a přidáme do listu, obarvíme *červeně*. Mohla vzniknout porucha vlastnosti 3, mezi x a otcem x. Musíme opravit strom. Idea opravy je, že odstraníme problém lokálně, nebo ho posuneme výš.

Rozbor

1. x je kořen, přebarvíme na černo
2. $father(x)$ je černý, strom je v pořádku
3. $father(x)$ je červený, $grandfather(x)$ existuje a je černý
   (a) $uncle(x)$ je červený, posun poruchy nahoru



(b) $uncle(x)$ je černý, lokální odstranění poruchy

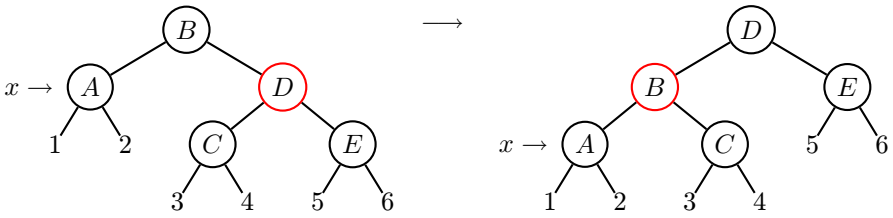


3.1.3 Vypouštění

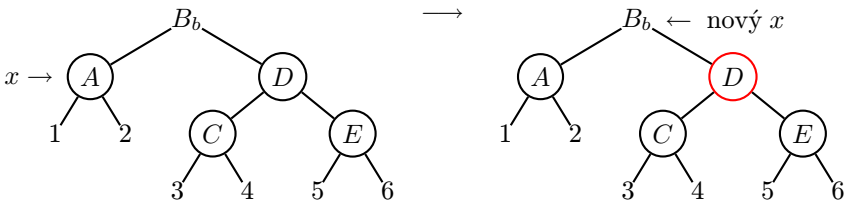
Vypouštíme x z T. Pokud má x oba syny, najdeme následníka x v T a přemístíme ho do x. Vzniká tak porucha kdy vrchol je dvojitě černý. Odstraňuji poruchu, ale lokálně zůstává černá hloubka stejná.

Odstraňování poruchy

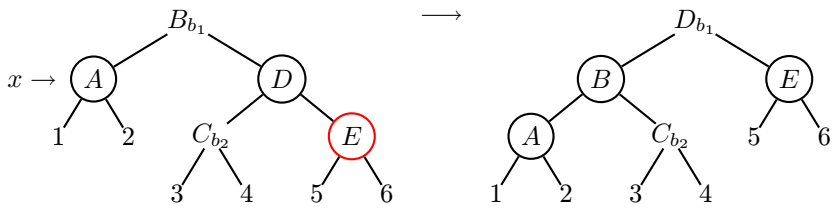
1.



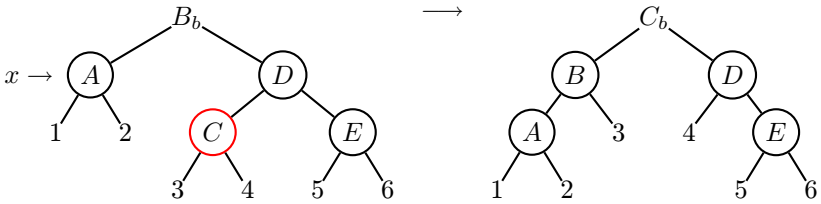
2.



3.



4.



4 AVL stromy

Definice (Adelson-Velskij, Landis) Binární vyhledávací strom je AVL stromem (vyváženým AVL stromem) právě tehdy, když pro každý uzel x ve stromě platí $|h(left(x)) - h(right(x))| \leq 1$ kde $h(x)$ je výška (pod)stromu.
Pro efektivitu operací si pamatujeme explicitně (v položce uzlu) aktuální vyvážení: $\{-1, 0, +1\}$.

Věta Výška AVL stromu s n vrcholy je $O(\log n)$.

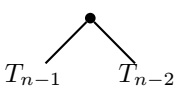
Důkaz (Idea) Konstruujeme nejnevyváženější strom s nejméně vrcholy při dané výšce. Označme p_n počet vrcholů takového stromu T_n s hloubkou n.

$T_0 =$ $p_0 = 1$

$T_1 =$ 

$p_1 = 2$

$\vdots$

$T_n =$ 

$p_n = p_{n-1} + p_{n-2} + 1 \approx Fibonaci_{n+2} - 1$

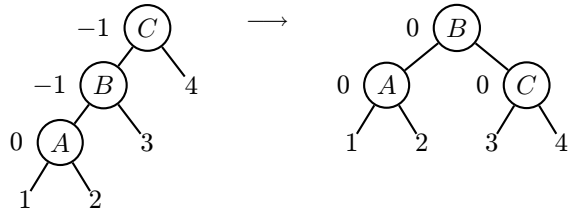
Důsledek Všechny dotazovací operace pro BVS fungují na AVL stromě stejně a mají složitost $O(\log n)$.

4.1 Operace

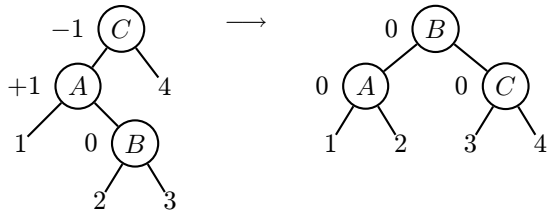
4.1.1 Vkládání

Postupujeme zdola, upravujeme vyvážení (pokud to stačí) a případně použijeme rotaci (jednoduchou, dvojitou). Rotace je časově náročnější než úprava vyvážení.

1. jednoduchá rotace



2. dvojitá rotace



Analogicky symetrické případy.

4.1.2 Vypouštění

Po rotaci musíme změnu propagovat nahoru, protože podstromy jako celek se snížili.

Rotace

1. jednoduchá
2. dvojitá

5 B-stromy

Jsou vyvážené vyhledávací stromy. Jsou s větším a proměnlivým počtem následníků. Vrchol x s $n(x)$ klíči má $n(x) + 1$ dětí.

Aplikace Data na disku. Přístup na disk je časově náročnější. Stránky se načítají celé.

Vlastnosti

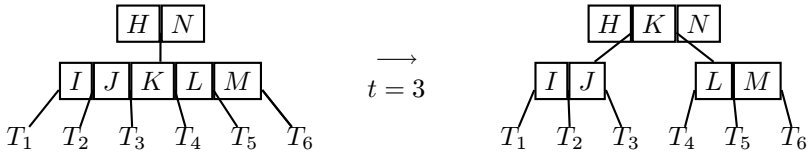
1. klíče jsou uloženy v neklesající posloupnosti
2. pokud je x vnitřní vrchol s $n(x)$ klíči, pak obsahuje $n(x)+1$ pointerů na děti
3. klíče ve vrcholech rozdělují intervaly klíčů v podstromech
4. každý list je ve stejné hloubce
5. pro nějaké pevné t platí, $t \geq 2$ (tzv. minimální stupeň)
   - každý 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.
   - každý vrchol má nejvíc $2t-1$ klíčů, tedy nejvíc $2t$ dětí.

Tvrzení Pro $n \geq 1$, každý B-strom T s n klíči, výškou h a minimalním stupněm $t \geq 2$ platí $h \geq log_t \frac{n+1}{2}$.

Důkaz Pro strom dané výšky h je počet vrcholů minimální, když vrchol obsahuje 1 klíč a ostatní vrcholy $t-1$ klíčů. Pak jsou 2 vrcholy v hloubce 1, $2t$ vrcholů v hlubce 2, $2t^2$ v hloubce 3 ..., $2t^{h-1}$ v hloubce h. Proto počet klíčů splňuje $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h-1}{t-1}\right) = 2t^h - 1$ odkud plyně tvrzení.

5.1 Operace

- *search*
- *create*
- *insert*
- *delete*
- *split*



5.1.1 Vkládání

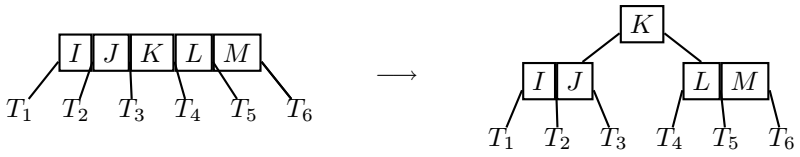
Složitost $O(h)$. Při průchodu dolů dělíme plné vrcholy (variantou je dělení při navracení, je nutné pamatovat si cestu). Speciálním případem je dělení kořene což znamená zvýšení výšky (o 1).

Invariant Vkládáme do ne-plného vrcholu.

5.1.2 Vypouštění

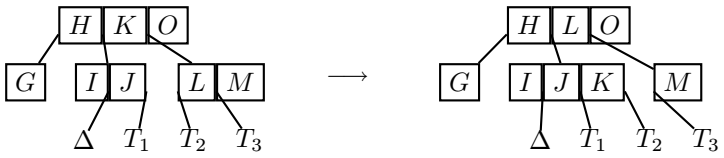
Rekurzivně od kořene procházíme stromem. Speciálním případem je kořen (v neprázdném stromě), pokud kořen nemá žádné klíče a pouze 1 syna, snížíme výšku stromu.

Invariant Počet klíčů ve vrcholu je aspoň t. Zabezpečení invariantu tím, že klíč z aktuálního vrcholu přesuneme do syna.



