Státnicové otázky

22. srpna 2011

Obsah

Kapitola 1

Státnice - Informatika - I2: Softwarové systémy

http://www.mff.cuni.cz/studium/bcmgr/ok/i3b52.htm

1.1 Databázové systémy

- Formální základy databázové technologie
- Databázové modely a jazyky
- Implementace databázových systémů

1.2 Softwarové inženýrství

- Programovací jazyky a překladače
- Objektově orientované a komponentové systémy
- Analýza a návrh softwarových systémů

1.3 Systémové architektury

- Operační systémy (státnice)
- Distribuované systémy
- Architektura počítačů a sítí

1.4 Spolehlivé systémy

- Modely a verifikace programů
- Vestavěné systémy a systémy reálného času
- Moderní softwarové systémy

1.5 Počítačová grafika

- Geometrické modelování a výpočetní geometrie
- Analýza a zpracování obrazu, počítačové vidění a robotika
- 2D počítačová grafika, komprese obrazu a videa
- Realistická syntéza obrazu, virtuální realita

1.6 Okruhy povinné pro obory I2 a I3

Kapitola 2

Státnice - Metody tvorby algoritmů

2.1 Popis složitosti algoritmů

Definice (Velikost dat, krok algoritmu)

Velikost dat je obvykle počet bitů, potřebných k jejich zapsání, např. data D jako pro čísla $a_1, \ldots, a_n$ je velikost dat $|D| = \sum_{i=1}^{n} \lceil \log a_i \rceil$.

Krok algoritmu je jedna operace daného abstraktního stroje (např. Turingův stroj, stroj RAM), zjednodušeně jde o nějakou operaci, proveditelnou v konstantním čase (např. aritmetické operace, porovnání hodnot, přiřazení – pro jednoduché číselné typy).

Definice (Časová složitost)

Časová složitost je funkce $f : \mathbb{N} \to \mathbb{N}$ taková, že $f(|D|)$ udává počet kroků daného algoritmu, pokud je spuštěn na datech D.

Definice (Asymptotická složitost)

Řekneme, že funkce $f(n)$ je asymptoticky menší nebo rovna než $g(n)$, značíme $f(n)$ je $O(g(n))$, právě tehdy, když

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

Funkce $f(n)$ je asymptoticky větší nebo rovna než $g(n)$, značíme $f(n)$ je $\Omega(g(n))$, právě tehdy, když

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

Funkce $f(n)$ je asymptoticky stejná jako $g(n)$, značíme $f(n)$ je $\Theta(g(n))$, právě tehdy, když

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

Funkce $f(n)$ je asymptoticky ostře menší než $g(n)$ ($f(n)$ je $o(g(n))$, když

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

Funkce $f(n)$ je asymptoticky ostře větší než $g(n)$ ($f(n)$ je $w(g(n))$, když

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

Poznámka

Asymptotická složitost zkoumá chování algoritmů na velkých datech, zařazuje je podle toho do kategorií. Zanedbává multiplikativní a aditivní konstanty.

2.2 Rozděl a panuj

Definice (Metoda rozděl a panuj)

Rozděl a panuj je metoda návrhu algoritmů (ne strukturované programování), která má 3 kroky:

1. rozděl – rozdělí úlohu na několik podúloh stejného typu, ale menší velikosti
2. vyřeš – vyřeší podúlohy a to buď přímo pro dostatečně malé, nebo rekurzivně pro větší
3. sjednoť – sjednotí řešení podúloh do řešení původní úlohy

Aplikace

- QUICKSORT
- Hledání mediánu

Analýza složitosti algoritmů rozděl a panuj

Poznámka (Vytvoření rekurentní rovnice)

Pro časovou složitost algoritmů typu rozděl a panuj zpravidla dostávám nějakou rekurentní rovnici.

- $T(n)$ budiž doba zpracování úlohy velikosti n, za předpokladu, že $T(n) = \Theta(1)$ pro $n \leq n_0$.
- $D(n)$ budiž doba na rozdělení úlohy velikosti n na a podúloh stejné velikosti $\frac{n}{c}$.
- $S(n)$ budiž doba na sjednocení řešení podúloh velikosti $\frac{n}{c}$ na jednu úlohu velikosti n. Dostávám rovnici

$$T(n) = \begin{cases} D(n) + aT(\frac{n}{c}) + S(n) & n > n_0 \\ \Theta(1) & n \leq n_0 \end{cases}$$

Poznámka

Při řešení rekurentních rovnic:

- Zanedbávám celočíselnost ($\frac{n}{2}$ místo $\lceil \frac{n}{2} \rceil$ a $\lfloor \frac{n}{2} \rfloor$)
- Nehledím na konkrétní hodnoty aditivních a multiplikativních konstant, asymptotické notace používám i v zadání rekurentních rovnic, i v jejich řešení.

Věta (Substituční metoda)

1. Uhodnu asymptoticky správné řešení
2. Indukcí ověřím správnost (zvláště horní a dolní odhad)

Věta (Metoda "kuchařka" (Master Theorem))

Nechť $a \geq 1, c > 1, d \geq 0 \in \mathbb{R}$ a nechť $T : \mathbb{N} \to \mathbb{N}$ je neklesající funkce taková, že $\forall n$ tvaru c^k platí

$$T(n) = aT(\frac{n}{c}) + \Theta(n^d)$$

Potom

1. Je-li $\log_c a \neq d$, pak $T(n)$ je $\Theta(n^{\max\{\log_c a, d\}})$
2. Je-li $\log_c a = d$, pak $T(n)$ je $\Theta(n^d \log_c n)$

Věta (Master Theorem, varianta 2)

Nechť $0 < a_i < 1$, kde $i \in \{1, \ldots, k\}$ a $d \geq 0$ jsou reálná čísla a nechť $T : \mathbb{N} \to \mathbb{N}$ splňuje rekurenci

$$T(n) = \sum_{i=1}^{k} T(a_i \cdot n) + \Theta(n^d)$$

Nechť je číslo x řešením rovnice $\sum_{i=1}^{k} a_i^x = 1$. Potom

1. Je-li $x \neq d$ (tedy $\sum_{i=1}^{k} a_i^d \neq 1$), pak $T(n)$ je $\Theta(n^{\max\{x,d\}})$
2. Je-li $x = d$ (tedy $\sum_{i=1}^{k} a_i^d = 1$), pak $T(n)$ je $\Theta(n^d \log n)$

2.3 Dynamické programování

Dynamické programování je metoda řešení problemů, které v sobě obsahují překrývající se subproblémy.

Příklad

Typickým příkladem je výpočet fibonaciho čísla. Fib. posloupnost je definována jako:

$$f(0) = 1, f(1) = 1$$

$$f(n+2) = f(n+1) + f(n)$$

Výpočet třeba 4. čísla by pak byl $f(4) = f(3)+f(2) = f(2)+f(1)+f(1)+f(0) = f(1)+f(0)+f(1)+f(1)+f(0) = 5$. Vidíme, že jsme $f(2)$ počítali dvakrát, $f(1)$ třikrát a $f(0)$ dvakrát. Ve větším měřítku toto zbytečně počítání vede k exponenciální složitosti algoritmu. Lepší cestou je počítat "od spodu", kdy pomoci $f(0)$ a $f(1)$ spočteme $f(2)$, pak s jeho pomoci $f(3)$ nakonec $f(4)$ s lineární složitosti.

V dynamickém programování se například vytvoří mapa Fibonacciho čísel a jejich hodnot. Před tím, než začnu počítat hodnotu nějakého Fibonacciho čísla, se podívám do mapy.

Nutné podmínky

V dynamickém programování využíváme:

1. Překrývání podproblému — problém lze rozdělit na podproblémy, jejichž řešení se využívá opakovaně.
2. Optimální podstruktury — optimální řešení lze zkonstruovat z optimálních řešení podproblémů.

Postupy

Obvykle se používá jeden ze dvou přístupů k dynamickému programování:

- Top-down — problém se rekurzivně dělí na podproblémy a po jejich vyřešení se zapamatují výsledky, které se použijí při případném opětovném řešení daného podproblému.
- Bottom-up — nejdříve se spočítají všechny možné podproblémy (viz příklad s Fibonacciho posloupnosti), které se potom skládají do řešení větších problémů. Tento přístup je výhodnější z hlediska počtu volání funkci a místa na zásobníků, ale ne vždy musí být zřejmě, které všechny subproblémy je třeba předem spočítat.

Použití

Problém batohu – máme věci různé váhy a batoh určité nosnosti. Které věci máme dát do batohu, abychom jej co nejlépe zaplnili (jednorozměrná varianta problému batohu)? Speciální případ je součet podmnožiny (SP). Algoritmus je popsán v sekci o NP-úplnosti.

Uzávorkování součinu matic tak, aby počet skalárních součinů byl co nejmenší. Dělá se pomocí 2 čtvercových matic (M a K) řádu rovného počtu násobených matic, kde pracujeme jen nad diagonálou. Hodnota na M_{ij} udává minimální počet skalárních součinů při nejlepším uzávorkování matic i až j, hodnota K_{ij} určuje matici, která rozděluje závorkování na dvě podmnožiny matic. Hodnotu M_{ij} lze zkonstruovat ze všech M_{ik} a M_{kj} pro $i < k < j$.

2.4 Hladové algoritmy

Motivace

Problém 1

Dán souvislý neorientovaný graf $G = (V, E)$ a funkce $d : E \to \mathbb{R}^+$, udávající délky hran. Najděte minimální kostru grafu G, tj. kostru $G' = (V, E')$ tak, že $\sum_{e \in E'} d(e)$ je minimální.

Problém 2

Je dána množina $S = \{1, \ldots, n\}$ úkolů jednotkové délky. Ke každému úkolu je dána lhůta dokončení $d_i \in \mathbb{N}$ a pokud $w_i \in \mathbb{N}$, kterou je úkol i penalizován, není-li hotov do své lhůty. Najděte rozvrh (permutaci úkolů od času 0 do času n), který minimalizuje celkovou pokutu.

Problém 3

Je dána množina $S = \{1, \ldots, n\}$ úkolů. Ke každému úkolu je dán čas jeho zahájení s_i ukončení f_i, $s_i \leq f_i$. Úkoly i a j jsou kompatibilní, pokud se intervaly $[s_i, f_i)$ a $[s_j, f_j)$ nepřekrývají. Najděte co největší množinu (po dvou) kompatibilních úkolů.

Matroid

Definice (Matroid)

Matroid je uspořádaná dvojice $M = (S, I)$, splňující:

- S je konečná neprázdná množina (prvky matroidu M)
- I je neprázdná množina podmnožin S (nezávislé podmnožiny), která má

1. dědičnou vlastnost: $B \in I \wedge A \subseteq B \quad \Rightarrow \quad A \in I$,
2. výměnnou vlastnost: $A, B \in I \wedge |A| < |B| \quad \Rightarrow \quad \exists x \in B \backslash A : A \cup \{x\} \in I$.

Věta (O velikosti maximálních nezávislých podmnožin)

Všechny maximální (maximální vzhledem k inkluzi) nezávislé podmnožiny v matroidu mají stejnou velikost.

Důkaz

Z výměnné vlastnosti, nechť $A, B \in I$ maximální, $|A| < |B|$, pak $A \cup \{x\} \in I, x \notin A$, což je spor.

Definice (Vážený matroid)

Matroid $M = (S, I)$ je vážený, pokud je dána funkce $w : S \to \mathbb{R}^+$ a její rozšíření na podmnožiny množiny S je definováno předpisem:

$$A \in S \Rightarrow w(A) = \sum_{x \in A} w(x)$$

Problém 1 a 2 – zobecněný

Pro daný vážený matroid nalezněte nezávislou podmnožinu s co největší vahou (optimální množinu). Protože váhy jsou kladné, vždy se bude jednat o maximální nez. množinu.

Problém 1 je spec. případ tohoto, protože můžeme uvažovat grafový matroid (nad hranami) – $M_G = (S, I)$, kde $S = E$ a pro $A \subseteq E$ platí $A \in I$, pokud hrany z A netvoří cyklus (tj. indukovaný podgraf tvoří les).

Dědičná vlastnost této struktury je zřejmá, výměnnost se dá dokázat následovně: mějme $A, B \subseteq E, |A| < |B|$. Pak lesy z A, B mají $n - a > n - b$ stromů (vč. izolovaných vrcholů). V B musí být strom, který se dotýká ≥ 2 různých stromů z A. Ten obsahuje hranu, která není v žádném stromě A a netvoří cyklus a tak ji můžu k A přidat.

Váhovou funkci převedu na hledání maxim: $w(e) = c - d(e)$, kde c je dost velká konstanta, aby všechny váhy byly kladné. Algoritmus pak najde max. množinu, kde hrany netvoří cyklus. Implicitně bude mít $n - 1$ hran, takže půjde o kostru a její w bude maximální, tedy původní váha minimální.