Rozbor případů

1. vypuštíme klíč k z listu x – přímo
2. vypouštíme klíč k z vnitřního vrcholu x
   (a) 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. 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 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



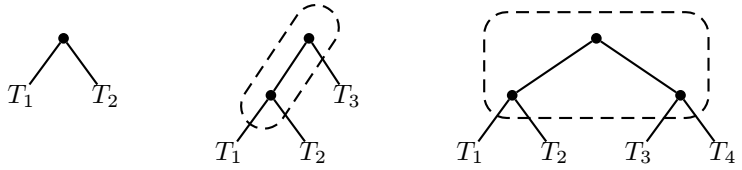
Pozorování

- uspořádání je zachováno $J < T_1 < K < T_2 < L < T_3 < M$
- hloubka se nezmění ani neporuší
- nový vrchol IJK má t klíčů, to jsme chtěli; nový vrchol MN má aspoň $t-1$ klíčů po úpravě

(b) pokud oba sousedé mají $t-1$, slej c_i s jedním ze sousedů, přitom se přesune 1 klíč z vrcholu x do nově vytvářeného vcholu (jako medián)

Složitost vypouštění ze stromu výšky h $O(1)$ diskových přístupů na 1 hladině, $O(h)$ diskových operací celkem.

Souvislosti Pro $t = 2$ má vrchol 2-4 syny, tyto stromy se označují jako tzv. 2-3-4 stromy.



2 3 4

Varianty

- vrchol má aspoň t klíčů a nejvýše $2t$ klíčů
- všechna data v listech
- provázané stromy - pointry na sousedy

6 Intervalové stromy

Rozšíření datové struktury o další informace a operace. Pracujeme s uzavřenými intervaly $[t_1, t_2], t_1 \leq t_2$. Interval $[t_1, t_2]$ je objekt i s položkami $low(i) = t_1$ (dolní konec) a $high(i) = t_2$ (horní konec).

Definice Intervaly i a i' se překrývají pokud $i \cap i' \neq 0$, tj. $low(i) \leq high(i')$ a $low(i') \leq high(i)$.
Platí intervalová trichotomie. Nastává právě jedna z možností

1. i a i' se překrývají
2. $high(i) < low(i')$
3. $high(i') < low(i)$

Intervalový strom je červeno-černý strom, kde každý prvek x obsahuje interval $int(x)$.

Podporované operace

- *insert*
- *delete*
- *search*

Datová struktura Vrchol x obsahuje $int(x)$ a klíč je dolní konec $low(int(x))$.

Dodatečné informace Vrchol obsahuje hodnotu $max(x)$, maximální hodnotu horního konce pro nějaký interval v podstromě x.

Údržba informace při přidávání, vypouštění a rotaci

$$max(x) = max(high(int(x)), max(left(x)), max(right(x)))$$

Vyhledávání i v T Najde jeden interval ve stromě T, který se překrývá s intervalem i, pokud existuje. Idea je v porovnání $max(left(x))$ a $low(i)$

$\geq$ **doleva** vlevo existuje překrývající se interval a nebo vpravo neexistuje

$\leq$ **doprava** vlevo neexistuje překrývající se interval.

Varianty

- vyhledávání v otevřených intervalech
- vyhledávání přesně daného intervalu (obě meze odpovídají) v $O(\log n)$
- vyhledávání překrývajících se intervalů v čase $O(min\{n, k \log n\})$, kde k je počet nalezených intervalů ve výsledném seznamu (težší varianta beze změny stromu)

7 Hašování

Ideově vychází přímo z adresovatelných tabulek, které mají malé univerzum klíčů U a prvky nemají stené klíče. Operace *insert*, *search*, *delete* v čase $O(1)$. Realizace pomocí pole, hodnoty pole obsahují reprezentované prvky (a nebo odkazy na ně) nebo *nil*. Ale přímo adresovatelné tabulky nevyhovují vždy, univerzum klíčů U je velké vzhledem k množině klíču K, které jsou aktuálně uloženy ve struktuře, prvky mají stejné klíče.

Idea Adresu budeme z klíče počítat. Hašovací funkce $h : U \to \{0, 1, \ldots, n-1\}$ mapuje univerzum klíčů U do položek hašovací tabulky $T\{0, \ldots, m-1\}$, redukce paměti. Problémem jsou vznikající kolize, dva klíče se hašují na stejnou hodnotu.

Pozorování Kolizím se nevyhneme, pokud $|U| > m$.

Definice Faktor naplnění $\alpha = \frac{n}{m}$ pro tabulku T velikosti m ve které je uloženo n prvků.

7.1 Volba hašovacích funkcí

Dobrá hašovací funkce splňuje (přibližně) předpoklady jednoduchého uniformního hašování. Pro rozložení pravděpodobnosti P zvolení klíče k a univerza U chceme $\sum_{k:h(k)=j} P(k) = \frac{1}{m}$, pro $j = 0, 1, \ldots, m-1$, ale rozložení pravděpodobností obvykle neznáme. Interpretujeme klíče jako přirozená čísla, aby se daly používat aritmetické funkce.

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

násobení $h(k) = \lfloor m(k \cdot A \mod 1) \rfloor$, obvykle pro $m = 2^p$[1]

univerzální hašování zvolíme hašovací funkci náhodně a nezávisle na klíčích (počas běhu, z vhodné množiny funkcí), použití *randomize*

7.2 Zřetězení prvků

Předpoklady Jednoduché uniformní hašovaní. Každý prvek se hašuje do m položek tabulky se stejnou pravděpodobností, nezávisle na jiných prvcích. Hodnota hašovací funkce $h(k)$ se počítá v čase $O(1)$.

7.2.1 Analýza hašování se zřetězením

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

Důkaz 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 seznamu je α. Proto je očekávaný počet navštívených prvků α a celkový čas (včetně výpočtu hašovací funkce $h(k)$) je $\Theta(1+\alpha)$.

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

[1] Knuth doporučuje $A \approx \frac{(\sqrt{5}-1)}{2} = 0,618033$, tzv. zlatý řez

Důkaz Předpokládáme, že vyhledáváme každý z n uložených klíčů se stejnou pravděpodobností. Předpokládáme, že nové prvky se ukládají na konec seznamu. Očekávaný počet navštívených prvků 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 seznamu je $\frac{i-1}{m}$. Dostaneme $\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) = 1+\frac{1}{nm}\frac{(n-1)n}{2} = 1+\frac{\alpha}{2}-\frac{1}{2m} = \Theta(1+\alpha)$

Závěr Pokud $n = O(m)$, pak $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$. Po vyhledání, vlastní operace pro přidání, resp. vypuštění prvku v čase $O(1)$.

7.3 Otevřené adresování

Všechny prvky jsou uloženy v tabulce $\alpha < 1$. Pro řešení kolizí nepotřebujeme pointery, ale počítáme adresy navštívených prvků, ve stejné paměti tedy máme větší tabulku než při zřetězení.
Posloupnost zkoušených pozic závisí na klíči a pořadí pokusu $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$. Prohledáváme pozice v posloupnosti $\langle h(x,0), h(x,1), \ldots, h(x,m-1)\rangle$.

Operace

- *search*
- *insert*
- *delete*

Předpoklad uniformního hašování Každá permutace pozicí $\{0, \ldots, m-1\}$ je pro každý klíč vybrána stejně pravděpodobně jako posloupnost zkoušených pozicí. Je těžké implementovat, používají se metody, které ho nesplňují.

7.3.1 Metody

Lineární zkoušení

$$h' : U \to \{0, \ldots, m-1\},\ h(x,i) = (h'(x) + i) \mod m$$

Pouze m ruzných posloupností zkoušených pozicí. Problém primárního klastrování – vznikají dlouhé úseky obsazených pozic. Pravděpodobné obsazení 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 $\frac{1}{m}$.
Lze implementovat *delete* tak, že následující prvky posuneme.

Kvadratické zkoušení

$$h(x,i) = (h'(x) + c_1 i + c_2 i^2) \mod m,\ c_1 \neq 0, c_2 \neq 0$$

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 pozicí, problémem je druhotné klastrování. Pouze m ruzných posloupností.

Dvojité hašování

$$h(x,i) = (h_1(x) + i \cdot h_2(x)) \mod m,\ h_1, h_2 \text{ jsou pomocné hašovací funkce}$$

$h_2(x)$ nesoudělné s m, aby se prošla celá tabulka. Obvykle $m = 2^p$ a $h_2(x)$ je liché, nebo m prvočíslo a $0 \leq h_2(x) < m$.

7.3.2 Analýza hašování s otevřeným adresováním

Předpokladem je uniformní hašovaní. Pro každý klíč x je posloupnost zkoušených pozic libovolná permutace $\{0, \ldots, m-1\}$ se stejnou pravděpodobností.

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

Důkaz Všechny zkoušky pozice kromě poslední našly obsazenou pozici. Definujeme $p_i = P(\text{právě } i \text{ zkoušek našlo obsazenou pozici})$. Očekávaný počet zkoušek je $1 + \sum_{i=0}^{\infty} i \cdot p_i$. Definujeme $q_i = P(\text{aspoň } i \text{ zkoušek našlo obsazenou pozici})$. Platí $\sum_{i=0}^{\infty} i \cdot p_i = \sum_{i=0}^{\infty} q_i$. První zkouška narazí na obsazenou pozici s pravděpodobností $\frac{n}{m} = q_1$. Obecně $q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\ldots\left(\frac{n-i+i}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$. Spočteme $1 + \sum_{i=0}^{\infty} i \cdot p_i = 1 + \sum_{i=0}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \frac{1}{1-\alpha}$.

Důsledek Vkládání prvku vyžaduje nejvýš $\frac{1}{1-\alpha}$ zkoušek, za stejných předpokladů.

7.4 Univerzální hašování

Idea Zvolíme hašovací funkci náhodně a nezávisle na klíče (počas běhu, z vhodné množiny funkcí), použití *randomize*.