Pro problém 2 zavedeme pojem kanonického rozvrhu – takového rozvrhu, kde jsou všechny včasné úkoly rozvrženy před všemi zpožděnými a uspořádány podle neklesající lhůty dokončení (tohle na celkové pokutě nic nezmění a máme bijekci mezi množinou včasných úkolů a kanonickými rozvrhy).

Optimální množinu pak lze hledat jen nad kanonickými rozvrhy – nezávislou množinou úkolů nazvu takovou, pro kterou existuje kanonický rozvrh tak, že žádný úkol v ní obsažený není zpožděný. Potom hledání max. nezávislé množiny při ohodnocení pokutami je hledání nejmenší pokuty (odeberu co nejvíc z možné celkové pokuty).

Pak zbývá dokázat, že takto vytvořená struktura je matroid. Dědičná vlastnost je triviální – vyhodím-li něco z nezávislé množiny, nechám v rozvrhu mezery a bude platit stále. Pro výměnnou vlastnost zavedu pomocnou funkci $N_t(C) = |\{i \in C | d_i \leq t\}|$, udávající počet úkolů z množiny C se lhůtou do času t. Pak množina C je nezávislá, právě když $\forall t \in \{1, \ldots, n\} : N_t(C) \leq t$.

Pak máme-li 2 nezávislé A, B, $|B| > |A|$, označíme k největší okamžik takový, že $N_k(B) \leq N_k(A)$, tj. od $k+1$ dál platí $N_t(A) < N_t(B)$. To skutečně nastane, protože $|B| = N_n(B) > N_n(A) = |A|$. Pak určitě v B je ostře víc úkolů s $d_i = k + 1$ než v A, tj. $\exists x \in B \backslash A$ se lhůtou $k + 1$. $A \cup \{x\}$ je nezávislá, protože $N_t(A \cup \{x\}) = N_t(A)$ pro $t \leq k$ a $N_t(A \cup \{x\}) \leq N_t(B)$ pro $t \geq k + 1$.

Hladový algoritmus na váženém matroidu

Algoritmus (Greedy)

Mám zadaný matroid $M = (S, I)$, kde $S = \{x_1, \ldots, x_n\}$ a $w : S \to \mathbb{R}^+$. Potom

1. $A := \emptyset$, setřiď a přeznač S sestupně podle vah
2. pro každé x_i zkoušej: je-li $A \cup \{x_i\} \in I$, tak $A := A \cup \{x_i\}$
3. vrať A jako maximální nezávislou množinu

Pokud přidám nějaké x_i, nikdy nezruším nezávislost množiny; s už přidanými prvky se nic nestane. Časová složitost je $\Theta(n \log n)$ na setřídění podle vah, testování nezávislosti množiny závisí na typu matriodu ($f(n)$), takže celkem něco jako $\Theta(n \log n + n \cdot f(n))$.

Důkaz

- Vyhození prvků, které samy o sobě nejsou nezávislé množiny, nic nezkazí (z dědičné vlastnosti).
- První vybrané x_i je "legální" (nezablokuje mi cestu), protože mezi max. nez. množinami určitě existuje taková, která jím mohla vzniknout (z výměnné vlastnosti – vezmeme první prvek, který je sám o sobě nez. mn. a s pomocí nějaké max. nez. množiny B ho doplníme, vzniklé $\{x_i\} \cup (B \backslash \{x_j\})$ musí být také maximální).
- Nalezení optimální množiny obsahující pevné (první vybrané) x_i v $M = (S, I)$ je ekvivalentní nalezení optimální množiny v $M' = (S', I')$, kde $S' = \{y \in S | \{x_i, y\} \in I\}$ a $I' = \{B \subset S \backslash \{x_i\} | B \cup \{x_i\} \in I\}$ (je-li S' neprázdná, je to matroid, vlastnosti se přenesou z původního; pokud je S' prázdná, tak algoritmus končí) a tedy když vyberu x_i, nezatarasím si cestu k optimu – stačí ukázat, že $A \cup \{x\}$ je optimální v M právě když A je optimální v M'.
- Takže když budu vybírat (indukcí opakovaně) x_i podle váhy, dojdu k optimální množině.

Algoritmus (Hladový algoritmus na problém 3)

Máme $S = \{1, \ldots, n\}$ množinu úkolů s časy startů s_i a konců f_i. Provedeme:

1. $A := \emptyset$, $f_0 := 0, j := 0$
2. setřiď úkoly podle f_i vzestupně a přeznač
3. pro každé i zkoušej: je-li $s_i \geq f_j$, pak $A := A \cup \{i\}$ a $j := i$
4. vrať A jako max. množinu nekryjících se úkolů

Složitost je $n \log n$ na setřídění, zbytek už lineární. Sice to funguje, ale tahle struktura NENÍ matroid – nesplňuje výměnnou vlastnost. Algoritmus má ale stejný důkaz jako předchozí na matroidech (legálnost hladového výběru – existuje max. množina obsahující prvek 1 – a existenci optimální množiny – převod na "menší" zadání).

Kapitola 3

Státnice - Odhady složitosti

3.1 Dolní odhady složitosti problémů

Definice (Složitost problému)

Složitost problému je složitost asymptoticky nejlepšího možného algoritmu, který řeší daný problém (ne nejlepšího známého).

Každý konkrétní algoritmus dává horní odhad složitosti. Dolní odhady (až na triviální – velikost vstupu, výstupu) jsou složitější.

Věta (Dolní odhad složitosti mediánu)

Pro výběr k-tého z n prvků je třeba alespoň $n-1$ porovnání, tj. problém je $\Omega(n)$.

Důkaz

Intuitivně potřebuju medián, i kdyby mi spadnul z nebe, porovnat se všemi ostatními prvky, abych vůbec zjistil, jestli to je medián ...

Věta (Dolní odhad složitosti třídění)

Pro každý třídící algoritmus, založený na porovnávání prvků, existuje vstupní posloupnost, pro kterou provede $\Omega(n \log n)$ porovnání.

Důkaz

Nakreslím si rozhodovací strom jako model algoritmu – všechny vnitřní uzly odpovídají nějakému porovnání, které algoritmus provedl, jejich synové jsou operace, které nasledovaly po různých výsledcích toho porovnání (BÚNO jsou-li prvky různé, bude strom binární). Listy odpovídají setříděným posloupnostem. Aby byl algoritmus korektní, musí mít strom listy se všemi $n!$ možnými pořadími prvků.

Pro plýtvající algoritmus mohou existovat listy, neodpovídající žádné permutaci, tj. porovnává stejnou dvojici prvků dvakrát (a jedna z možností už nemůže nastat). Pro rovnoměrné rozdělení je očekávaný čas průměrná délka cesty od kořene k listům, nejhorší čas je výška stromu.

Označím výšku jako h, pak počet listů je $\leq 2^h$ a tedy $n! \leq 2^h$, tj. $h \geq \log n!$, pro dolní odhad $\Omega(n \log n)$ stačí odhad faktoriálu $n! < n^{\frac{n}{2}}$, případně můžu použít Stirlingův vzorec $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ a dostávám $\Theta(n \log n)$.

3.2 Amortizovaná složitost

Definice (Amortizovaná složitost)

Typicky se používá pro počítání časové složitosti operací nad datovými strukturami, počítá průměrný čas na 1 operaci při provedení posloupnosti operací. Dává realističtější odhad složitosti posloupnosti všech operací, než měření všech nejhorším případem.

Známe 3 metody amortizované analýzy:

- agregační
- účetní
- potenciálová

Problémy:

- Inkrementace binárního čítače – do binárního čítače délky k postupně přičteme n-krát jedničku. Počet bitových operací na 1 přičtení je v nejhorším případě $O(\log n)$, ale amortizovaně dojdeme k $O(1)$.
- Vkládání do dynamického pole – začnu s prázdným polem délky 1 a postupně vkládám n prvků. Pokud je stávající pole plné, alokujeme dvojnásobné a kopírujeme prvky. Počet kopírování prvků na jedno vložení je až $O(n)$, ale amortizovaně opět $O(1)$.

Algoritmus (Agregační metoda)

Spočítáme nejhorší čas pro celou posloupnost n operací – $T(n)$, amortizovaný čas na 1 operaci je pak $\frac{T(n)}{n}$.

- **Binární sčítání**: v průběhu n přičtení se i-tý bit překlopí $\lfloor \frac{n}{2^i} \rfloor$-krát, takže celková cena překlopení je $\leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$, tj. amortizovaně na jedno přičtení $\frac{2n}{n} = \Theta(1)$
- **Vkládání**: cena i-tého vložení do pole je $c_i = \begin{cases} i \text{ pokud } \exists k : i - 1 = 2^k \\ 1 \text{ jinak} \end{cases}$. Celkem dostávám $T(n) = \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + 2n = 3n$, takže na jedno vložení vyjde $\frac{3n}{n} = \Theta(1)$.

Algoritmus (Účetní metoda)

Od každé operace vyberu urč. pevný "obnos", kterým onu operaci "zaplatím". Pokud něco zbyde, dám to na účet, pokud bude oprace naopak dražší než onen obnos, z účtu vybírám. Zůstatek na účtu musí být stále nezáporný – pokud uspěji, obnos je amortizovaná cena 1 operace.

- **Binární sčítání**: Při každém přičtení je právě jeden bit překlopen z 0 na 1. Proto každému bitu zavedeme účet a za přičtení budeme vybírat 2 jednotky. Jedna je použita na překlopení daného bitu z 0 na 1 a druhá uložena na právě jeho účet; překlápění z 1 na 0 jsou hrazeny z účtů (protože každý bit, který má nastavenu 1 má na účtu právě 1 jednotku, projde to). Amortizovaná cena tedy vyjde $2 = \Theta(1)$.
- **Vkládání**: Od každého vložení vyberu 3 jednotky – na vlastní vložení, na překopírování právě vloženého prvku při příštím vložení a na příští překopírování odpovídajícího prvku v levé polovině pole (na pozici $n - \frac{s}{2}$), který obnos ze svého vložení vyčerpal. Po expanzi je celkem na všech účtech 0, jindy víc, tj. amortizovaná cena operace je $3 = \Theta(1)$.

Algoritmus (Potenciálová metoda)

Je to podobné bankovní, roli účtu hraje nějaká funkce w, která popisuje vhodnost jednotlivých konfigurací $D_0, D_1, \ldots$. Potřebuji potom, aby $w(D_i) \geq 0 \ \forall i$. Amortizovaná složitost i -té operace o je potom:

$$am(o_i) = T(o_i) + w(D_{i+1}) - w(D_i)$$

Složitost nejhoršího případu celé posloupnosti operací může být mnohem "rychlejší" než posloupnost nejhorších případů jednotlivých operací:

$$\sum_{i=1}^{n} T(o_i) \leq \sum_{i=1}^{n} am(o_i) + w(D_0)$$

Kapitola 4

Státnice - NP-úplnost

4.1 Třídy P a NP, polynomiální převody, NP-úplnost

Definice (Úloha)

- Úloha je situace, kdy pro daný vstup (instanci úlohy) chceme získat výstup se zadanými vlastnostmi.
- Optimalizační úloha je úloha, kde cílem je získat optimální (zpravidla největší nebo nejmenší) výstup s danými vlastnostmi.
- Rozhodovací problém je úloha, jejímž výstupem je ANO/NE.

Definice (Kódování vstupů)

Každá instance problému Q je kódována jako posloupnost 0 a 1, tj. instance je slovo v abecedě $\{0,1\}^*$. Kódy všech instancí problému Q tvoří jazyk $L(Q)$ nad abecedou $\{0,1\}^*$, který se dělí na

- $L(Q)_Y$ – kódy instancí s odpovědí ANO (jazyk kladných instancí)
- $L(Q)_N$ – kódy instancí s odpovědí NE (jazyk záporných instancí)

Rozhodovací problém pak je rozhodnutí, zda $x \in L(Q)_Y$ nebo $x \in L(Q)_N$ (kde x je kód nějaké instance Q), když předpokládáme, že rozhodnutí $x \in L(Q)$ lze udělat v polynomiálním čase vzhledem k $|x|$.

Definice (Deterministický Turingův stroj)

DTS obsahuje řídící jednotku, čtecí a zápisovou hlavu a (nekonečnou) pásku. Program sestává z:

1. Konečné množiny Γ páskových symbolů, $\Sigma \subset \Gamma$ vstupních symbolů a $* \in \Gamma$ prázdného symbolu
2. Konečné množiny Q stavů řídící jednotky, která obsahuje startovní stav q_0 a 2 terminální stavy q_Y, q_N
3. Přechodové funkce $\delta : (Q \backslash \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\}$

DTS s programem M přijímá $x \in \Sigma^*$, právě když pro vstup x se M zastaví ve stavu q_Y. Jazyk rozpoznávaný programem M je $L(M) = \{x \in \Sigma^* | M \text{ prijima } x\}$.

DTS s programem M řeší problém Q, právě když výpočet M skončí pro každý vstup $x \in \Sigma^*$ a platí $L(M) = L(Q)_Y$.

Nechť M je program pro DTS, který skončí pro $\forall x \in \Sigma^*$. Časová složitost programu M je dána funkcí $T_M(n) = \max\{m | \exists x \in \Sigma^*, |x| = n$, výpočet na DTS s programem M a vstupem x skončí po m krocích stroje$\}$. Pokud existuje polynom p tak, že $T_M(n) \leq p(n) \forall n$, pak M je polynomiální DTS program.

Definice (Třída P)

Problém Q je ve třídě P, právě když existuje polynomiální DTS program M, který řeší Q.

Definice (Nedeterministický Turingův stroj)

Stejný jako DTS, ale místo přechodové funkce δ je zde zobrazení δ, které každé dvojici z $Q \times \Gamma$ přiřazuje množinu možných pokračování výpočtu, tj. trojic z $Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\}$.

NTS s programem M přijímá $x \in \Sigma^*$, právě když existuje přijímající výpočet programu M (tj. běh M, kdy na vstupu je x a končí se ve stavu q_Y). Jazyk rozpoznávaný programem M je $L(M) = \{x \in \Sigma^* | M \text{ prijima } x\}$.

Čas, ve kterém M přijímá $x \in \Sigma^*$ definujeme jako počet kroků nejkratšího přijímajícího výpočtu nad daty x.

Časová složitost programu je dána funkcí:

$$T_M(n) = \begin{cases} 1 \text{ neexistuje } x \text{ delky } n, \text{ ktere je prijimano} \\ \max\{m | \exists x \in \Sigma^*, |x| = n, M \text{ prijima } x \text{ v case } m\} \end{cases}$$

Pokud existuje polynom p takový, že $T_M(n) \leq p(n)$, pak M je polynomiální NTS program.

Definice (Třída NP)

Problém Q je ve třídě NP, právě když existuje polynomiální NTS program M, který řeší Q. Na rozdíl od deterministického případu netrváme na tom, že výpočet musí skončit i pro nepřijímané instance.

Poznámka (Jiný model NTS)

Přidáme další pásku (orákulum) a stroj pracuje ve 2 fázích:

1. Nedeterministicky hádá – zapíše problém do orákula.
2. Deterministicky ověřuje obsah orákula – práce DTS na původním vstupu plus obsahu orákula.

Je to ekvivalentní s původním – omezíme-li počet možných přechodů NTS na 2 (tím ho jen zpomalíme) a zapisujeme-li do orákula větve pokračování výpočtu (pak stačí na jednu jeden bit), převedeme veškerý nedeterminismus čistě na naplnění orákula.

Definice (Třída co-NP)

Problém Q je ve třídě co-NP, právě když existuje polynomiální NTS program M takový, že $L(M) = L(Q)_N$. O poměru množin co-NP a NP nevíme nic, jen to, že podmnožinou jejich průniku je P.

Převody a NP-úplnost

Definice (Polynomiálně vyčíslitelná funkce)

Funkce $f : \{0,1\}^* \to \{0,1\}^*$ je polynomiálně vyčíslitelná, právě když existuje polynom p a algoritmus A takový, že pro každý vstup $x \in \{0,1\}^*$ dává výstup $f(x)$ v čase nejvýše $p(|x|)$.

Definice (Polynomiální převoditelnost)

Jazyk L_1 je polynomiálně převoditelný na jazyk L_2 (píšeme $L_1 \propto L_2$), právě když existuje polynomiálně vyčíslitelná funkce f taková, že

$$\forall x \in \{0,1\}^* : x \in L_1 \equiv f(x) \in L_2$$

Definice (NP-těžký, NP-úplný problém)

- Problém Q je NP-těžký, právě když $\forall Q' \in \text{NP} : L(Q')_Y \propto L(Q)_Y$.
- Problém Q je NP-úplný, právě když je Q NP-těžký a $Q \in \text{NP}$.

Je-li nějaký NP-těžký problém převoditelný na jiný, pak ten musí být také NP-těžký.

4.2 Příklady NP-úplných problémů a převody mezi nimi

Cook-Levinova věta

Existuje NP-úplný problém.

Důkaz pro KACHL

Máme množinu barev B, čtvercová síť S s obvodem obarveným barvami z B a množinu K typů kachlíků, kde je každý typ definován svou horní, dolní, levou a pravou barvou.

Lze síť S vykachlíkovat pomocí kachlíků z množiny K (stejný typ lze použít libovolněkrát, kachlíky ale nelze otáčet) tak, aby:

- barvy kachlíků přilehlé k obvodu sítě souhlasily s barvami předepsanými tomto na obvodu sítě a
- každá dvojice barev na dotyku dvou kachlíků byla rovněž shodná?

NP-úplné problémy

Splnitelnost (SAT)

CNF (booleovská formule v konjunktivní normální formě, tj. konjunkce disjunkcí) F na n proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F?

Důkaz transformací **KACHL** $\propto$ **SAT**: pomocí proměnných x_{ijk}, kde $x_{ijk} = 1$, pokud na pozici $[i, j]$ se nachází kachlík typu k. Jednotlivé klauzule se vytvoří tak, aby zaručovaly, že na každé pozici je právě jeden kachlík, že kachlíky navazují horizontálně, vertikálně i na kraje stěny.

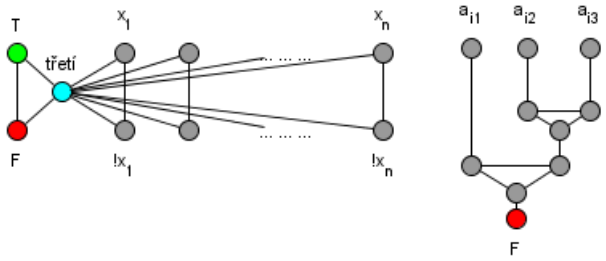
3-SAT

Kubická CNF (vždy jen 3 proměnné v jedné disjunkci) F na n Booleovských proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F?

Transformace **SAT** $\propto$ **3-SAT**: stačí každou klauzuli (disjunkci) rozložit s pomocí nových volných proměnných na několik kubických klauzulí: $(a_{i,1} \vee a_{i,2} \vee a_{i,3} \vee \cdots \vee a_{i,k_i})$ odpovídá $(a_{i,1} \vee a_{i,2} \vee y_{i,1}) \wedge (\neg y_{i,1} \vee a_{i,3} \vee y_{i,2}) \wedge (\neg y_{i,2} \dots) \wedge \cdots \wedge (\neg y_{i,k_i-3} \vee a_{i,k_i-1} \vee a_{i,k_i})$

3-COLOR

Tříobarvení grafu: Mějme neorientovaný graf $G = (V, E)$. Lze obarvit vrcholy ve V třemi barvami tak, aby žádná hrana v E neměla na obou koncích vrcholy stejné barvy?



Obrázek 4.1: Transformace **3-SAT** $\propto$ **3-COLOR**

Transformace **3-SAT** $\propto$ **3-COLOR**: Vytvořím pro všechny proměnné a jejich negace vrcholy grafu a spojím se třemi body (z nichž každý musí být jinak barevný podle obrázku), aby proměnné musely mít barvu T nebo F. Proměnné a negace jsou taky spojené, aby bylo jednoznačně dána hodnota každé z nich. Pro každou klauzuli **3-SAT** přidám grafík podle obrázku (napojím na proměnné, které představují literály klauzule a na druhé straně na barvu F), aby proměnné v něm nešly obarvit FFF.

KLIKA

Mějme neorientovaný graf $G = (V, E)$ a přirozené číslo k. Existuje $V' \subseteq V$, $|V'| = k$, indukující úplný podgraf grafu G?

Transformace **SAT** $\propto$ **KLIKA** – pro každý literál vytvořím bod grafu, spojím všechny body odpovídající literálům různých klauzulí, pokud se nejedná o komplementární proměnné, tj. mezi x_i a $\neg x_i$ nevede hrana.

Nezávislá Množina (NM)

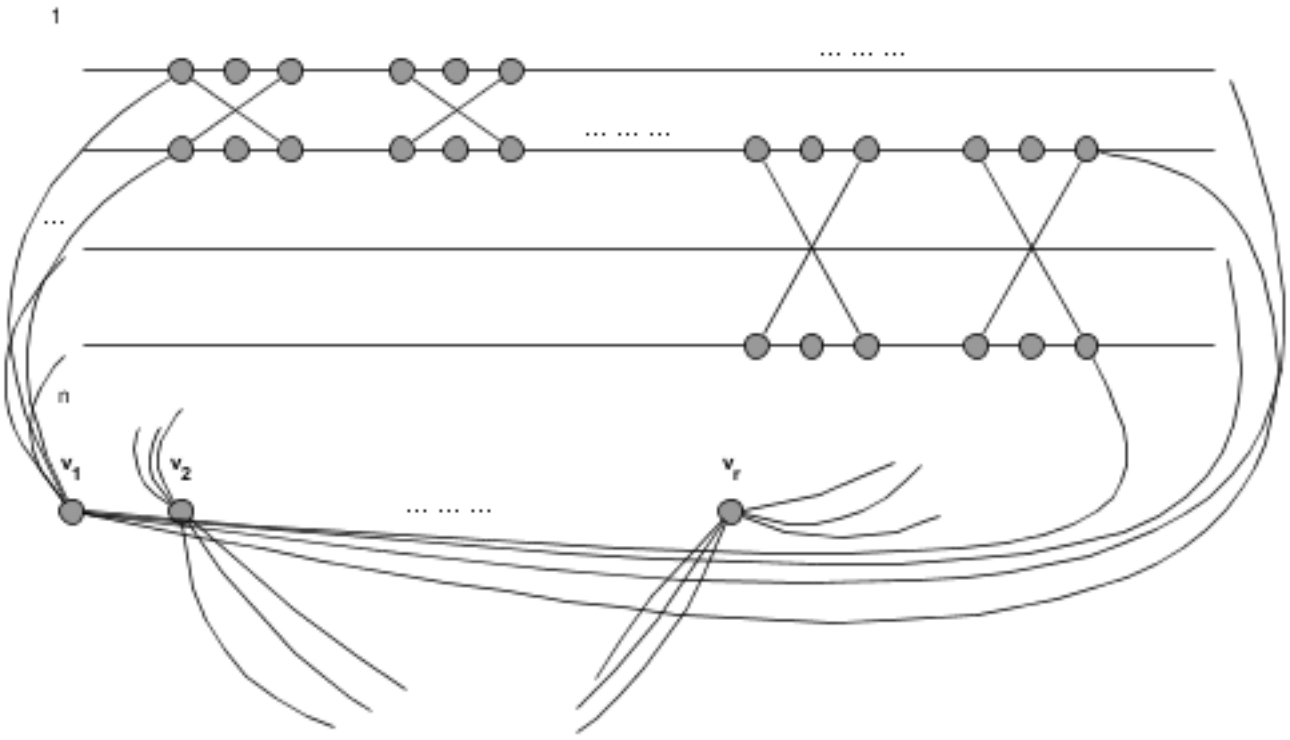
Mějme neorientovaný graf $G = (V, E)$ a přirozené číslo q. Existuje $V' \subseteq V$, $|V'| = q$, taková, že uvnitř V' nejsou žádné hrany?

Transformace **KLIKA** $\propto$ **NM**: stačí prohodit hrany a ne-hrany.

Vrcholové pokrytí (VP)

Máme neorientovaný graf $G = (V, E)$ a přirozené číslo r. Existuje $V' \subseteq V$, $|V'| = r$ taková, že každá hrana má ve V' alespoň jeden vrchol?

Transformace **NM** $\propto$ **VP**: **NM** je doplněk **VP** (vedou-li hrany do **VP**, už nemůžou vést mezi ostatními vrcholy).



Obrázek 4.2: Transformace **VP** $\propto$ **HK**

Hamiltonovská Kružnice (HK)

Máme neorientovaný graf $G = (V, E)$. Obsahuje G hamiltonovskou kružnici, tj. jednoduchou kružnici, která prochází každým vrcholem právě jednou?

Transformace **VP** $\propto$ **HK**: Na $|V|$ pomyslných linkách naskládám pro každou hranu původního grafu dvanáctici vrcholů spojených podle obrázku (widget). Krajní body všech linek spojím s vrcholy odpovídající původnímu **VP** $v_1, \ldots, v_r$. Protože widgety lze projít jen částečně (2x po linkách) nebo úplně (jednou všechny), bude **HK** vést částečným průchodem přes widgety, pokud oba vrcholy příslušné jejich hraně původního grafu patří do **VP** a úplným jinak.

Obchodní cestující (TSP)

Máme úplný neorientovaný graf $G = (V, E)$, váhy $w : E \to \mathbb{Z}_0^+$ a číslo $k \in \mathbb{Z}^+$. Existuje v G hamiltonovská kružnice s celkovou váhou nejvýše k? Někdy se počítá nad neúplným grafem a požaduje se hamiltonovský sled, tj. je možné opakovat vrcholy; to se ale na tuto definici snadno převede.