Nechť χ je konečná množina hašovacích funkcí z univerza klíčů U do $\{0, \ldots, m-1\}$. Množinu χ nazveme *univerzální*, pokud pro každé dva různé klíče $x, y \in U$ je počet hašovacích funkcí $h \in \chi$, pro které $h(x) = h(y)$ roven $\frac{|\chi|}{m}$, tj. pro náhodně zvolenou funkci h je pravděpodobnost kolize pro x a y, kde $x \neq y$, právě $\frac{1}{m}$. To je stejná pravděpodobnost, jako když jsou hodnoty $h(x)$ a $h(y)$ zvoleny náhodně z množiny $\{0, \ldots, m-1\}$.

Implementace Hašovací funkce závisí na parametrech, které se zvolí za běhu. $\chi = \{h_a(x) : U \to \{0, \ldots, m-1\} | a \in A,\ h_a(x) = h(a, x)$ kde a je parametr$\}$

Důsledky *randomize*

- žádný konkrétní vstup (konkrétních n klíčů) není a priori špatný
- opakované použití na stejný vstup volá (skoro jistě) ruzné hašovací funkce, průměrný počet případů nastane pro libovolný počet vstupních dat

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 učástní náhodně vybraný konkrétní klíč x je menší než 1.

Důkaz Pro každý pár ruzných 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 $\frac{1}{m}$, proto očekávaná hodnota $E(c_{yz}) = \frac{1}{m}$. 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 $E(C_x) < 1$.

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

7.4.1 Konstrukce univerzální množiny hašovacích funkcí

Zvolíme prvočíslo m jako velikost tabulky. Každý klíč x rozdělíme na $(r+1)$ částí (např. znaků, hodnota r závisí na velikosti klíčů). Píšeme $x = \langle x_0, x_1, \ldots, x_r \rangle$. Zvolené r splňuje podmínku, že každá část x_i je (ostře) menší než m.
Zvolme $a = \langle a_0, a_1, \ldots, a_r \rangle$ posloupnost $(r+1)$ čísel náhodně a nezávisle vybraných z množiny $\{0, 1, \ldots, m-1\}$.

Definujeme $h_a \in \chi : h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m$, $\chi = \cup_a \{h_a\}$. Platí $|\chi| = m^{r+1}$, tj. počet ruzných vektorů a.

Věta χ je univerzální množina hašovacích funkcí.

Důkaz Uvažujeme různé klíče x a y. Bez újmy na obecnosti $x_0 \neq y_0$. Pro pevné hodnoty $a_1, a_2, \ldots, a_r$ je právě jedna hodnota a_0, která splňuje $h(x) = h(y)$. Je to řešení rovnice $a_0(x_0 - y_0) \equiv -\sum_{i=0}^{r} a_i(x_i - y_i) \mod m$. Protože m je prvočíslo, nenulová hodnota $x_0 - y_0$ má jediný inverzní prvek modulo m a tedy existuje jediné řešení pro $a_0 \mod m$. Následně, každý pár klíčů x, y koliduje pro právě m^r hodnot vektoru a, protože kolidují právě jednou pro každou volbu $\langle a_1, a_2, \ldots, a_r \rangle$, tj. pro jedno a_0. Protože je m^{r+1} možností pro a, klíče kolidují s pravděpodobností $\frac{m^r}{m^{r+1}} = \frac{1}{m}$. Tedy χ je univerzální.

7.5 Hašování a rostoucí tabulky

Dynamizace hašovací tabulky. Chceme odstranit nevýhody hašování při zachování asymptotické ceny operací (tj. $O(1)$ pro pevné α)

- pevná velikost tabulky
- nedokonalá implementace *delete* (u některých metod).

Idea Periodická reorganizace datové struktury. Při růstu po dosažení naplnění α zvětšíme tabulku α-krát (např. $\alpha = 2$, z $m = 2^i$ na $m = 2^{i+1}$) a prvky přehašujeme (s novou hašovací funkcí). Dále pro $\alpha = 2$ aspoň $2^{i-1} \cdot \alpha$ operací, *insert* zaplatí $1\times$ při přidání do staré struktury, $2\times$ přehašování $2 \cdot 2^{i-1} \cdot \alpha$ prvků (kredit operace je obecně $\leq \frac{2\alpha-1}{\alpha-1}$).

Poznámka Vhodné α lze spočítat (z pohledu efektivity budoucích operací s a *bez* přehašování).

Obecné řešení při růstu i zmenšování tabulky Při přehašování je naplnění tabulky $\frac{1}{4}\alpha$ až $\frac{1}{2}\alpha$, $\overline{\alpha} = \alpha \cdot m$

- tabulku zvětšíme, pokud máme $\overline{\alpha}$ prvků (aspoň $\frac{1}{2}\overline{\alpha}$ operací)
- tabulku zmenšíme, pokud máme $\frac{1}{8}\overline{\alpha}$ prvků (aspoň $\frac{1}{8}\overline{\alpha}$ operací).

Poznámka Skutečný počet prvků si pamatujeme; pseudovolná místa uvolníme, vznik pseudovolného místa potřebuje (aspoň) 2 operace.

Existují i jiné metody, např. dynamizace datové struktury.

8 Haldy

Datová struktura pro implementaci ADT prioritní fronty (abstraktní datový typ).

Základní operace pro prioritní frontu

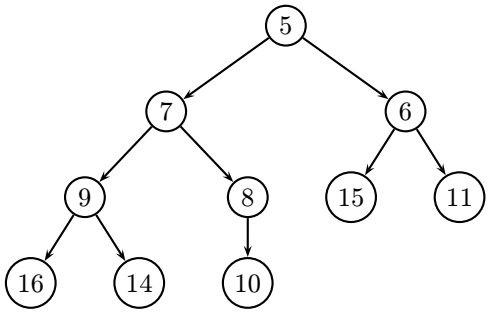
- *create*
- $insert(M, x)$
- $min(M)$
- $deletemin(M)$

Další operace pro haldu

- $lowerkey(M, x, k)$
- $delete(M, x)$
- $unificate(M1, M2)$

Známá implementace

binární halda



pole vrchol i má syny na adrese $2i$ a $2i + 1$

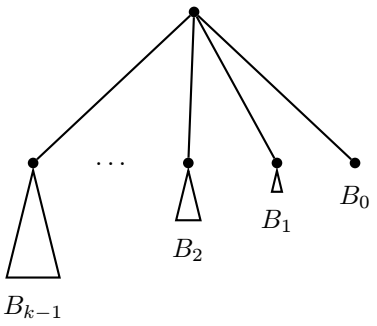
5	7	6	9	8	15	11	16	14	10

Časová složitost operací Naším cílem je aby operace byly logaritmické, ale halda nemusí mít tvar binárního stromu. Čím je halda užší, tím jednodušší je nalezení minima.

operace	binární halda	binomiální halda	Fibonacciho halda
insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
deletemin	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
$unificate$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
lowerkey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

8.1 Binomické stromy

Binomický strom B_k řádu k, $k \geq 0$.



B_k

Věta Binomický strom B_k

1. obsahuje právě 2^k vrcholů
2. má výšku k
3. má právě $\binom{k}{i}$ vrcholů v hloubce i, pro $i = 0, \ldots, k$
4. kořen má k synů, přičemž i-tý syn zprava (pro $i = 0, \ldots, k-1$) je kořenem podstromu B_i

Důkaz (indukcí)

1. $B_k = 2 \cdot B_{k-1} = 2 \cdot 2^{k-1} = 2^k$
2. $h(B_k) = h(B_{k-1}) + 1 = (k-1) + 1 = k$
3. $\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$

4. Pro kořen B_k platí že nejlevější syn je B_{k-1}, ostatní z indukčního předpokladu; má k synů

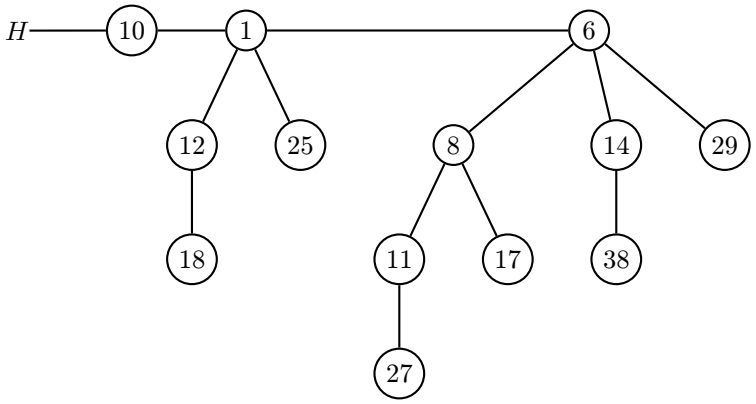
Důsledek Maximální stupeň vrcholu v binomiálním stromě s n vrcholy je $\log n$

8.2 Binomická halda

Spojový seznam binomických stromů s ohodnocenými vrcholy, které splňují:

1. pokud x je otcem y, potom $key(x) \leq key(y)$, strom je haldově uspořádán
2. pro každé $k \geq 0$ se strom B_k vyskytuje v haldě nejvýše jedenkrát
3. stromy jsou uspořádány podle řádu (v rostoucím pořadí)

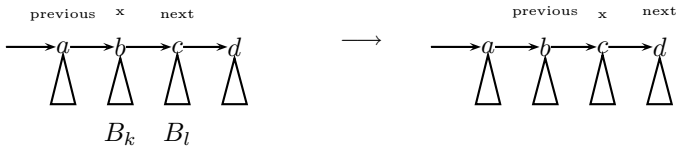
Příklad



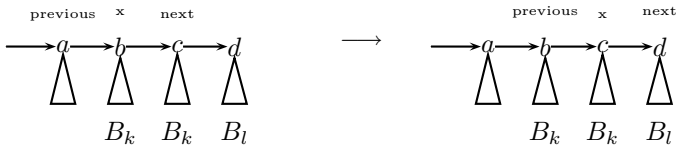
8.2.1 Sjednocení

`function` $unificate(H_1, H_2)$
$H := create()$
$head(H) := merge(H_1, H_2)$
`if` $head(H) = nil$ `then return` H
`else` previous := nil;$x := head(H)$; next := $brother(H)$
`while` next $\neq nil$ `do`