Transformace **HK** $\propto$ **TSP**: stačí nastavit váhy tak, že $w(e) = 1$, pokud e byla v původním grafu a $w(e) = 2$ jinak. Je-li chtěná váha rovna počtu hran původního grafu, řešení dává **HK** v něm.

Součet podmnožiny (SP)

Jsou daná čísla $a_1, \ldots, a_n, b \in \mathbb{Z}^+$. Existuje množina indexů $S \subseteq \{1, \ldots, n\}$ taková, že $\sum_{i \in S} a_i = b$?

Transformace **VP** $\propto$ **SP**: vyrobím incidenční matici grafu (řádky odp. vrcholům, sloupce hranám), kde budou jedničky na místech, kde daná hrana vede z daného vrcholu. Přidám k ní matici, jejíž řádky i sloupce odpovídají hranám a jedničky jsou pouze na diagonále (tj. každá hrana má jedničku ve "svém" řádku a sloupci). "Nalevo" od incidenční matice přidám sloupec plný jedniček. Řádky matice interpretuju jako čísla ve čtyřkové soustavě (v každém sloupci jsou tři jedničky, proto nedojde nikdy k přesunu řádů) a hledám součet podmnožiny jako číslo, které má na začátku velikost **VP** (sečte se ze sloupce jedniček) a následují samé dvojky (pro každou hranu).

4.3 Silná NP-úplnost, pseudopolynomiální algoritmy

Příklad

SP není exponenciální, ale polynomiální v počtu a velikosti čísel. Algoritmus (dynamické programování):

1. Nechť $a_1 \leq a_2 \leq \cdots \leq a_n$ a A je bitové pole délky b (kde 1 na pozici i bude indikovat možnost vytvoření podmnožiny se součtem i).
2. Všechny prvky pole A nastav na 0.
3. Pro i od 1 do n opakuj (hl. cyklus):
   (a) $A[a_i] := 1$
   (b) Pro j od a_{i-1} do b zkoušej: když $A[j] = 1$ a $j + a_i \leq b$, nastav $A[j + a_i] := 1$
4. Je-li $A[b] = 1$, podmnožina se součtem rovným b existuje.

Po i-tém průchodu hlavním cyklem obsahuje A jedničky právě u všech součtů neprázdných podmnožin $\{a_1, \ldots, a_i\}$. Důkaz – indukcí. Celk. složitost je $O(n \cdot b)$, což je exponenciální vzhledem k binárně kódovanému vstupu, ale polynomiální, máme-li na vstupu čísla konstantní délky.

Definice (Pseudopolynomiální algoritmus)

Nechť je dán rozhodovací problém Π a jeho instance I. Pak definujeme:

- kód(I) – délka zápisu (počet bitů) instance I v binárním kódování (či jiném na něj polynomiálně převoditelném)
- max(I) – velikost největšího čísla, vyskytujícího se v I (NE délka jeho binárního zápisu!)

Algoritmus se nazývá pseudopolynomiální, pokud je jeho časová složitost omezena polynomem v proměnných kód(I) a $max(I)$. Každý polynomiální algoritmus je tím pádem pseudopolynomiální.

Poznámka (O číselných problémech)

Pokud pro nějaký problém Π platí, že $\forall I : max(I) \leq p(\text{kód}(I))$ pro nějaký polynom p, pak všechny pseudopolynomiální algoritmy, řešící tento problém, jsou zároveň polynomiální.

Všechny problémy, kde tato rovnice neplatí(tj. neexistuje p, že by platila), nazýváme číselné problémy.

Věta (O pseudopolynomialitě a NP)

Nechť Π je NP-úplný problém a není číselný. Pak pokud P≠NP, nemůže být Π řešen pseudopolynomiálním algoritmem.

Poznámka

Ani ne každý číselný problém je řešitelný pseudopolynomiálním algoritmem.

Věta (O pseudopolynomialitě a podproblémech)

Nechť Π je rozhodovací problém a p polynom. Potom Π_p označme množinu instancí (podproblém) problému Π, pro které platí $max(I) \leq p(\text{kód}(I))$. Potom máme-li pseudopolynomiální algoritmus A, který řeší problém Π, určitě existuje polynomiální algoritmus, řešící Π_p. Toto platí pro libovolné p.

Důkaz

Algoritmus A', řešící Π_p v polynomiálním čase, otestuje x na přítomnost v Π_p (spočítá kód(x) a $max(x)$) a pokud $x \in \Pi_p$, chová se stejně jako A, takže běží v čase $q(\text{kód}(x), max(x)) \leq q(\text{kód}(x), p(\text{kód}(x))) = q'(\text{kód}(x))$.

Definice (Silně NP-úplný problém)

Rozhodovací problém Π je silně NP-úplný, pokud $\Pi \in$NP a existuje polynom p takový, že podproblém Π_p je NP-úplný.

Věta (O silné NP-úplnosti)

Nechť problém Π je silně NP-úplný. Potom, pokud P≠NP, neexistuje pseudopolynomiální algoritmus, který by řešil Π.

Důkaz

Plyne z předchozí věty.

Příklady

TSP je silně NP-úplný. Je to číselný problém, protože váhy hran nejsou omezené. Když váhy na hranách omezím, dostanu NP-úplný podproblém (jde na něj převést **HK**).

3-ROZDĚLENÍ je silně NP-úplné. Problém: máme $a_1, \dots a_{3m}, b \in \mathbb{N}$ takové, že $\forall j : \frac{1}{4}b \leq a_j \leq \frac{1}{2}b$ a navíc $\sum_{j=1}^{3m} a_j = mb$. Existuje $S_1, \dots S_m$ disjunktní rozdělení množiny $\{1, \dots, 3m\}$ takové, že $\forall i : \sum_{j \in S_i} a_j = b$?

Důkaz se provádí převodem z 3DM (třídimenzionální párování na tripartitních grafech), všechna čísla konstruovaná pro převod jsou polynomiálně velká vzhledem ke $|G|$ (v převodu $VP \propto SP$ byla exponenciálně velká).

Kapitola 5

Státnice - Aproximační algoritmy a schémata

5.1 Aproximační algoritmy

Definice (Aproximační algoritmus)

Aproximační algoritmus běží v polynomiálním čase a vrací řešení "blízká" optimu. Je nutné mít nějakou míru kvality řešení. Označme:

- C^* hodnotu optimálního řešení
- C hodnotu nalezenou aproximačním algoritmem

A předpokládejme nezáporné hodnoty řešení.

Definice (Poměrová chyba)

Řekneme, že algoritmus řeší problém s poměrovou chybou $\rho(n)$, pokud pro každé zadání velikosti n platí:

$$\max\{\frac{C^*}{C}, \frac{C}{C^*}\} \leq \rho(n)$$

Definice (Relativní chyba)

Řekneme, že algoritmus řeší problém s relativní chybou $\varepsilon(n)$, pokud pro každé zadání velikosti n platí:

$$\frac{|C - C^*|}{C^*} \leq \varepsilon(n)$$

Poznámka (O převodu chyb)

Z jedné chyby se dá vyjádřit druhá:

- V případě maximalizační úlohy: $\varepsilon(n) = \frac{C - C^*}{C^*} = \frac{C}{C^*} - 1 = \rho(n) - 1$
- V případě minimalizační úlohy: $\varepsilon(n) = \frac{C^* - C}{C^*} = 1 - \frac{1}{\rho(n)}$

Příklad (Aproximační algoritmy pro vrcholové pokrytí)

- "Brát vrcholy od nejvyššího stupně, dokud nemám celé pokrytí" nemá konstantní relativní chybu – ex. protipříklad, kdy $\rho(k) \leq a \cdot \ln k$
- "Vzít libovolnou hranu, dát do pokrytí její dva konce, odstranit její incidentní hrany a projít tak celé E" má relativní chybu 2 – žádné 2 hrany nemají společný vrchol, tj. mám pokrytí o velikosti $2 \times$ |mn. disj. hran|. Každé vrcholové pokrytí je ale $\geq$ |mn. disj. hran|.

Příklad (Aproximační algoritmy pro TSP)

Omezení na trojúhelníkovou nerovnost – pořád je NP (převod HK→TSP zachovával trojúhelníkovou nerovnost). Algoritmus:

1. Najdi minimální kostru T
2. Zvol lib. vrchol a pomocí DFS nad T v PRE-ORDERu očísluj vrcholy.
3. Cesta (s opakováním) po kostře T přes všechny vrcholy := X. Pak $w(X) = 2w(T)$ (každou hranou kostry jdu tam a zpět).
4. Výslednou HK H vyrobím zkrácením cesty (vypouštěním již navštívených vrcholů), z trojúhelníkové nerovnosti $w(H) \leq w(X)$. Celkem tedy dává $w(H) \leq w(X) \leq 2w(H^*)$, protože $w(T) \leq w(H^*)$ (H^* je kostra, bez jedné hrany).

Bez omezení – pro žádné konstantní ρ neexistuje polynomiální algoritmus, řešící obecný TSP s poměrovou chybou ρ. Mohu totiž mít HK v grafu $G = (V, E)$, pak zadání TSP zkonstruovat jako $K_{|V|}(V, V \times V)$, kde $w(e) = \begin{cases} 1 & e \in E \\ |V|\rho & e \notin E \end{cases}$. Pak by aproximační algoritmus s chybou ρ musel určitě vždy vrátit přesné řešení, takže by musel být NP-těžký.

5.2 Aproximační schémata

Definice (Aproximační schéma)

Aproximační schéma pro optimalizační úlohu je aproximační algoritmus, který má jako vstup instanci dané úlohy a číslo $\varepsilon > 0$, a který pro libovolné ε pracuje jako aproximační algoritmus s relativní chybou ε. Doba běhu může být exponenciální jak vzhledem k n – velikosti vstupní instance, tak vzhledem k $\frac{1}{\varepsilon}$.

Definice (Polynomiální aproximační schéma)

Polynomiální aproximační schéma je takové aproximační schéma, které pro každé pevné $\varepsilon > 0$ běží v polynomiálním čase vzhledem k n (ale stále může být exponenciální vzhledem k $\frac{1}{\varepsilon}$.

Definice (Úplně polynomiální aproximační schéma)

Úplně polynomiální aproximační schéma je polynomiální aprox. schéma, běžící také v polynomiálním čase vzhledem k $\frac{1}{\varepsilon}$ (tj. algoritmus s konstantně-krát menší relativní chybou běží v konstantně-krát delším čase).

Úplná polynomiálna aproximačná schéma pre problém batohu

Zadanie pre "dvojrozmernú variantu" problému batohu je n hmotností predmetov $w_1, w_2, \ldots w_n$, obmedzenie W na celkovú hmotnosť a hodnoty predmetov $v_1, v_2, \ldots, v_n$. Úlohou je nájsť takú množinu $S \subset \{1, 2, \ldots, n\}$, aby $\sum_{i \in S} w_i \leq W$ a $\sum_{i \in S} v_i$ bolo čo najväčšie.

Nech $V = \max\{v_1, v_2, \ldots, v_n\}$. Zadefinujme si $W(i, v)$ ako najmenšia hmotnosť takej podmnožiny predmetov $\{w_1, \ldots w_i\}$, ktorých celková hodnota je práve v. Potom:

$$W(i,v) = \begin{cases} 0 & v = 0 \\ \infty & i = 0, v > 0 \\ W(i-1, v) & v_i > v \\ \min\{W(i-1, v), w_i + W(i-1, v-v_i)\} & \text{inak} \end{cases}$$

V treťom prípade nepridáme vec i, lebo by sme prekročili cenu v (spomeňme si na definíciu $W(i, v)$), v štvrtom ju pridáme ak nám nepokazí hmotnosť.

Pomocou tohto môžeme napísať algoritmus využívajúci dynamické programovanie a ako výsledok vrátime $\max\{v | W(n, v) \leq W\}$. Tento algoritmus bude mať zložitosť n^2V, čo je iba pseudopolynomiálny algoritmus. Ten však môžeme upraviť.

Upravme si zadanie tak, že hodnoty všetkých predmetov orežeme o najnižšie bity. Ak chceme relatívnu chybu $\epsilon > 0$, tak orežeme $b = \lceil log \frac{\epsilon V}{n} \rceil$ bitov (nahradíme ich nulami) (prečo práve toľko bude vysvetlené nižšie). Tým sme získali novú inštanciu problému batohu, kde hmotnosti a limit sú rovnaké, ale pre každé i je hodnota predmetu $v_i' = 2^b \lfloor \frac{v_j}{2^b} \rfloor$. Náš algoritmus bude potrebovať čas $O(\frac{n^2 V}{2^b})$ (pretože ignorujeme nulové bity na konci každej hodnoty predmetu). Riešenie S', ktoré dostaneme bude možno rôzne od optima S pôvodnej úlohy, ale bude platiť:

$$\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i \geq \sum_{i \in S'} v_i' \geq \sum_{i \in S} v_i' \geq \sum_{i \in S} (v_i - 2^b) \geq \sum_{i \in S} v_i - n2^b$$

Prvá nerovnosť platí, lebo S je optimum v pôvodnej úlohe, druhá lebo $v'_i \leq v_i$, tretia lebo S' je optimum novej úlohy, štvrtá lebo $v'_i \geq v_i - 2^b$ a posledná lebo $|S| \leq n$. Naše riešenie je teda najviac $n2^b$ pod optimom. Dolný odhad optima je V (predpokladáme, že každý predmet sa samotný vmestí do batoha, ináč môžeme príliš ťažké predmety vyhodiť ako preprocessing). Relatívna chyba je teda $\frac{\text{approx}-\text{opt}}{\text{opt}} \leq \frac{\text{approx}-\text{opt}}{V} \leq \frac{n2^b}{V} = \epsilon$

Takže pre zadané ϵ orežeme $b = \lceil log\frac{\epsilon V}{n} \rceil$ bitov a dostaneme algoritmus, ktorého relatívna chyba je ϵ a jeho časová zložitosť je $O(\frac{n^2 V}{2^b}) = O(\frac{n^3}{\epsilon})$, čo je polynomiálne aj v n aj v $\frac{1}{\epsilon}$, preto je to úplná polynomiálna aproximačná schéma.

Kapitola 6

Státnice - Algoritmicky vyčíslitelné funkce

6.1 Částečně rekurzivní funkce

K. Gödel v 30. letech vynalezl primitivní rekurzivní funkce, později společně s dalšími částečně rekurzivní funkce. Jde o funkcionální přístup k algoritmům. Lze se na ně dívat i jako na logiku 1. řádu: základní funkce jsou axiomy, máme operátory – odvozovací pravidla – a z toho vyrábíme formule – rekurzivní funkce.

Definice (Podmíněná rovnost, konvergence, divergence)

- $\simeq$ značí "podmíněnou rovnost", tj. v případě, že alespoň jedna strana má smysl, tak má smysl i druhá a rovnají se.
- $P_1(D)\downarrow$ značí, že predikát je definován, tj "konverguje" (občas se značí ! místo $\downarrow$)
- $P_1(D)\uparrow$ značí, že predikát není definován, tj. "diverguje"

Značky konvergence, divergence i podmíněné rovnosti se vztahují jak na predikáty, tak na funkce.

Definice (Základní funkce)

- $o(x) \simeq 0 \quad \forall x \in \mathbb{N}$ ("nula")
- $s(x) \simeq x + 1 \quad \forall x \in \mathbb{N}$ ("následník")
- $I_n^j(x_1, \ldots, x_n) \simeq x_j \quad 1 \leq j \leq n$ ("projekce", vybrání jedne ze složek)

Definice (Základní operátory)

- $R_n \quad (n \geq 1)$ – primitivní rekurze
  Funkcím f ($n-1$ proměnných) a g ($n+1$ proměnných) přiřadí $R_n(f,g) = h$, kde $h(0, x_2, \ldots, x_n) \simeq f(x_2, \ldots, x_n)$ a $h(y+1, x_2, \ldots, x_n) \simeq g(y, h(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$ (analogické k for-cyklu).
- S_n^m – substituce
  Funkci f (m proměnných) a m funkcím g_i (všechny n proměnných) přiřadí funkci h (n proměnných) předpisem $h = S_n^m(f, g_1, \ldots, g_m) \equiv h(x_1, \ldots, x_n) \simeq f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ (analogické k podprogramu).
- M_n – minimalizace
  Funkci f ($n+1$ proměnných) přiřadí h (n proměnných) tak, že

  $$h(x_1, \ldots, x_n)\downarrow \wedge h(x_1, \ldots, x_n) \simeq y \equiv f(x_1, \ldots, x_n, y)\downarrow, \simeq 0 \wedge f(x_1, \ldots, x_n, j)\downarrow, \not\simeq 0 \ \forall j < y$$

  (analogické k while-cyklu).

Další značení:

- $\mu_y P(x,y)$ je funkce proměnné x, která vrátí nejmenší y takové, aby platil predikát $P(x,y)$. Lze jí sestrojit pomocí operátoru minimalizace.

Definice (Třída primitivně a částečně rekurzivních funkcí)

- Třída primitivně rekurzivních funkcí je nejmenší třída funkcí $f : \mathbb{N}^k \to \mathbb{N}$, která obsahuje základní funkce a je uzavřená na R_n a S_n^m.
- Třída částečně rekurzivních funkcí je nejmenší třída, která obsahuje zákl. funkce a je uzavřená na R_n, S_n^m a M_n.

Poznámka (Vlastnosti zákl. funkcí a operátorů)

- Všechny zákl. funkce jsou všude definované ("totální") a efektivně vyčíslitelné.
- Všechny zákl. operátory zachovávají efektivní vyčíslitelnost.
- R_n, S_n^m zachovávají totálnost.
- PRF jsou efektivně vyčíslitelné a totální.

Definice (Odvození funkce)

Odvození funkce je konečná posloupnost funkcí, z nichž každá je buď funkce základní, nebo vzniká z už odvozených funkcí pomocí nějakého operátoru. Ke každé funkci si pamatujeme, jak vznikla (toto v praxi hraje roli programu).

Definice (Obecně rekurzivní funkce)

Funkce je obecně rekurzivní (ORF), jestliže je ČRF a totální.

Operace s PRF, predikáty

Poznámka (Některé PRF)

Pomocí PRF lze popsat např.:

- součet
- součin
- mocninu, faktoriál
- operaci $x \dot{-} y$, kde $x \dot{-} y = x - y$ pro $x \geq y$, jinak 0
- operátory sg a $\overline{\text{sg}}$ (testy na nenulovost, resp. nulovost argumentu)
- minimum, maximum, absolutní hodnotu rozdílu

Definice (Charakteristická funkce)

Mějme predikát P (libovolné tvrzení) o n proměnných. Potom c_P je jeho charakteristická funkce, když je to všude definovaná funkce daná následovně:

$$c_P(x_1, \ldots, x_n) \simeq \begin{cases} 1 & \text{pokud } P(x_1, \ldots, x_n) \\ 0 & \text{jinak} \end{cases}$$

Částečná charakteristická funkce pro nějaký predikát P o n proměnných je funkce f o n proměnných taková, že $f(x_1, \ldots, x_n) \downarrow \Leftrightarrow P(x_1, \ldots, x_n)$ a $f(x_1, \ldots, x_n) \downarrow \Rightarrow f(x_1, \ldots, x_n) = 1$.

Definice (PR, OR, RS Predikáty)

Řekneme, že predikát je primitivně (obecně) rekurzivní, jestliže jeho charakteristická funkce je primitivně (obecně) rekurzivní. Predikát je rekurzivně spočetný, jestliže jeho částečná charakteristická funkce je částečně rekurzivní.

S funkcemi a predikáty se operuje docela nedůsledně, dají se v podstatě ztotožnit.

Poznámka (Jiná možnost nahlížení)

ČRF odpovídají funkcionální logice 1. řádu:

- termy číselné: $0, x, x+1, \ldots$
- termy funkční: $o, I_1^1, s, R_2(I_1^1, S_3^1(s, I_3^2)), \ldots$
- pravidlo aplikace: $Ap(f, x) = \cdots = f(x)$ (kde "..." je proces vyhodnocení termu, potenciálně nekonečný, dává z funkce číselný term)
- pravidlo zobecnění: $\lambda xy(x+y)$ dává z číselného termu $x+y$ funkci

Poznámka (Operace zachovávající PR)

PR jsou:

- Rozšíření počtu proměnných, konstantní funkce
- Permutace a ztotožnění proměnných
- Kódování $\mathbb{N}^k$ do $\mathbb{N}$ – iterace Cantorova diagonálního kódování dvojic ($\langle x, y\rangle_2 = \frac{(x+y)(x+y+1)}{2} + x$)
- Opačná operace – dekódování
- Funkce $p(i)$ – i-té prvočíslo
- Predikát rovnosti a $<, >$
- Logické spojky $\vee, \wedge, \neg$, omezené kvantifikátory (kvantifikace spočetně mnoha prvků)
- Gödelovo prvočíselné kódování: slovo $a_{i_0} \dots a_{i_k}$ do $p(0)^{i_0} \cdot \dots \cdot p(k)^{i_k}$

Ackermannova funkce

Definice (Ackermannova funkce)

Ackermannova funkce je funkce definovaná jako:

$$A(0,x) = \begin{cases} 1 & x = 0 \\ 2 & x = 1 \\ x+2 & x > 1 \end{cases}$$

$$A(y,0) = 1$$

$$A(y+1, x+1) = A(y, A(y+1, x))$$

Definice (Strukturální složitost)

Definujeme strukturální složitost – hloubku rekurze (intuitivně: počet vnořených for-cyklů – syntakticky, ne výpočtem) jako 0 pro základní funkce a

$$h(R_n(P,Q)) = \max(h(P), h(Q)+1),\ h(S_n^m(P, Q_0, \dots, Q_k)) = \max(h(P), h(Q_0), \dots, h(Q_k))$$

Pak $\mathcal{R}_i$ je třída PRF, které lze získat pomocí PR-termů hloubky $\leq i$ a PRF samo je $\cup_{i=1}^{\infty} \mathcal{R}_i$

Věta (O Ackermannově funkci)

Ackermannova funkce není PRF, ale je ORF.

Důkaz

- Určitě je ORF – důkaz se provádí transfinitní indukcí typu ω^2; pro výpočet každé hodnoty potřebuji jen konečně mnoho předchozích hodnot – stačí mi μ_z, kde z je nejmenší kus $\mathbb{N}^2$, který stačí k výpočtu $A(y,x)$ (dá práci dokázat, že je konečný, potřeba ordinálů, lexikografického uspořádání).
- A roste rychleji než každá PRF: $\forall \varphi$ PRF (jedné proměnné) $\exists x_0 : \forall x \geq x_0 : \varphi(x) < A(x,x)$.
- Uvažujme $A(y,x)$ jako matici funkcí $f_y(x)$. Potom určitě $f_i \in \mathcal{R}_i \backslash \mathcal{R}_{i-1}$ a $f_y(x)$ je (až na konečně mnoho x) rostoucí. Navíc pro libovolnou $\varphi \in \mathcal{R}_i$ existuje x_0 takové, že $\forall x \geq x_0 : \varphi(x) < f_{i+1}(x)$, tedy f_{i+1} majorizuje všechny funkce z $\mathcal{R}_i$
- Nechť pro spor má $A(x,x)$ hloubku i. Potom $A(x,x) = \varphi(x) < f_{i+1}(x)$ pro nějaké pevné i. Potom ale $A(x,x) < f_{i+1}(x) < f_x(x) = A(x,x)$, tj. pro $x > i+1$ máme spor.

Věta (O vztahu PRF, ORF a ČRF)

Platí PRF$\subset$ ORF $\subset$ ČRF a inkluze jsou ostré.

Důkaz

Pro ORF$\subset$ ČRF mám funkci $g(x,y) \simeq y+1$ a $h(x) \simeq \mu_y(g(x,y) \simeq 0)$, ta není nikde definovaná. Pro PRF$\subset$ ORF mám Ackermannovu funkci.

6.2 Univerzální funkce

Definice (Univerzální funkce)

Mějme $\mathcal{T}$ – spočetnou množinu ČRF jedné proměnné. Potom $\mathcal{U}(i,x)$ je univerzální funkce třídy $\mathcal{T}$, jestliže:

- $\forall$ přirozené $i : \lambda x\mathcal{U}(i,x) \in \mathcal{T}$
- $\forall \varphi \in \mathcal{T} : \exists i_0 : \varphi = \lambda x\mathcal{U}(i_0,x)$

A $\mathcal{U}$ tedy indexuje všechny funkce třídy $\mathcal{T}$. Podobně se definují i univerzální funkce pro ČRF více proměnných. Platí, že $\{\lambda x\mathcal{U}(i,x)\}_{y\geq 0}$ je posloupnost všech funkcí z $\mathcal{T}$, takže $\mathcal{U}$ určuje numeraci prvků $\mathcal{T}$

Věta (O univerzální funkci PRF)

Existuje ORF, která je univerzální pro třídu PRF (jedné proměnné). Taková funkce pak nemůže být PRF.

Důkaz

- Seřadím všechny PR-termy (PR-programy) do posloupnosti (máme 3 axiomy a 3 odvozovací pravidla, seřazení je možné).
- Potom $U(x,y) := h_x(y)$, kde h_x vyčísluje x-tý program.
- Sporem nechť $U(x,y)$ je PRF. Pak i $U(x,x)$ je PRF, $1\dot{-}U(x,x)$ je PRF, z toho $1\dot{-}U(x,x) = U(x_0,x)$. Dosadím $x = x_0$ a mám spor, neboť obě strany jsou definovány (toto je příklad použití Cantorovy diagonální metody).