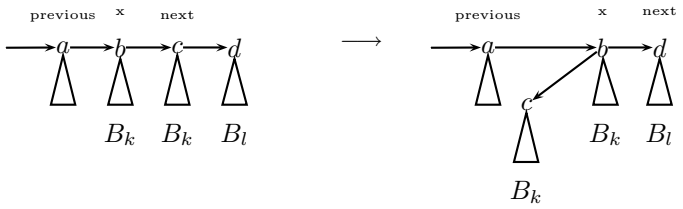
1. $k < l$



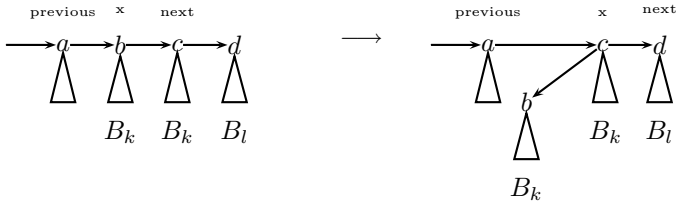
2. $k = l$



3. $k < l,\ key(x) \leq key(next)$



4. $k < l,\ key(x) > key(next)$



8.2.2 Další operace

Všechny tyto operace mají logaritmickou složitost.

`function` $insert(H, x)$
vytvoř haldu H' obsahující jediný prvek x
$H := unificate(H, H')$

`function` $deletemin(H)$
ve spojovém seznamu H najdi vrchol x s minimálním klíčem a vyjmi ho ze seznamu
vytvoř prázdnou haldu H'
ulož syny vrcholu x do seznamu v obráceném pořadí a hlavu seznamu ulož do $head(H')$
$H := unificate(H, H')$
`return` x

`function` $min(H)$
prohledej spojový seznam kořenů binomických stromů, na který ukazuje $head(H)$

`return` prvek s minimální hodnotou

`function` $lowerkey(H, x, k)$
`if` $k > key(x)$ `then error`
`else` $key(x) := k$
porovnávej $key(x)$ s klíčem otce x a probublávej směrem ke kořeni stromu

`function` $delete(H, x)$
$lowerkey(H, x, -\infty)$
$deletemin(H)$

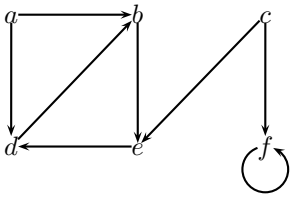
8.2.3 Implementace

Zde zbrklá (eager, dychtivá) kdy strukturu reorganizujeme hned po její změně, versus líná (lazy) kdy strukturu reorganizujeme až je to skutečně potřeba.

9 Grafové algoritmy

9.1 Reprezentace grafu

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



$$G = (V, E)$$
$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, b), (a, d), (b, e), (c, e), (c, f), (d, b), (e, d), (f, f)\}$$

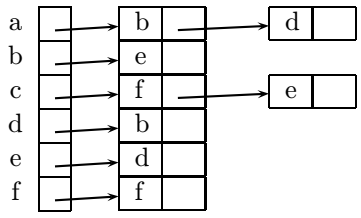
Matice sousednosti

$A = (a_{ij})$ typu $|V| \times |V|$

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

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Seznamy sousedů Pole $neighbours$ velikosti $|V|$, pro $u \in V$ je $neighbours(u)$ hlavou seznamu obsahujícího vrcholy v, pro které platí $(u, v) \in E$. Spotřeba paměti je $O(|V| + |E|) = O(max(|V|, |E|))$.



Varianty

- neorientovaný graf
- (hranově) ohodnocený graf
- seznamy sousedů generované dynamicky

9.2 Prohledávání grafů

9.2.1 Prohledávání do hloubky (Depth First Search)

Vstup Graf $G = (V, E)$, zadaný pomocí seznamu sousedů.

Pomocné datové struktury

$\pi(v)$ otec vrcholu v ve stromu prohledávání

$o(v)$ pořadí, v němž jsou vrcholy $v \in V$ navštíveny

order globální proměnná, slouží k číslování vrcholů

Algoritmus

```
function dfs(G)
forall u ∈ V do
  o(u) := 0
  π(u) = nil
od
order := 0
forall u ∈ V do
 if o(u) = 0 then
  visit(u)
 fi
od

function visit(u)
order++
o(u) := order
forall v ∈ neighbours(u) do
 if o(v) = 0 then
  π(v) = u
  visit(u)
 fi
od
```

Časová složitost $\Theta(|V| + |E|)$

Graf průchodu do hloubky (DFS-les) $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi(v), v) | v \in V \text{ a } \pi(v) \neq nil\}$

Klasifikace hran grafu G

stromová hrana vede do nového vrcholu

zpětná hrana vede do už navštíveného vrcholu (neuzavřeného)

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

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

uzavřený vrchol všechny hrany vedoucí z něho jsme prohledali

Aplikace DFS

- komponenty grafu
- existence kružnic

9.2.2 Prohledávání do šířky (Breadth First Search)

Vstup Graf $G = (V, E)$, zadaný pomocí seznamu 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 $v \in V$

Q fronta

Algoritmus

```
function bfs(G)
forall u ∈ V \ {s} do
  d(u) := ∞
  π(u) = nil
od
d(s) := 0
π(s) := nil
Q := {s}
while Q ≠ ∅ do
  u := deletemin(Q)
  forall v ∈ neighbours(u) do
    if d(v) = ∞ then
      d(v) := d(u) + 1
      π(v) := u
```

$Q := Q \cup \{v\}$
fi
od
od

Časová složitost $O(|V| + |E|)$

Graf průchodu do hloubky (BFS-strom) $G_\pi = (V, E_\pi)$, $V = \{v \in V | \pi(v) \neq nil\} \cup \{s\}$, $E_\pi = \{(\pi(v), v) \in E | v \in V \setminus \{s\}\}$

Aplikace

- $d(v)$ je nejkratší cesta z s do v
- rekonstrukce cesty pomocí π

9.3 Topologické uspořádání

$v_1, v_2, \ldots, v_n$ je topologické uspořádání vrcholů orientovaného grafu G, pokud pro každou hranu $(v_i, v_j) \in E$ platí $i < j$.
Graf G lze topologicky uspořádat $\Leftrightarrow$ G je acyklický $\Leftrightarrow$ DFS nenajde zpětnou hranu. Tyto grafy jsou nazývány DAG (direct acyclic graph).

Pojmy

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

Algoritmus
`function` *topologic*(G)
volej $dfs(G)$ a spočítej časy uzavření vrcholů
`if` existuje zpětná hrana `then`
`return` "G není acyklický"
každý vrchol který je uzavřen ulož na začátek spojového seznamu S
`return` S

Časová složitost Celková složitost je $\Theta(|V| + |E|)$. Prohledávání do hloubky má složitost $\Theta(|V| + |E|)$, vkládání vrcholů do seznamu má složitost $|V| \cdot O(1)$.

9.4 Silně souvislé komponenty

Definice 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.

Pojmy

- 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
`function` $ssk(G)$
algoritmem $dfs(G)$ urči časy uzavření $k(v)$ pro $\forall v \in V$
vytvoř G^T
uspořádej vrcholy do klesající posloupnosti podle $k(v)$ a v tomto pořadí je zpracuj algoritmem $dfs(G^T)$
silně souvislé komponenty grafu G jsou právě podgrafy indukované vrcholovými množinami jednotlivých DFS-stromů z minulého kroku

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 $o(U) = min\{o(u)|u \in U\}$ což je čas prvního navštívení vrcholu u (otevření) a $k(U) = max\{k(u)|u \in U\}$ což 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.

Důkaz Indukcí podle počtu projitých stromů. Z jednoho vrcholu komponenty projdu celou komponentu v G^T i v G. Pokud se dostanu hranou mimo komponentu (do vrcholu v), je v už uzavřený.

9.5 Problém nejkratší cesty

Pojmy

- orientovaný graf $G = (V, E)$
- ohodnocení hran $w : E \to \mathbb{R}$
- cena orientované cesty $P = v_0, v_1, \ldots v_k$

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

- cena nejkratší cesty z u do v

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

- 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 cesta neexistuje, $\infty + r = \infty, \infty > r$ pro $\forall r \in \mathbb{R}$

Varianty problému Najít nejkratší cestu

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

`function` $releaseedge(u,v)$
`if` $d(v) > d(u) + w(u,v)$ then
  $d(v) := d(u) + w(u,v)$
  $\pi(v) := u$
`fi`

Rekonstrukce cesty pomocí předchůdců π.

9.5.1 Dijkstrův algoritmus

Vstup Orientovaný graf $G = (V,E)$, nezáporné ohodnocení hran $w : E \to \mathbb{R}$, počáteční vrchol $s \in V$.