Definice (Turingovsky vyčíslitelná funkce)

Vezmeme Turingovy stroje s vnější abecedou, jejíž prvním znakem je "|". Čísla $0, 1, \ldots$ zapisujme na pásku jako $|, ||, |||, \ldots$, n-tice oddělujme znakem λ. Potom:

- Řekneme, že stroj M je n-aritmetický, pokud pro každou n-tici přir. čísel $x_1, \ldots, x_n$ reprezentovanou počáteční konfigurací S platí: je-li M použitelný k S (zastaví-li se výpočet nad ní) a je-li výsledná konfigurace T, pak v T je na pásce nějaké jedno přirozené číslo a hlava stroje M stojí nad jeho posledním znakem |.
- Stroj je dále n-aritmetický typu 0/1, pokud má abecedu $\{|, \lambda\}$ a jediný koncový stav.
- Řekneme, že M vyčísluje funkci f o n proměnných, pokud M je n-aritmetický a pro každou n-tici přir. čísel $x_1, \ldots, x_n$ v poč. konfiguraci S platí: M je použitelný, právě když je f pro $x_1, \ldots, x_n$ definovaná a je-li f definovaná, pak ve výsledné konfiguraci T je na pásce stroje číslo $f(x_1, \ldots, x_n)$ a hlava stojí nad jeho posledním znakem.
- Řekneme, že funkce je turingovsky vyčíslitelná, pokud existuje nějaký n-aritmetický TS (typu 0/1), který ji vyčísluje.

Věta (O ekvivalenci TS a ČRF)

Funkce n proměnných je částečně rekurzivní, právě když existuje n-aritmetický Turingův stroj typu 0/1, který ji vyčísluje.

Důkaz

"$\Leftarrow$": Každá ČRF je T-vyčíslitelná.

Důkaz indukcí podle složitosti funkce – pro základní funkce to jistě platí, R_n, M_n, S_n^m toto zachovávají (S_n^m znamená použití více pásek, vyčíslení a složení, R_n znamená vyčíslení f a y-krát "otočení" g, M_n je vyčíslování f na vstupu a zvětšujícím se počítadle cyklů, dokud nedostanu 0 – pak vrátím hodnotu počítadla).

"$\Rightarrow$": Pro každý TS M existuje ČRF, která dává stejný výsledek

Je nutné zavést kódy konfigurací; TS navíc nemají žádné podtřídy, tedy nelze postupovat induktivně. Platí:

- $\text{step}_M(X)$ – jeden krok stroje je PR záležitost (pracuje se nad konfiguracemi TS $UqsV$, z obou stran obalenými spec. znakem h, pak slovo není nekonečné a lok. změna se dá spočítat). Existuje určitě PR funkce, která popisuje lokální změnu (jde vlastně o rozhodovací strom, který se staví pomocí R_n).
- $\text{comp}_M(X,i)$ – výsledek stroje po i krocích práce je stále PR (for-cyklus – R_n)
- $\mu_i(\text{comp}_M(X,i)$ obsahuje $q_0)$ – q_0 je koncový stav (pracuj, dokud neskončíš – while)
- Potom výsledná ČRF g je dána jako $g(\text{kód}(S)) \simeq \text{result}(\mu_i(\text{comp}_M(X,i)$ obsahuje $q_0))$, kde result je jednoduchá funkce smazání okrajů atp. BÚNO je takový stroj úplný a q_0 jeho jediný koncový stav. Operátor minimalizace se vyskytuje jen jednou, proto je vhodné ho vysunout co nejvíce "ven" v uzávorkování.

Pak také platí, že mám-li nějakou částečnou funkci (tj. nemusí být totální), která je turingovsky vyčíslitelná, pak je ČR.

Kleenova věta

Věta (Kleenova o normální formě)

Pro každé $k \geq 1$ existují

- ČRF Ψ_k $k+1$ proměnných
- PRP T_k $k+2$ proměnných (Kleeneův predikát)
- PRF U jedné proměnné
- PRF s_k $k+1$ proměnných

takové, že:

1. Ψ_k je univerzální funkcí pro třídu všech ČRF k proměnných. $\Psi_k(e, x_1, \ldots, x_k)$ vyčísluje e-tou ČRF k proměnných. Navíc z odvození ČRF lze efektivně získat e a naopak z e lze efektivně získat odvození příslušné ČRF.
2. $\Psi_k(e, x_1, \ldots, x_k) \simeq U(\mu_y T_k(e, x_1, \ldots, x_k, y))$, kde T_k odpovídá výpočtu Turingova stroje, $y = \langle y_0, y_1 \rangle$, y_0 je doba výpočtu, y_1 výsledek a U vydělí z $\langle y_0, y_1 \rangle$ druhou složku.
3. s_k je prostá funkce rostoucí ve všech proměnných, o které platí (tato část Věty o normální formě se nazývá S-m-n věta):
   $\Psi_{m+n}(e, z_1, \ldots, z_m, x_1, \ldots, x_n) \simeq \Psi_n(s_m(e, z_1, \ldots, z_m), x_1, \ldots, x_n)$
   $T_{m+n}(e, \vec{z}, \vec{x}) \equiv T_n(s_m(e, \vec{z}), \vec{x})$
4. $T_k(e, x_1, \ldots, x_k, y) \wedge T_k(e, x_1, \ldots, x_k, z) \Rightarrow y = z$

Díky tomu lze ČRF efektivně očíslovat. $\varphi_e(x_1, \ldots, x_k)$ pak značí e-tou funkci k proměnných. Indexu e se říká Gödelovo číslo funkce.

Důkaz

- Oklikou přes Univerzální Turingův stroj: ke každé ČRF máme TS a jeho kód e. Vezmeme si proto UTS, který s kódy umí počítat, a hledáme jeho ČRF.
- Páska univerzálního stroje vypadá v obecném případě následovně:

  $$Y \text{ blok1 } Y \text{ blok2 } \Delta \text{ blok3 } \times O_1 \times O_2 \ldots Y$$

  První blok je aktuální konfigurace, druhý číslo stavu a třetí aktuální políčko, zbytek je program. Čísla kódujeme unárně (x jako $x+1$ čar).
- Základní idea – bez proměnných $x_1, \ldots x_k$ páska UTS vypadá takto: $Y\ M\ Y$ blok2 Δ blok3 $\times O_1 \times O_2 \ldots Y$ (M je kód programu).
- Konstrukce $\Psi_m(e, x_1, \ldots, x_m)$:
  - Zkontrolujeme, zda e po rozkódování obsahuje nějaký kód programu M.
  - Jestliže ne, je výsledkem nulová funkce (syntax error).
  - Jestliže ano, nejlevější výskyt M nahradíme $||\ldots|\lambda||\ldots|\lambda\ldots\lambda||\ldots|M$ (kódování vstupních dat $x_1, \ldots, x_n$; substituce) a spustíme program e na UTS, podle toho získáme výsledek – Ψ_k
- $s_k(e, y_1, \ldots, y_k)$ odpovídá: čekej na $x_1, \ldots, x_j$, přidej k nim $y_1, \ldots, y_k$ a spusť program e.

Věta (Vlastnosti predikátu Ψ_k)

1. Predikát $\Psi_k(e, x_1, \ldots, x_k)\downarrow$ je rekurzivně spočetný, není rekurzivní.
2. jeho negace $\Psi_k(e, x_1, \ldots, x_k)\uparrow$ není rekurzivně spočetná.
3. Dále Ψ_k nelze rozšířit do ORF. Dokonce pokud α je ČRF, která je rozšířením Ψ_k, potom lze efektivně nalézt vstup $\vec{z}$ takový, že $\alpha(\vec{z})\uparrow$.

Univerzální funkce pro danou třídu funkcí tedy buď nemůže patřit do této třídy, nebo nemůže být totální.

Důkaz

- Z definice je zřejmé, že $\Psi_k(\ldots)\downarrow$ je rekurzivně spočetný predikát. Stačí ukázat, že $\Psi_k(\ldots)\uparrow$ není rekurzivně spočetný. Z toho přímo plyne, že $\Psi_k(\ldots)\downarrow$ není rekurzivní.
- Bez újmy na obecnosti uvažujme k=1. Použijeme Cantorovu diagonální metodu.
- Kdyby $\Psi_1(\ldots)\downarrow$ byl rekurzivní, potom by $\Psi_1(x,x)\uparrow$ byl také rekurzivní, tím spíše rekurzivně spočetný. Tedy pro nějakou ČRF φ by platilo $\Psi_1(x,x)\uparrow\Leftrightarrow \varphi(x)\downarrow$. Vezmeme-li index funkce φ (označme jej x_0), dostáváme $\Psi_1(x,x)\uparrow\Leftrightarrow \Psi_1(x_0,x)\downarrow$, po dosazení $x = x_0$ dostáváme $\Psi_1(x_0,x_0)\uparrow\Leftrightarrow \Psi_1(x_0,x_0)\downarrow$, což je spor.
- Pro důkaz zbytku tvrzení předpokládejme, že $h(e,x)$ je ORF rozšířením $\Psi_1(e,x)$. Potom $1\dot{-}h(x,x)$ je ORF g. Nechť g má index x_0, tj. $g(x) \simeq \Psi_1(x_0,x)$. Protože g je ORF, pro všechna x platí $\Psi_1(x_0,x)\downarrow$, tedy $\Psi_1(x_0,x_0)\downarrow$. Dostáváme $h(x_0,x_0) = \Psi_1(x_0,x_0)$, což ovšem vede ke sporu: $1\dot{-}\Psi_1(x_0,x_0) \simeq h(x_0,x_0) \simeq \Psi_1(x_0,x_0)$.
- Pokud nějaká ČRF β je rozšířením Ψ_1, umím pro β (podle předch. důkazu) najít e takové, že $\beta(e,e)\uparrow$.
- Myšlenka obsažená v předchozím důkazu je založená na Cantorově diagonální metodě. Spor na diagonále si vynutí divergenci, neboť rovnost funkcí je jenom podmíněná, tedy v případě divergence je vše v pořádku.

Kapitola 7

Státnice - Rekurzivní a rekurzivně spočetné množiny

7.1 Rekurzivně spočetné množiny

Definice (Rekurzivní a rekurzivně spočetná množina)

Charakteristická funkce množiny M označuje charakteristickou funkci predikátu náležení do množiny, tj. funkci $c_M(x)$, kde $c_M(x) =\downarrow 1$ pro $x \in M$ a $c_M(x) \downarrow= 0$ pro $x \notin M$.

Analogicky se definuje částečná charakteristická funkce množiny – $c_M(x) =\downarrow 1$ pro $x \in M$ a $c_M(x) \uparrow$ pro $x \notin M$.

Množina M je rekurzivní, je-li její charakteristická funkce obecně rekurzivní (každá char. fce je totální, takže ČRF by bylo totéž). Množina M je rekurzivně spočetná, jestliže je definičním oborem nějaké ČRF (neboli jestliže je její částečná char. funkce částečně rekurzivní).

Množina je rekurzivní, jestliže existuje program, který se na libovolném vstupu zastaví a rozhodne, zda do ní vstup patří. Množina je rekurzivně spočetná, jestliže existuje program, který se zastaví právě na jejích prvcích. Je-li množina rekurzivní, je i rekurzivně spočetná, opačně to neplatí.

Definice (dom, rng)

V následujícím dom značí definiční obor, rng obor hodnot.

Definice (x-tá rekurzivně spočetná množina)

$$W_x \ (x\text{-tá rekurzivně spočetná množina}) = dom(\varphi_x) = \{y : \varphi_x(y)\downarrow\}$$

Definice (K)

$$K = \{x : x \in W_x\} = \{x : \varphi_x(x)\downarrow\} = \{x : \Psi_1(x, x)\downarrow\}$$

Množina K vlastně odpovídá halting problému. Platí o ní následující tvrzení.

Věta (Rekurzivní spočetnost K)

Množina K je rekurzivně spočetná, není rekurzivní, $\overline{K}$ není rekurzivně spočetná.

Důkaz

K není rekurzivní, neboť $\overline{K}$ není rekurzivně spočetná. $\overline{K}$ není rekurzivně spočetná, neboť kdyby byla, měla by index x_0. Jednoduchou diagonalizací dostáváme $x_0 \in \overline{K} \Leftrightarrow x_0 \in W_{x_0} \Leftrightarrow x_0 \in K$. Spor.

7.2 1-převeditelnost, m-převeditelnost

Definice (1-převeditelnost, m-převeditelnost, 1-úplnost, m-úplnost)

- Množina A je 1-převedilná na B (značíme $A \leq_1 B$), jestliže existuje prostá ORF f taková, že $x \in A \Leftrightarrow f(x) \in B$.
- Množina A je m-převedilná na B (značíme $A \leq_m B$), jestliže existuje ORF f (ne nutně prostá) taková, že $x \in A \Leftrightarrow f(x) \in B$.
- Množina M je 1-úplná, jestliže M je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni 1-převedilná.

- Množina M je m-úplná, jestliže M je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni m-převedilná.

Věta (1-úplnost K)

K je 1-úplná. Tedy halting problem je vzhledem k 1 a m-převoditelnosti nejtěžší mezi rekurzivně spočetnými problémy.

Důkaz

Mějme libovolnou rekurzivně spočetnou množinu W_x.

Mějme ČRF $\alpha(y,x,w)$, popisující x-tou rekurzivně spočetnou množinu. Tedy $\alpha(y,x,w)\downarrow \Leftrightarrow y \in W_x \Leftrightarrow \Psi_1(x,y)\downarrow \Leftrightarrow \varphi_x(y)\downarrow$. w je tady fiktivní proměnná, funkce α na její hodnotě nezáleží. Z s-m-n věty dostáváme: $\alpha(y,x,w) \simeq \Psi_3(a,y,x,w) \simeq \Psi_1(s_2(a,y,x),w) \simeq \varphi_{s_2(a,y,x)}(w)$. Označme $h(y,x) = s_2(a,y,x)$ (s_2 je PRF, tím spíše ORF). $y \in W_x \Leftrightarrow \alpha(y,x,w)\downarrow \Leftrightarrow \varphi_{h(y,x)}(w)\downarrow \Leftrightarrow \varphi_{h(y,x)}(h(y,x))\downarrow \Leftrightarrow h(y,x) \in K$ Zde jsme mohli za w dosadit $h(y,x)$, neboť hodnota α na w nezáleží! Tedy $W_x \leq_1 K$ pomocí funkce $\lambda y : h(y,x)$.

Lemma (K_0 je 1-úplná)

$K_0 = \{\langle y,x\rangle : y \in W_x\}$ je 1-úplná.

Důkaz

Zřejmé. $K \leq_1 K_0$ a K je 1-úplná.

Lemma (Poznámky k 1-převeditelnosti)

1. Relace $\leq_1$ a $\leq_m$ jsou tranzitivní, reflexivní.
2. $A \leq_1 B \Rightarrow A \leq_m B$
3. B rekurzivní, $A \leq_m B \Rightarrow A$ rekurzivní.
4. B rekurzivně spočetná, $A \leq_m B \Rightarrow A$ rekurzivně spočetná.

Důkaz

1. Zřejmé.
2. Zřejmé.
3. Složením funkce dokazující $\leq_m$ s procedurou, která rozhoduje o $x \in B$, dostaneme proceduru rozhodující o $x \in A$. Dostáváme $c_A(x) = c_B(f(x))$.
4. Stejně.

Důsledek

K a $\overline{K}$ jsou m-nesrovnatelné.

Důkaz

Plyne z faktu, že K je rekurzivně spočetná, $\overline{K}$ není, a z bodu 4 předchozího lemma.

Definice (Rekurzivní permutace)

Permutace na $\mathbb{N}$, která je ORF, se nazývá rekurzivní permutace.

Definice (Rekurzivní isomorfismus)

Množiny A a B jsou rekurzivně isomorfní, jestliže existuje rekurzivní permutace p taková, že $p(A) = B$. Značíme $A \equiv B$.

Definice (1-ekvivalence a m-ekvivalence)

- $A \equiv_1 B$, jestliže $A \leq_1 B \wedge B \leq_1 A$.
- $A \equiv_m B$, jestliže $A \leq_m B \wedge B \leq_m A$.

Věta (Myhillova)

$$A \equiv B \Leftrightarrow A \equiv_1 B$$

Důkaz

Jedná se o vlastně o obdobu Cantor-Bernsteinovy věty.

$\Rightarrow$ Triviální.

$\Leftarrow$ Z předpokladů máme dvě prosté ORF f, g převádějící vzájemně A na B a opačně. Chceme sestrojit rekurzivní permutaci h takovou, že $h(A) = B$.

Plán: v krocích budeme generovat graf h tak, že v kroku n bude platit $\{0, \ldots, n\} \subseteq dom(h), \{0, \ldots, n\} \subseteq rng(h)$.

Z toho plyne, že h bude definovaná na celém $\mathbb{N}$ a bude na. Současně zajistíme, že h bude prostá.

Navíc budeme chtít, aby platilo $y \in A \Leftrightarrow h(y) \in B$, tedy aby h převáděla A na B.

Začneme v bodě 0 a položíme $h(0) = f(0)$. Rozlišíme následující případy:

1. $f(0) = 0$: vše je v pořádku, $h(0) = f(0) = 0$ a $0 \in A \Leftrightarrow 0 \in B$, pokračujeme dalším prvkem.

2. $f(0) \neq 0$: rozlišíme dva podpřípady

   (a) $g(0) \neq 0$: definujeme $h(g(0)) = 0$.
   Tedy $0 \in dom(h) \cap rng(h)$.

   (b) $g(0) = 0$: nemůžeme použít $h(g(0)) = 0$, protože v bodě 0 je již h definována. Najdeme tedy volný bod: definujme $h(g(f(0))) = 0$. Určitě $g(f(0)) \neq 0$, protože g je prostá a $f(0) \neq 0$. Tímto jsme opět dostali bod 0 do definičního oboru h i oboru hodnot. Zároveň funkci h definujeme podle f a g, tedy převádí vzájemně A na B.

Indukční krok: nechť v kroku k je z první volný prvek. Všechna čísla menší než z máme v $dom(h) \cap rng(h)$. Podíváme se, zda je $f(z)$ volný. Jestliže ano, položíme $h(z) = f(z)$. Jestliže $f(z)$ není volný, hledám "cik-cak" další volný (podobně jako pro 0, maximálně z prvků je blokovaných, tj. maximálně po z iteracích tohoto postupu dojdu k volnému prvku).

Důsledek

$K \equiv K_0$.

Důkaz

Zřejmé, neboť $K \equiv_1 K_0$ (obě množiny jsou 1-úplné).

7.3 Rekurzivně spočetné predikáty

Lemma (ORF $\rightarrow$ RSP)

Je-li Q obecně rekurzivní predikát, potom $\exists y : Q$ je rekurzivně spočetný predikát.

Důkaz

$\mu_y Q$ je ČRF, její definiční obor je $\{\exists y : Q\}$.

Věta (Univerzální RSP)

Predikát $\exists y T_k(e, x_1, \ldots, x_k, y)$ je univerzálním RSP pro třídu RSP k proměnných, tj. lze definovat index (Gödelovo číslo) rekurzivně spočetného predikátu.

Důkaz

Z věty o normální formě – numerace ČRF nám dává numeraci predikátů.

Věta (Log. spojky a rek. spočetnost)

Konjukce a disjunkce zachovávají rekurzivní spočetnost. Tedy průnik a sjednocení rekurzivně spočetných množin je rekurzivně spočetná množina. Stejně pro predikáty.

Důkaz

Pro průnik spustíme oba programy současně a čekáme, až se oba zastaví. Pro sjednocení čekáme, až se zastaví alespoň jeden.

Formálně pro průnik ($(w)_{2,1}$ znamená to, že w kóduje usp. dvojici a vybíráme z ní první prvek; to je PRF): $\exists s_1 T_k(a,\vec{x},w_1) \wedge \exists s_2 T_k(b,\vec{x},w_2) \Leftrightarrow \exists w(T_k(a,\vec{x},(w)_{2,1}) \wedge T_k(b,\vec{x},(w)_{2,2}))$. Uvedený predikát je rekurzivně spočetný, tedy má nějaký index, tj. ekvivalence pokračuje: $\exists w T_{k+2}(e,a,b,\vec{x},w) \Leftrightarrow \exists w T_k(s_2(e,a,b),\vec{x},w)$

Poznámka

Konjunkce a disjunkce tedy rek. spočetnost zachovávají, o negaci (tj. doplňku) to ale už samozřejmě neplatí.

Věta (Kvantifikace a rek. spočetnost)

Omezená kvantifikace $(\forall y)_{y\le t}$ a existenční kvantifikace (pro $k \ge 2$) zachovávají rekurzivní spočetnost.

Důkaz

Neformálně: omezený kvantifikátor lze zkontrolovat for cyklem.

Formálně: $(\forall y)_{y\le t}\ \exists s : T_k(e,x_1,\ldots,x_{k-1},y,s) \Leftrightarrow \exists$ kód $(t+1)$-tice w : $(\forall y)_{y\le t}\ T_k(e,x_1,\ldots,x_{k-1},y,(w)_{t+1,y})$.

y můžeme zkoušet primitivní rekurzí, w minimalizací, dostáváme tedy rekurzivně spočetný predikát, který má nějaký index b, dále můžeme použít S-m-n větu. $\exists s : T_{k+1}(b,e,x_1,\ldots,x_{k-1},t,s) \Leftrightarrow \exists s : T_k(s_1(b,e),x_1,\ldots,x_{k-1},t,s)$.

Pro existenční kvantifikátor je situace ještě jednodušší. Kvantifikaci přes dvě proměnné převedeme na kvantifikaci přes jednu, kterou budeme považovat za kód dvojice a v predikátu potom vydělíme jednotlivé složky (a použijeme opět S-m-n větu). Dostáváme predikát $k-1$ proměnných, proto je ve větě požadavek na minimální velikost $k \ge 2$.

$$\exists y : \exists s : T_k(e,x_1,\ldots,x_{k-1},y,s) \Leftrightarrow \exists w : T_k(e,x_1,\ldots,x_{k-1},(w)_{2,1},(w)_{2,2})$$

$$\Leftrightarrow \exists s : T_k(b,e,x_1,\ldots,x_{k-1},s) \Leftrightarrow \exists s : T_{k-1}(s_1(b,e),x_1,\ldots,x_{k-1},s)$$

Poznámka

Neomezená obecná kvantifikace ($\forall$) rekurzivní spočetnost nezachovává.

Věta (O selektoru)

Nechť Q je RSP $k+1$ proměnných. Potom existuje ČRF φ k proměnných taková, že:

$$\varphi(x_1,\ldots,x_k)\downarrow \Leftrightarrow \exists y : Q(x_1,\ldots,x_k,y)$$

$$\varphi(x_1,\ldots,x_k)\downarrow \Rightarrow Q(x_1,\ldots,x_k,\varphi(x_1,\ldots,x_k))$$

Věta říká, že pro každý rekurzivně spočetný predikát existuje ČRF taková, že konverguje, právě když existuje y splňující predikát. Tato funkce navíc přímo vrací jedno takové y, pro které predikát platí. Tato φ je selektor na grafu Q.

Důkaz

Dáno $\vec{x}$, hledáme nejmenší dvojici (y,s) takovou, že za s kroků ověříme, že $Q(\vec{x},y)$ (tj. program pro Q konverguje za s kroků). Pak vydáme y.

Obecně: univerzální vyjádření RSP $\exists s : T_{k+1}(e,\vec{x},y,s)$, hledáme nejmenší w (kód dvojice) takové, že $\varphi(\vec{x}) \simeq (\mu_w T_{k+1}(e,\vec{x},(w)_{2,1},(w)_{2,2}))_{2,1}$. Funkce φ vrací první složku z první dvojice, kterou najde (v uspořádání daném naším kódováním dvojic).

Věta (Vztah ČRF a RS grafů)

Funkce je ČRF $\Leftrightarrow$ má rekurzivně spočetný graf.

Důkaz

Je-li φ ČRF, je její graf rekurzivně spočetný: $\langle x_1,\ldots,x_k,y\rangle \in$ Graf $\Leftrightarrow \exists s$: za s kroků program konverguje.

Opačně, je-li graf funkce φ rekurzivně spočetný, je selektor na něm ČRF, ale selektor na grafu funkce je přímo ona funkce.

Věta (Postova)

Množina M je rekurzivní, právě když M i $\overline{M}$ jsou rekurzivně spočetné.

Predikát Q je ORP, právě když Q i $\neg Q$ jsou RSP.

Důkaz

"$\Rightarrow$": Triviální.

"$\Leftarrow$": Intuitivně: $M = dom(P_1)$, $\overline{M} = dom(P_2)$. Pustíme oba programy současně a čekáme, který se zastaví. Zastaví se právě jeden.

Formálně: $(x \in M \wedge y = 1) \vee (x \in \overline{M} \wedge y = 0)$ je rekurzivně spočetný predikát, selektor na něm je ORF, která je charakteristickou funkcí pro M.

7.4 Generování rekurzivně spočetných množin

Lemma (Rek. spočetná množina je obor hodnot ČRF)

Každá rekurzivně spočetná množina je oborem hodnot nějaké ČRF.

Důkaz

Pro každou množinu W_x vytvoříme množinu dvojic $R = \{\langle y, y\rangle : y \in W_x\}$. Množina R je rekurzivně spočetná, tedy má ČRF selektor φ, platí $dom(\varphi) = rng(\varphi) = W_x$.

Myšlenka toho důkazu je, že body, kde φ_x konverguje, vyneseme na diagonálu a vytvoříme selektor. Jeho definiční obor bude zároveň jeho oborem hodnot.

Věta (ČRF odpovídá Rek. spočetným množinám)

Každý obor hodnot ČRF je rekurzivně spočetná množina.

Důkaz

Máme ČRF g a její obor hodnot. Zkonstruujeme pseudoinverzní funkci h k ČRF g, tj. funkci takovou, že $dom(h) = rng(g)$ a to tak, že vyrobíme RS predikát $Q(x, y) \Leftrightarrow g(x) \simeq y$ a to má ČRF selektor, který hledáme – h.