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

Algoritmus
`function` $dijkstra(G)$
`forall` $v \in V$ `do`
  $d(v)$:=∞
  $\pi(v)$:=nil
`od`
$d(s) := 0$
$D := 0$
$Q := V$
`while` $Q \neq \emptyset$ `do`
 u := $deletemin(Q)$
 D := D$\cup\{u\}$
 `forall` $v \in neighbours(u), v \in Q$ `do`

```
  releaseedge(u, v)
 od
od
```

Časová složitost $n = |V|$, $m = |E|$, operace *deletemin* se vykonává $n\times$, operace *lowerkey* se vykonává $m\times \Rightarrow T(disjktra) = n \cdot T(deletemin) + m \cdot T(lowerkey)$.

struktura	$T(deletemin)$	$T(lowerkey)$	$T(dijsktra)$
pole	$O(n)$	$O(1)$	$O(n^2)$
binární halda	$O(\log n)$	$O(\log n)$	$O(m \cdot \log n)$
Fibonacciho halda	$O(\log n)$	$O(1)$	$O(n \cdot \log n + m)$

Lemma 1 (optimální podstruktura) Je-li $v_1, \ldots, v_k$ nejkratší cesta z v_1 do v_k, potom $v_i, \ldots, v_j$ je nejkratší cesta z v_i do v_j pro $\forall i, j, 1 \leq i < j \leq k$.

Lemma 2 (Δ nerovnost) $\delta(s, v) \leq \delta(s, u) + w(u, v)$ pro každou hranu u, v.
Po provedení inicializace je ohodnocení vrcholů měněno prostřednictvím *releaseedge*.

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 $v_0, \ldots, v_k$ nejkratší cesta z $s = v_0$ do v_k, potom po uvolnění hran v pořadí $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ je $d(v) = \delta(s, v_k)$.

Věta Pokud Dijkstrův algoritmus provedeme na orientovaném ohodnoceném grafu s nezáporným ohodnocením hran, s počátečním vrcholem s, pak po ukončení algoritmu platí $d(v) = \delta(s, v)$ pro všechny vrcholy $v \in V$.

Důkaz (Idea) $d(y) = \delta(s, y)$ pro vrchol y vkládaný do D.

9.5.2 Bellman-Fordův algoritmus

Vstup Orientovaný graf $G = (V, E)$, ohodnocení hran $w : E \to \mathbb{R}$, 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
`function` $bf(G)$
$initialization(G, s)$
`for` $i := 1$ to $|V| - 1$ `do`
 `forall` $(u, v) \in E$ `do`
  $releaseedge(u, v)$
 `od`
`od`
`forall` $(u, v) \in E$ `do`
 `if` $d(v) > d(u) + w(u, v)$ `then return` "NE" `fi`
`od`
`return` "ANO"

Časová složitost $O(|V| \cdot |E|)$

Lemma Pro graf $G = (V, E)$ s počtem vrcholů s, cenou $w : E \to \mathbb{R}$, ve kterém není záporný cyklus dosažitelný z s, skončí algoritmus Bellman-Ford tak, že platí $d(v) = \delta(s, v)$ pro všechny vrcholy v dosažitelné z s.

Důkaz(Idea) Indukcí podle počtu vnějších cyklů. Po i-tém cyklu jsou minimální cesty délky i správně spočítány.

Záporné cykly se projeví jako záporné číslo na diagonále.

9.5.3 Floyd-Warshallův algoritmus

Řeší problém nalézt $\delta(u, v)$ pro každé $u, v \in V$.

Idea $\delta_k(i, j)$ = délka nejkratší cesty z i do j, jejíž všechny vnitřní vrcholy jsou v množině $\{1, 2, \ldots, k\}$.

$$\delta_k(i,j) = \begin{cases} w(i,j) & \text{pro } k = 0 \\ min\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\} & \text{pro } k > 0 \end{cases}$$

Vstup Orientovaný graf $G = (V, E)$, nezáporné ohodnocení hran $w : E \to \mathbb{R}_0^+$ (v matici).

Výstup Matice $D_{i,j} = \delta(i, j)$, $\Pi_{i,j}$ = předchůdce vrcholu j na nejkratší cestě z i do j.

Algoritmus
`function` $fw(G)$
`for` $i := 1$ `to` n `do`
 `for` $j := 1$ `to` n `do`
  $\Pi_{i,j} := nil$
  `if` $i = j$ `then` $D_{i,j} := 0$

```
  else if (i,j) ∉ E then D_{i,j} := ∞
  else D_{i,j} := w(i,j); Π_{i,j} := i
  fi
 od
od
for k := 1 to n do
 for i := 1 to n do
  for j := 1 to n do
    if D_{i,k} + D_{k,j} < D_{i,j} then
       D_{i,j} := D_{i,k} + D_{k,j}
       Π_{i,j} := Π_{k,j}
    fi
  od
 od
od
```

Časová složitost $O(n^3)$, paměťová $O(n^2)$

9.6 Algoritmy násobení matic

Postupujeme indukcí podle počtu hran na nejkratší cestě. Definujeme d_{ij}^k =minimální cena cesty z i do j s nejvýše k hranami, kde pro hodnotu k platí

$$k = 1 : \begin{cases} d_{ij}^k = 0 & i = j \\ d_{ij}^k = w(i,j) & \text{pokud hrana } (i,j) \text{ existuje} \\ d_{ij}^k = \infty & \text{jinak} \end{cases}$$

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

Hodnoty d_{ij}^k jsou v matici $D^{(k)}$, hodnoty $w(i,j)$ v matici W. Potom $D^{(k+1)} = D^{(k)} \otimes W$, kde pro maticové násobení $\otimes$ používáme skalární součin, v němž je

- násobení nahrazeno sčítáním
- sčítání nahrazeno minimem.

Pokud v G nejsou záporné cykly, potom je každá nejkratší cesta jednoduchá (tj. bez cyklů), tedy každá nejkratší cesta má nejvýše $n-1$ hran, $D^{(n-1)} = D^{(n)} = D^{(n+1)} = \ldots = D$.

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

Rychlá verze algoritmu Využijeme asociativitu operace $\otimes$ a počítáme pouze mocniny, $\lceil \log n \rceil$ násobení, složitost $\lceil \log n \rceil \cdot \Theta(n^3) = \Theta(n^3 \log n)$.

Poznámka Pro $\otimes$ nelze použít rychlé násobení matic (Strassenův algoritmus).

Algoritmy lze adaptovat na tranzitivní uzávěr grafu. Pro $G = (V, E)$, je $G^* = (V, E^*)$ tranzitivní uzávěr G, kde $E^* = \{(i, j)|\text{existuje cesta z } i \text{ do } j \text{ v } G\}$. Konstruovaná matice dosažitelnosti obsahuje boolovské hodnoty (nebo 0-1) a používají se boolovské operace.

Algoritmy pro všechny cesty lze získat n-násobným spuštěním (pro $\forall$ vrcholy) algoritmů pro nejkratší cesty z 1 zdroje (Dijkstra, kritická cesta).

9.7 Extremální cesty v acyklickém grafu

Z jednoho vrcholu s nalézt cesty do ostatních, tedy úloha 2. Nejkratší cesta je vždy dobře definovaná, protože i když se vyskytují záporné hrany, neexistují záporné cykly.

Idea Využijeme topologické uspořádání vrcholů

9.7.1 Nejkratší cesta v acyklickém grafu

Vstup Acyklický orientovaný graf $G = (V, E)$, $w : E \to \mathbb{R}$, $s \in V$.

Výstup $d(v) = \delta(s, v)$, $\pi(v)$ pro $\forall v \in V$.

Algoritmus
topologicky uspořádat vrcholy G
$initialize(G, s)$
`for` každý vrchol u v pořadí topologického uspořádání `do`
 `forall` $v \in neighbours(u)$ `do`
  $releaseedge(u, v)$
 `od`
`od`

Časová složitost $\Theta(|V| + |E|)$, protože

- topologické uspořádání – $\Theta(|V| + |E|)$
- zpracování vrcholů v cyklu – každý jednou v $O(1)$
- zpracování hran ve vnitřním cyklu – každá jednou, při zpracování počátečního vrcholu hrany, vlastní zpracování $O(1)$ (na rozdíl od Dijsktrova algoritmu)

9.8 PERT-kritické cesty

"Program evaluation and review technique".

Problém Je dána množina úloh a délka vykonávání každé z nich. Některé dvojice úloh mohou na sobě záviset, tzn. jedna úloha musí skončit dřív než druhá začne. Cílem je získat 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) dokončena před vykonáním 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 vykonání uspořádané posloupnosti úloh. Cena kritické cesty je dolní odhad pro celkový čas vykonání všech úloh.

Algoritmus nalezení kritické cesty

1. znegování cen hran a hledání nejkratší cesty
2. hledáním nejdelší cesty v DAG (změna ∞ na $-\infty$ v inicializaci a $>$ na $<$ v uvolnění)

Proč je bod 2 v DAG možný? Platí princip optimality – libovolná část nejdelší cesty je opět nejdelší.

Implementace Nulové hrany pro závislosti. Přirozenější reprezentací je že úlohy jsou vrcholy, hrany odpovídají závislostem – hrana (u, v) znamená, že se u vykonává před v.

9.9 Minimální kostra grafu

Vstup Souvislý graf $G = (V, E)$ s hranovým ohodnocením $w : E \to \mathbb{R}$.

Kostra Podgraf T splňující $V(T) = V$, který je stromem.

Minimální kostra Minimalizuje $w(T) = \sum_{e \in E(T)} w(e)$.

9.9.1 Kruskalův algoritmus

uspořádej hrany z E tak, aby $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$
$E(T) := 0$
$i := 1$
`while` $|E(T)| < |V| - 1$ `do`
 `if` $E(T) \cup \{e_i\}$ neobsahuje kružnici `then`
  přidej e_i do $E(T)$

```
 fi
 i++
od
```

Časová složitost Třídění hran má složitost $\Theta(|E| \cdot \log |E|)$ ($=O(|E| \cdot \log |V|)$), zpracování hrany při vhodné reprezentaci má složitost $< \Theta(\log |E|)$, tzv. union-find (faktorová množina komponent).

Datová struktura Ke každé komponentě si pamatujeme reprezentanta (vrchol) v poli velikosti $|V|$ anebo pomocí pointerů.

Pomocné operace
`function` $unificate(u, v)$
$r_u := find(u)$
$r_v := find(v)$
$representative(r_u) := r_v$ (1)

`function` $find(u)$
projde reprezentany od u

Při testování hrany zjišťujeme, zda reprezentanti koncových vrcholů jsou stejné (konce jsou ve stejné komponentě). Složitost operace je rovna hloubce stromu reprezentantů. Pokud v (1) připojujeme menší komponentu k větší, dostaneme hloubku $O(\log n)$, kde n je velikost komponenty.

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

Definice Rozklad množiny vrcholů na dvě části $(S, V \setminus S)$ se nazývá *řez*. Hranu $(u, v) \in E$ 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 nejmenší ze všech hran křížících řez.

Definice Nechť 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 Nechť $G = (V, E)$ je souvislý, neorientovaný graf s váhovou funkcí $w : E \to \mathbb{R}$, nechť $A \subseteq E$ je podmnožinou nějaké minimální kostry a nechť $(S, V \setminus S)$ 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.

Důkaz Nechť T je minimální kostra, která obsahuje A a nebsahuje lehkou hranu (u, v). Zkonstruujeme T' – minimální kostru, 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). $(x, y) \notin A$, protože řez respektuje A. Odstranění (x, y) z T ji rozdělí na dvě komponenty a přidání (u, v) 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 minimální kostra. Protože $A \cup \{(u, v)\} \subseteq T'$, je (u, v) bezpečná pro A.

Důsledek Nechť $G = (V, E)$ je souvislý orientovaný graf s váhovou funkcí $w : E \to \mathbb{R}$, nechť $A \subseteq E$ je podmnožinou nějaké minimální kostry a nechť C je souvislá komponenta (strom) podgrafu zadaného množinou A. Pokud je $(u, v) \in E$ hrana s minimální vahou spojující C s jinými komponentami grafu E_A zadaného množinou A, potom (u, v) je bezpečná pro A.

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

9.9.2 Jarníkův, Primův algoritmus

```
forall v ∈ V do
  key(v):=∞
od
key(s) := 0
π(s) := nil
Q := V
while Q ≠ ∅ do
  u := deletemin(Q)
  forall v ∈ neighbours(u), v ∈ Q do
    if w(u, v) < key(v) then
      π(v) := u
      key(v) := w(u, v)^(1)
```

Časová složitost $O(|E| \cdot \log |V|)$ pomocí binární haldy. Postavení haldy má složitost $O(|V|)$, operace *deletemin* má složitost $|V| \cdot O(\log |V|)$, operace $lowerkey^{(1)}$ má složitost $|E| \cdot O(\log |V|)$. Při použití Fibonacciho haldy má operace $lowerkey^{(1)}$ složitost $|E| \cdot O(1)$ (amortizovanou), celkem tedy $O(|E| + |V| \log |V|)$. Při reprezentaci v poli je složitost $\Theta(|V|^2)$.

10 Hladový algoritmus

Motivační příklad Obnos o nominální hodnotě u rozměnit na minimální počet mincí o denominacích 1, 5, 10, 25.

Přímočaré řešení Vyzkoušet všechny možnosti.

Efektivní algoritmus V každém kroku zvolíme minci. O maximální denominaci, jejíž hodnota $\leq$ než obnos, který zbývá rozměnit.

Dostaneme vždy optimální řešení? Při variantě denominace 1, 3, 4, 5, nejde obnos 7 měnit hladově.

Obecný popis hladových algoritmů Řešíme optimalizační problém ve kterém hledáme maximum/minimum nějaké veličiny. Typickou strategií je, že řešení budujeme v posloupnosti kroků. V každém stavu je konečně mnoho pokračování. Hladový algoritmus volí tu možnost, která se v daném stavu jeví jako nejlepší. Volba lokálních optim nás přivede ke globálnímu optimu.

10.1 Hladový algoritmus pro plánování úloh

Vstup Množina úloh $S = \{1, \ldots, n\}$, úloha i probíhá v čase $\langle z_i, k_i)$.

Výstup $M \subseteq S$, s maximální hodnotou $|M|$ splňující $i, j \in M$, $i \neq j \Rightarrow \langle z_i, k_i) \cap \langle z_j, k_j) = \emptyset$ (tj. úlohy se nepřekrývají).

Algoritmus
`function` *plan*
uspořádej prvky S tak, aby $k_1 \leq k_2 \leq \ldots \leq k_n$
$M := \{1\}$
$j := 1$
`for` $i := 2$ to n `do`
 `if` $z_i \geq k_j$ `then`
  $M := M \cap \{i\}$
  $j := i$
 `fi`
`od`
`return` M

Důkaz korektnosti V každém kroku hladového algoritmu existuje optimální řešení M^* takové, že $M \subseteq M^*$.

$M :=$ $\{1\}$ Buď M^* libovolné optimální řešení. Buď $u \in M^*$ úloha s minimálním k_u. Pak $M^* \setminus \{u\} \cup \{1\}$ je taky optimální řešení ($k_1 \leq k_u \Rightarrow$ úlohy se nepřekrývají).

$M :=$ $\{1, \ldots, z+1\}$ Předpokládejme, že $\exists$ optimální řešení $M^* \supseteq \{1, \ldots, i\}$. Pak $M^* \setminus \{1, \ldots, i\}$ je optimální řešení problému plánování pro množinu úloh $S' = \{u \in S | z_n \geq k_i\}$.

Varování Modifikace plánování úloh, při které se vybírá úloha nepřekrývající se s již vybranými a trvající nejkratší dobu nedává správné výsledky, tedy ne každý hladový výběr je správný.

10.2 Charakterizace problémů které lze řešit hladovým algoritmem

Obecný popis není znám.

10.2.1 Pravidlo hladového výběru

Ke globálně optimálnímu řešení se dostaneme lokálně optimálním (hladovým) krokem (na rozdíl od dynamického programování nebere do úvahy řešení podproblémů).
Buď S množina úloh, u úloha, která skončí nejdříve. Pak $\exists$ optimální řešení $M \subseteq S$ takové, že $u \in M$.

Optimální podstruktura

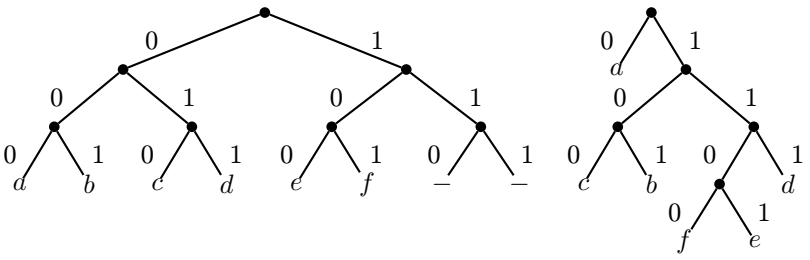
Optimální řešení problému obsahuje optimální řešení podproblémů.
Buď u úloha v optimálním řešení M, která skončí nejdříve. Pak $M \setminus \{u\}$ je optimální řešení pro $S' = \{i \in S | z_i \geq k_u\}$.

10.3 Huffmanův kód

Autorem Hufmanova kódu je David Hufman (1951). Huffmanův kód je optimální prefixoý kód.

Aplikace Komprese dat, tedy jejich kódování s cílem zmenšit jejich objem. Data musí být možno obnovit dekódovacím algoritmem. V případě bezztrátové komprese musí být obnovená data identická.
Kód pro vstupní abecedu A_1 a výstupní abecedu A_2 je funkce $f : A_1 \to A_2^*$.

Prefixový kód Žádné kódované slovo není předponou jiného kódového slova (lze ho tedy jednoznačně dekódovat). Každý prefixový kód lze znázornit binárním stromem (tzv. prefixovým stromem).



Situace Je dána abeceda A a slovo s nad touto abecedou. Pro každý znak $z \in A$ známe jeho četnost $f(z) =$ počet výskytů znaku z v slově s.

Pozorování V optimálním prefixovém kódu musí mít znaky s větší četností stejně dlouhá a nebo kratší kódová slova než znaky s menší četností. Pro libovolné dva znaky s nejmenšími četnostmi existuje optimální prefixový kód, v němž mají tyto znaky kódová slova o stejné délce, která se liší pouze v posledním bitu.

10.3.1 Algoritmus konstrukce

vstup Množina znaků M, četnost $f(a)$ ($f(z) \neq 0$ pro $\forall z \in M$).

Výstup Prefixový strom (imlicitně reprezentující kód).

Datové struktury Prioritní fronta F, klíčem prvku je jeho četnost $f(x)$.

Algoritmus
`function` *tree*
$F := M$
`for` $i := n - 1$ `do`
  vytvoř nový vrchol v
  $x := leftson(v) := extractminimum(F)$
  $y := rightson(v) := extractminimum(F)$
  $f(v) := f(x) + f(y)$
  $insert(F, v)$
`od`
`return` $extractminimum(F)$

$s \in A^*$, T prefixový strom pro s, $\forall z \in A$ nechť $f(z)$ je četnost (počet výskytů) z ve slově s, $d_T(l)$ je hloubka listu l ve stromě T (délka cesty z kořene do l), pak $d_T(z)$ = délka kódového slova znaku z a cena stromu T je $B(T) = \sum_{z \in A} f(z) \cdot d_T(z)$. $B(T)$ je délka kódu slova s.

Lemma 0 Pokud prefixový strom T ($|V(T)| > 2$) obsahuje vrchol s právě jedním synem, pak T není optimální.

Lemma 1 (hladový výběr) Buď dána abeceda A s četnostmi $f(z)$ pro každý znak $z \in A$. Budtě $x, y \in A$ znaky s nejnižšími četnostmi. Pak existuje optimální prefixový strom T pro A a f, v němž jsou x, y listy maximální hloubky a x je bratrem y.

Lemma 2 (optimální podstruktura) Buď T optimální prefixový strom pro abecedu A s četnostmi $f(z)$, pro každý znak $z \in A$. Budtě x, y listy v T a z jejich otec. Pak $T' = T \setminus \{x, y\}$ je optimální prefixový strom pro abecedu $A' = A \setminus \{x, y\} \cup \{z\}$, kde $f(z) = f(x) + f(y)$.

11 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, rekurzivně
- sloučíme získaná řešení na řešení původní úlohy

11.1 Analýza složitosti

$T(n)$ je čas zpracování úlohy velikosti n, pro $n < c$ předpokládáme $T(n) = \Theta(1)$. $D(n)$ je čas na rozdělení úlohy velikosti n na a podúloh (stejné) velikosti $n/c +$ čas na sloučení řešení podúloh. Dostáváme rekurentní rovnice

$$T(n) = a \cdot T(n/c) + D(n) \text{ pro } n \geq c,\ T(n) = \Theta(1) \text{ pro } n < c.$$

Příklad Třídění algoritmem Mergesort.

$$\underbrace{\underbrace{a_1 \ldots a_{n/2}}_{Mergesort}\ \underbrace{a_{n/2+1} \ldots a_n}_{Mergesort}}_{merge}$$

Dvě ($a = 2$) rekurzívní podúlohy poloviční velikosti ($c = 2$). Dělení sudá-lichá má složitost $O(n)$, první/druhá polovina (v poli) $O(1)$. Sloučení má složitost $O(n)$, celkem $D(n) = O(n)$.
Výsledná rovnice má tvar

$$T(n) = 2 \cdot T(n/2) + \underbrace{O(n)}_{\text{dekompozice, syntéza}}.$$

Používáme zjednodušení

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

11.2 Substituční metoda

Uhádneme asymptoticky správné řešení. Dokážeme, typicky indukcí, správnost odhadu (zvlášť pro dolní a horní odhad). Častou chybou je, že odhady v indukci musí vyjít se stejnou asymptotickou konstantou jako v indukčním předpokladu.

Důkaz (složitosti algoritmu Mergesort)

$$T(n) = 2 \cdot T(n/2) + O(n)$$

$$T(n) = \Theta(n \log n),\ \exists n_0\ \forall n > n_0,\ \exists c_1, c_2,\ 0 \leq c_1 n \log n \leq T(n) \leq c_2 n \log n$$

Dolní odhad

$$T(n/2) \geq c_1 (n/2) \log n/2$$

$$T(n) = 2 \cdot T(n/2) + bn \geq 2 \cdot c_1 \frac{n}{2} \log \frac{n}{2} + bn =$$

$$= c_1 n(\log n - 1) + bn = c_1 n \log n + \underbrace{(b - c_1)}_{\geq 0} n \geq c_1 n \log n$$

$$c_1' = min(c_1, b)$$

Horní odhad

$$T(n/2) \leq c_2 (n/2) \log n/2$$

$$T(n) = 2 \cdot T(n/2) + bn \leq 2 \cdot c_2 \frac{n}{2} \log \frac{n}{2} + bn =$$

$$= c_2 n(\log n - 1) + bn = c_2 n \log n + \underbrace{(b - c_2)}_{\leq 0} n \leq c_2 n \log n$$

$$c_2' = max(c_2, b)$$

11.3 Master theorem

Nechť $a \geq 1$, $c > 1$, $d \geq 0$ jsou reálná čísla a nechť $T : \mathbb{N} \to \mathbb{N}$ je neklesající funkce taková, že pro všechna n ve tvaru c^k, pro $k \in \mathbb{N}$, platí

$$T(n) = a \cdot T(n/c) + F(n),$$

kde pro funkci $F : \mathbb{N} \to \mathbb{N}$ platí

$$F(n) = O(n^d).$$

Označme $x = \log_c a$. Potom

1. pokud $a < c^d$, tj. $x < d$, potom $T(n) = O(n^d)$
2. pokud $a = c^d$, tj. $x = d$, potom $T(n) = O(n^d \cdot \log_c n)$
3. pokud $a > c^d$, tj. $x > d$, potom $T(n) = O(n^x)$

Příklady

- Mergesort: $T(n) = 2 \cdot T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- binární vyhledávání v utříděném poli: $T(n) = 1 \cdot T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$
- násobení dlouhých čísel – klasické: $T(n) = 4 \cdot T(n/2) + O(n) \Rightarrow T(n) = O(n^2)$
- násobení dlouhých čísel – rychlé: $T(n) = 3 \cdot T(n/2) + O(n) \Rightarrow T(n) = O(n^{\log 3)}$
- násobení matic – klasické: $T(n) = 8 \cdot T(n/2) + O(n^2) \Rightarrow T(n) = O(n^3)$
- kreslení fraktální křivky: $T(n) = 4 \cdot T(n/2) + O(1) \Rightarrow T(n) = O(n^{\log_3 4})$
- hledání mediánu: $T(n) = T(n/5) + T(7n/10) + O(n) \Rightarrow T(n) = O(n)$ substituční metodou

Důkaz

$$T(n) = a \cdot T(n/c) + \underbrace{F(n)}_{O(n^d)}$$

$$\exists e, n_0,\ \forall n \geq n_0\ F(n) \leq en^d$$

$$\exists m,\ c^m \geq n_0\ \Rightarrow\ \forall k \geq m\ F(c^k) \leq ec^{kd}$$

$$b = max\{T(c^m), ec^{md}\}$$

$$n = c^{m+k} = c^m \cdot c^k$$

$$T(n) \leq a \cdot T(n/c) + b \cdot (c^k)^d,\ k > 0$$

$$T(n) \leq bc^{kd} \cdot \underbrace{\sum_{i=0}^{k} \left(\frac{a}{c^d}\right)^i}_{s_k = \frac{\left(\frac{a}{c^d}\right)^k - 1}{\frac{a}{c^d} - 1}}$$

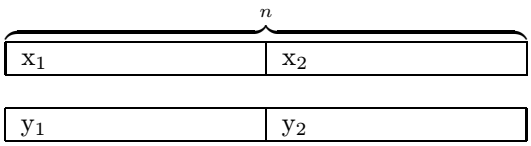
1. pro $a < c^d$ platí:
   $s_k < s = \dfrac{c^d}{c^d - a} = \text{konstanta}$
   $T(n) \leq \text{konstanta} \cdot c^{kd} = O(n^d)$
2. pro $a = c^d$ platí:
   $s_k = k$
   $T(n) \leq \text{konstanta} \cdot c^{kd} k = O(n^d \cdot \log_c n)$

3. pro $a > c^d$ platí:

$$s_k < \frac{\left(\frac{a}{c^d}\right)^k}{\frac{a}{c^d} - 1} = \text{konstanta} \cdot \left(\frac{a}{c^d}\right)^k$$

$$T(n) \leq \text{konstanta} \cdot c^{kd} \left(\frac{a}{c^d}\right)^k = \text{konstanta} \cdot c^{(k \cdot \log_c a)} = O(n^{\log_c a})$$

11.4 Násobení binárních čísel



$x = x_1 \cdot 2^m + x_2,\ y = y_1 \cdot 2^m + y_2$ přičemž $m = n$ div 2
$x \cdot y = x_1 \cdot y_1 \cdot 2^m + (x_1 \cdot y_2 + x_2 \cdot y_1) \cdot 2^m + x_2 \cdot y_2$

$$T(n) = 4 \cdot T(n/2) + O(n),\ n = 2^k \ \Rightarrow\ T(n) = O(n^{log_2 4}) = O(n^2)$$

Master theorem $a = 4$, $c = 2$, $d = 1$, $a > c^d$.

Idea zlepšení složitosti je zmenšit a.

$u = (x_1 + x_2) \cdot (y_1 + y_2),\ v = x_1 + y_1,\ w = x_2 + y_2$
$x \cdot y = z = v \cdot 2^n + (u - v - w) \cdot 2^m + w$

$$T(n) = 3 \cdot T(n/2) + O(n) \ \Rightarrow\ T(n) = O(n^{\log_2 3}) = O(n^{1,59})$$

Master theorem $a = 3$, $c = 2$, $d = 1$.

Obecně $x_1 + x_2$ a $y_1 + y_2$ mohou být o 1 bit delší než $n/2$ což ošetříme rozborem případů.

11.5 Násobení čtvercových matic

Úlohou je pro dané 2 matice A, B řádu $n \times n$ spočítat $C = A \otimes B$, řádu $n \times n$.

11.5.1 Klasicky

Klasický algoritmus má složitost n^3, počítáme n^2 skalárních součinů délky n.

11.5.2 Rozděl a panuj

Předpokládejme že n je mocnina čísla 2, tj. $\exists k,\ n = 2^k$. Potom vstupní matice můžeme dělit na 4 matice poloviního řádu (až do matic 1×1).
Použijeme metodu rozděl a panuj.

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

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

Počet maticových operací na maticích řádu $n/2$ je 8 násobení $\otimes$ a 4 sčítání $\oplus$ (a pomocné operace). Počet sčítání realných čísel v maticovém sčítání $4(n/2)^2 = n^2$. Vyšla rovnice

$$T(n) = 8 \cdot T(n/2) + O(n^2)$$

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) $d = 2$.

11.5.3 Strassenův algoritmus násobení matic

Používá pouze 7 násobení matic řádu $n/2$.
$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$
$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$
$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$
$M_4 = (A_{11} \ominus A_{12}) \otimes B_{22}$
$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$
$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$
$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$

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 $n/2$ je 7 násobení $\otimes$ a celkem 18 sčítání $\oplus$ a odčítání $\ominus$. Složitost

$$T(n) = 7 \cdot T(n/2) + O(n^2)$$

Master theorem $a = 7$, $c = 2$, $\log_c a = 7 = x$, $d = 2$, tedy $T(n) = O(n^x) = O(n^{2,81})$.
Praktické použití pro husté matice řádu $n > 45$ (větší asymptotická konstanta než u klasického násobení).

Poznámka Strassenův algoritmus používá odčítání, tj. inverzní prvky vzhledem ke sčítání, tedy pracuje nad (maticí) okruhem (s operací + a ×). Proto nejde použít pro počítání minimálních cest (s operací *min* a +) ani pro boolovské matice (s operacemi *and* a *or*). Ale lze ho použít na 0-1 matice (místo boolovských) pro výpočet tranzitivního úzávěru grafu kde 0 reprezentuje $false$, kladné číslo *true*.

$$\begin{array}{c} \\ A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{array} \begin{array}{c} \begin{array}{cccc} B_{11} & B_{21} & B_{12} & B_{22} \end{array} \\ \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}$$

$$\mathrm{M}_1 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix}$$

$$\mathrm{M}_2 = \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}$$

$$\mathrm{M}_3 = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{M}_4 = \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{M}_5 = \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{M}_6 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{M}_7 = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{C}_{11} = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{C}_{12} = \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{C}_{21} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix}$$

$$\mathrm{C}_{22} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

11.6 Quicksort

`function` $quicksort(M)$
`if` $|M| > 1$ `then`
  $pivot :=$ libovolný prvek z M
  $M_1 := \{m | m < pivot\}$
  $M_2 := \{m | m \geq pivot\} \setminus \{pivot\}$
  $quicksort(M_1)$
  $quicksort(M_2)$
`fi`

Analýza složitosti

$$T(0) = T(1) = 1$$

$$T(n) = 1 + n + T(n-k) + T(k-1), \text{ pivot je } k\text{-tý}$$

V nejlepším případě kdy $k = n/2$ je

$$T(n) = 2 \cdot T(n/2) + O(n) \;\Rightarrow\; T(n) = O(n \log n).$$

V nejhorší případě kdy $k = 1$ ($k = n$) je

$$T(n) = 1 + n + T(n-1) \;\Rightarrow\; T(n) = O(n^2).$$

Předpoklady o pravděpodobnostním rozložení Vstupní permutaci čísel $1 \ldots n$, všechny se stejnou pravděpodobností, z toho vyplývá že $pivot = k$ se stejnou pravděpodobností. Při vytvoření posloupnosti M_1 a M_2 potřebujeme zaručit, že jsou to náhodné permutace vhodnou volbou algoritmu.

Očekávaná doba výpočtu

$$ET(0) = ET(1) = 0$$

$$ET(n) = \sum_{k=1}^{n} (n + 1 + ET(k-1) + ET(n-k)) \cdot \frac{1}{n},\ n \geq 2$$

$$ET(n) = 1 + n + \frac{2}{n} \cdot \sum_{k=0}^{n-1} ET(k)$$

$$\left(ET(n+1) = 2 + \frac{n+2}{n+1} \cdot ET(n) \right)$$

$$ET(n) = \sum_{i=2}^{n} 2 \cdot \frac{n+1}{i+1} \approx 2(n+1) \log(n+1) = O(n \log n)$$

Používáme vztah pro harmonická čísla $M_n = \sum_{i=1}^{n} \frac{1}{i} \approx \log n$. Analogický výpočet pro průměrnou hloubku binárního stromu.

11.6.1 Randomizovaný Quicksort

Problém pevného výběru pivota Určité vstupní posloupnosti se třídí v čase $O(n^2)$. Pokud se nevhodné posloupnosti vyskytují s větší pravděpodobností, pak se očekávaný čas může blížit $O(n^2)$.

Řešení Volání pivota náhodně. 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 vstp můžeme zvolit špatné pivoty (špatná volba vždy existuje), kdy doba výpočtu je $O(n^2)$.

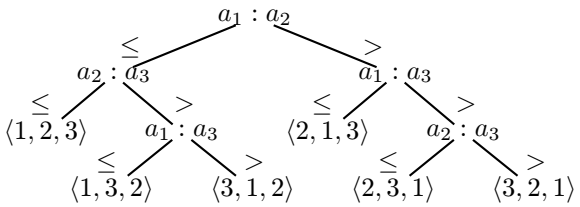
12 Dolní odhad třídění založeného na porovnávání prvků

Rozhodovací strom Reprezentuje porovnávání vykonaná při běhu třídícího algoritmu (na $a_1, \dots a_n$)

- vnitřní uzly označené $a_i : a_j$ ($a_i \leq a_j$, $a_i > a_j$), pro $1 \leq i, j \leq n$
- listy označené permutací $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ odpovídají výsledku $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$

Běh algoritmu odpovídá cestě z kořene do listu. V listech se musí objevit všech $n!$ permutací (pokud se neobjeví nějaká permutace, jsme schopni předložit inverzní permutaci na vstup a algoritmus ji nedokáže utřídit).

Insertsort, n=3



Věta Libovolný rozhodovací strom pro n prvků má výšku $\Omega(n \log n)$.

Důkaz Pro hloubku h stromu platí $n! \leq 2^h$, protože 2^h je maximální počet listů. Odkud zlogaritmováním $h \geq \log n!$.

(Odhad faktoriálu: $n! \geq n \cdot (n-1) \cdot \ldots \cdot 2 \geq n \cdot (n-1) \cdot \ldots \cdot \frac{n}{2} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$)

$$h \geq \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{n}{2} \log \left(\frac{n}{2}\right) = \frac{1}{2} n (\log n - 1) = \Omega(n \log n)$$

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

12.1 Lineární třídící algoritmy

12.1.1 Radixsort

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

Pozorování Pokud byly přihrádky utříděny podle méně významných cifer a třídění bylo stabilní, máme utříděnou posloupnost.

Algoritmus Pro d-místná čísla.
`function` $radixsort(M)$
`for` $i := 1$ `to` d `do`
  utřiď stabilním tříděním podle i-té cifry
`od`

Složitost $O(d \cdot (n + p))$, d se předpokládá konstantní.

12.1.2 Countingsort

Pro přirozená čísla v rozsahu $1, \ldots, k$. Pomocná paměť pro pole C velikosti k což je pole výsledků.

Idea

- při prvním průchodu do pole C uložíme počet výskytů vstupních čísel
- projdeme pole C a sečítáme odspodu počet výskytů menších a rovných čísel (tj. rozmístění ve výsledku)
- při druhém průchodu uložíme vstupní číslo x na $C(x)$-té místo B a dekrementujeme $C(x)$

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

13 LUP dekompozice

Definice 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 $S^{m \times n}$.

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

- transponovaná matice A^T vznikne výměnou sloupců a řádků matice A, tj. matice $A^T = (a_{ij})$
- 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 případy 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 úhlopříčce, sloupce jsou jednotkové vektory e_i, píšeme I bez indexu, pokud lze rozměry odvodit z kontextu

- horní trojúhelníková 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 pod diagonálou jsou 0), jednotková dolní trojúhelníková matice má navíc na diagonále pouze jedničky
- 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), taková že 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

13.1 Řešení soustav lineárních rovnic pomocí LUP dekompozice

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

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ & & & & & & & \vdots & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n \end{array}$$

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 $x = A^{-1}Ax = A^{-1}b$. Řešení x je potom jediné.

13.1.1 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.

13.1.2 Metoda LUP

Pro A najdeme tři matice L, U, P rozměrů $n \times n$ tzv. LUP dekompozicí takovou, že $PA = LU$, 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

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

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 soustavy

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

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

$$x_n = y_n/u_{nm},$$

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

obecně

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

Program řešení LUP Přepisem vrozců. Permutační matice P je reprezentována polem π velikosti n, kde $\pi(i) = j$ znamená, že i-tý řádek P obsahuje 1 v j-tém sloupci.

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

13.1.3 Výpočet LU dekompozice

Nejprve jednodušší případ, když matice P chybí (tj. $P = I_n$).

Idea 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 násobily 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, který je roven $L'U'$. S využitím maticových operací odvodíme

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

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 $k = 0, \ldots, n-1$. Výpočet jedné úrovně rekurze (tj. hlavního cyklu) trvá $\Theta(k^2)$ a celkový čas lze odhadnout $\sum_{k=0}^{n-1} k^2 = \Theta(n^3)$.

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 v sloupci nenulový prvek. Takový musí existovat, pokud je matice nesingulární.

Algoritmus

```
function lup(A)
n := rows(A)
for i := 1 to n do
 π(i) := i
od
for k := 1 to n − 1 do
 p := 0
 for i := k to n do
  if |A_ik| > p then
   p := |A_ik|
   k′ := i
  fi
 od
od
if p = 0 then
 return "singular matrix"
fi
exchange(π(k), π(k′))
for i := 1 to n do
 exchange(A_ki, A_k′i)
od
for i := k + 1 to n do
 A_ki := A_ik/A_kk
 for j := k + 1 to n do
  A_ij := A_ij - A_ik * A_kj
 od
od
```

Složitost $\Theta(n^3)$

Implementační poznámky

- stačí počítat nenulové prvky
- obě matice můžeme uložit "na místě", pokud ukládáme pouze významné prvky, tj. nenulové a diagonálu U

13.1.4 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 tvaru $AX = I_n$ můžeme považovat za n ruzných 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 $\Theta(n^3)$, tedy celkem inverzi A^{-1} matice A spočítáme v čase $\Theta(n^3)$.

Souvislosti Lze ukázat že

$$T_{\text{inverze}} = \Theta(T_{\text{nasobení(n)}}),$$

tj. složitost počítání inverze je stejná jako násobení matic.