Definice (Úseková funkce)

Funkce f je úseková, jestliže jejím definičním oborem je počáteční úsek $\mathbb{N}$ (nebo celé $\mathbb{N}$).

Věta (Rek. množiny a úsekové ČRF)

Rekurzivní množiny jsou právě obory hodnot rostoucích úsekových ČRF.

Důkaz

$\Rightarrow$: Definujeme ČRF f, která bude rostoucí a úseková.

- Začneme $f(0) \simeq \mu_x(x \in M)$.
- Dále $f(n+1) \simeq \mu_y(y > f(n) \wedge y \in M)$

$\Leftarrow$: Máme f rostoucí úsekovou ČRF.

1. V případě, že je f má konečné dom (tohle ale nejsme schopni efektivně rozpoznat!), víme jak, známe $D = dom(f)$ a tedy $rng(f)$ je rekurzivní.
2. V případě, že je f je všude definovaná (totální): $y \in M = rng(f) \Leftrightarrow \exists x : (f(x) = y)) \Leftrightarrow \exists x \leq y : (f(x) = y)$ Poslední ekvivalence platí, protože f je rostoucí a úseková. Tedy $y \in M \Leftrightarrow y \in \{f(0), \ldots, f(y)\}$.

Věta (O generování)

Mějme nekonečnou množinu M. Potom:

- Množina M je rekurzivní, právě když M lze generovat rostoucí ORF.
- Množina M je rekurzivně spočetná, právě když M lze generovat prostou ORF.

Důkaz

Důsledek předchozí, resp. následující věty.

Věta (Rek. spočetné množiny a prosté úsekové ČRF)

Rekurzivně spočetné množiny jsou právě obory hodnot prostých úsekových ČRF.

Důkaz

"$\Leftarrow$": Víme, obor hodnot ČRF je rekurzivně spočetná množina (z věty o tom, že ČRF odpovídají RSM).

"$\Rightarrow$": Mějme ČRF φ ($M = rng(\varphi)$ pro nějaké φ, z lemmatu o tom, že RSM je obor hodnot ČRF).

Důkaz provedeme pomocí rekurzivní množiny $B = \{\langle x, s\rangle : \varphi(x)\downarrow \text{ přesně za } s \text{ kroků }\}$. Je vidět, že každé x bude pouze v jednom z párů $\langle x, s\rangle$.

Množinu B lze, protože je rekurzivní, generovat pomocí rostoucí úsekové ČRF h. Funkce h generuje dvojice, definujeme tedy $g(x) \simeq (h(x))_{2,1}$. Zřejmě g je prostá, úseková a ČRF (a generuje $rng(\varphi)$).

Důsledek

Každá nekonečná rekurzivně spočetná množina obsahuje nekonečnou rekurzivní podmnožinu.

Důkaz

Mějme f, která prostě generuje M. Vyber rostoucí podposloupnost. Ta je rekurzivní.

$$g(0) = f(0)$$

$$g(n+1) = f(\mu_j(f(j) > g(n)))$$

Obor hodnot g je nekonečná rekurzivní množina a je podmnožinou M.

Kapitola 8

Státnice - Algoritmicky nerozhodnutelné problémy

8.1 Turingovy stroje

Definice (Turingův stroj)

Deterministický Turingův stroj (DTS) M s k-páskami, kde k je konstanta, je pětice

$$M = (Q, \Sigma, \delta, q_0, F)$$

- $Q =$ konečná množina stavů řídící jednotky
- $\Sigma =$ konečná pásková abeceda
- $\delta : (Q \backslash F) \times \Sigma^k \mapsto Q \times \Sigma^k \times \{L, N, R\}^k$ je přechodová funkce (částečná)
- $q_0 \in Q =$ počáteční stav
- $F \subseteq Q =$ množina přijímajících stavů

Definice (Konfigurace TS/Postovo slovo)

Konfigurace (jednopáskového) TS, neboli Postovo slovo, je obsah nejmenší souvislé části pásky, která obsahuje všechna neprázdná a čtené políčko; poloha hlavy a stav. Zapisujeme:

$$UqsV$$

kde U, V jsou části pásky nalevo a napravo od hlavy, q je stav a s čtené políčko.

Poznámka (Kombinování TS)

- Spojení dvou TS: napřed počítá $M1$, na výsledek pustím $M2$, tj. $M1 \wedge M2$
- Větvení (if-then-else): ve stroji $M1$ ze stavu q_0 přechod do (poč. stavu) $M2$, z q_1 do $M3$
- While-cyklus: složené spojení a větvení

Nutná opatrnost – stejná vnější abeceda, disjunktní vnitřní stavy atd.

Poznámka (Modifikace a omezení (stručně))

- Omezení pohybu na jeden směr – síla stroje klesne na úroveň konečných automatů
- Omezení na povinný pohyb (L/P) – OK
- Jen jedna činnost v taktu (buď zápis, nebo posuv) – OK
- Jednostranně omezená páska, více pásek, více hlav – stále stejná síla
- Okraje pásky z obou stran – páska není nekonečná, mám jen konfiguraci stroje – můžu mít potřebu pásku zvětšovat a zmenšovat (je možné)
- Omezení na 2 aktivní stavy – OK (jeden ale nestačí)

- Omezení na 2 symboly abecedy – OK (z toho jedno je "blank")
- K simulaci TS stačí 2 zásobníkové automaty – z jednoho zásobníku uděláme obsah pásky nalevo, z druhého napravo (vč. čteného znaku) a přehazujeme znaky.

8.2 Univerzální Turingův stroj

Věta (Univ. Turingův stroj)

Máme dánu abecedu A. Existuje univerzální TS $\mathcal{U}$ nad A, který pro každý TS nad A simuluje výpočet.

$$\mathcal{U}(\text{kód}(T) + \text{kód}(S)) \simeq T(S)$$

Důkaz

- Vezmeme $A = \{\lambda, |\}$, což stačí. BÚNO má každý TS jediný koncový stav q_f, počáteční stav buď q_s. Počet stavů – m – může být velký. Kód stavu q_i budiž blok znaků délky $m + 2$ ($| + i$-krát $| + m - i$-krát $\lambda + \lambda$).
- Pro $i \geq 1$ máme vždy dvě instrukce (jedna pro λ, druhá pro $|$). Ty se dají zakódovat do bloku $\times O_1 \times O_2 \times O_3 \cdots \times O_m \times$, kde $\times$ je pomocný symbol (v abecedě $\mathcal{U}$ být může) a O_i jsou kódy obou instrukcí pro stav r_i – kód zapisovaného písmene, pohybu a cílového stavu.
- Páska $\mathcal{U}$ pak vypadá následovně:

$$Y[\text{blok1}]Y[\text{blok2}]\Delta[\text{blok3}] \times O_1 \times O_2 \cdots \times O_m \times$$

  - "blok1" je konfigurace pův. stroje, jen obsah právě čteného pole je nahrazen pomocným symbolem M.
  - "blok2" kóduje aktuální stav pův. stroje.
  - "blok3" je jedna buňka, v níž je uložen obsah právě čteného pole.

- Univerzální stroj potom sestává z testu bloku2, zda obsahuje koncový stav, procedury vyčištění pásky a vydání výsledku a odsimulování jednoho kroku práce původního stroje
- Simulace:

1. najít relevantní blok O_i – stav i si nelze pamatovat přímo, proto musím z bloku 2 postupně umazávat $|$ a posouvat nějaký spec. symbol "zarážku" doprava
2. posunout zarážku na konkrétní instrukci podle bloku3 (čteného znaku)
3. provést instrukci (po kouskách přenést nový stav do bloku 2, pak 6 možností zapisování písmene a pohybu, při pohybu a mazání pozor na okraje pásky)

Důsledek

Díky tomu lze všechny TS očíslovat.

Věta (Halting problém)

Halting problém není algoritmicky rozhodnutelný.

Důkaz

Sporem nechť máme TS $H(T, K)$ rozhodující, zda se TS T zastaví nad daty K (a H se zastaví vždy a vydá buď 0 nebo 1). Potom lze vyrobit $Alg(K)$ takový, že $Alg(K)$ se zastaví, právě když $\mathcal{U}(K + K)$ se nezastaví (pomocí H). Pak $Alg(K)$ má nějaký kód, nazveme jej Q. Pak ale

$$Alg(Q) \text{ zastavi} \Leftrightarrow \mathcal{U}(Q + Q) \text{ nezastavi} \Leftrightarrow Alg(Q) \text{ nezastavi}$$

a to je spor.

Poznámka (Silně a slabě omezené mazání)

Omezíme mazání v TS:

- **slabě** – máme spec. symbol "kaňka" ($*$) a pravidla:
  - $\lambda \to$ cokoliv
  - cokoliv $\neq \lambda \to *$
- **silně** – máme abecedu jen $\{\lambda, *\}$ a povolený jen přepis $\lambda \to *$.

Oba dva případy mají stejnou sílu jako běžný TS (silné se slabým dá simulovat: kaňku kódovat jako blok samých kaněk, převést abecedu; normální TS se dá simulovat slabým postupným překreslováním konfigurací vedle sebe na pásku se současným měněním stavu).

Lze algoritmicky rozhodnout, zda TS T s konfigurací K někdy přepíše λ na něco jiného (existuje horní odhad počtu kroků v popsané části pásky). Nelze ale rozhodnout, zda TS T s konfigurací K někdy přepíše ne-λ na λ – to je ekvivalentní Halting problému (T simulujme silně omezeným T_1 a přidejme T_2, který smaže 1 kaňku. Pokud se T_1T_2 zastaví, musel se zastavit i T_1 a tím bychom rozhodli zastavení T).

Kapitola 9

Státnice - Věty o rekurzi

9.1 Věty o rekurzi

Věta (O rekurzi, o pevném bodě, self-reference)

Jestliže f je ORF jedné proměnné, potom existuje a takové, že $\varphi_{f(a)}(x) \simeq \varphi_a(x)$ pro všechna x (kde $\varphi_a(x)$ značí a-tou funkci, tedy odpovídá $\Psi_1(a, x)$).

Důkaz

Zjevně platí následující – první výraz je ČRF, má tedy své číslo e, druhá rovnost plyne ze S-m-n věty:

$$\lambda z,x(\varphi_{f(s_1(z,z))}(x)) \simeq \Psi_2(e, z, x) \simeq \varphi_{s_1(e,z)}(x)$$

Dosadíme $z = e$ a dostáváme hledané $a = s_1(e, e)$. Platí totiž $\varphi_{f(s_1(e,e))}(x) \simeq \Psi_2(e, e, x) \simeq \varphi_{s_1(e,e)}(x)$.

Vlastnosti programů a a $f(a)$

Funkce f zobrazuje program na program. Bod a je pevným bodem zobrazení f. Jak vypadají programy a a $f(a)$? Který z nich počítá déle? Uvidíme, že program a počítá déle než $f(a)$.

Co dělá program e na datech (z, x)? Počítá $\varphi_{f(s_1(z,z))}$, tj. vezme z a spočítá neprve $s_1(z, z)$, potom $f(s_1(z, z))$, který ale nemusí konvergovat. Jestliže $f(s_1(z, z))\downarrow$, spustí se na vstup x.

Co dělá program a? Program a vznikne jako $s_1(e, e)$. Mějme na vstupu x. Program a vezme e a přidá ho k x a spustí program e na (e, x). Co udělá program e na těchto datech? Spočítá $s_1(e, e)$ (tedy spočítá a), potom $f(s_1(e, e)) = f(a)$ a spustí program $f(a)$ na x.

Program a tedy neprve spočítá a, potom spočítá $f(a)$ (pokud konverguje) a ten simuluje na vstupu x. Program a je tedy složitější než $f(a)$ a počítá déle.

Poznámka z λkalkulu

V λkalkule sa ekvivalentné tvrdenie ukazuje trošku jednoduchšie. Pre každý λterm F (program F) existuje λterm X taký, že $X = FX$ (program F aplikovaný na X sa rovná X).

Dôkaz je nasledovný

- Majme F, pre ktoré chceme nájsť jeho pevný bod X.
- Nech $W = \lambda x.F(xx)$ (to je funkcia, ktorá x priradí $F(xx)$).
- $X = WW$ (to môžeme chápať ako program/funkciu W aplikovaný na W)
- $X = WW = (\lambda x.F(xx))W = F(WW) = F(X)$ (tretia rovnosť je βpravidlo λkalkulu. Ak si ale $(\lambda x.F(xx))W$ predstavíme ako funkciu, ktorá x priradí $F(xx)$ aplikovanú na W, rovnosť je (snáď) jasnejšia).

Věta (O generování pevných bodů)

Pro každou ORF f existuje prostá rostoucí PRF g taková, že platí:

$$\varphi_{f(g(j))}(x) \simeq \varphi_{g(j)}(x)$$

Tedy g rostoucím způsobem generuje nekonečně mnoho pevných bodů funkce f.

Důkaz

Postupujeme stejně jako v důkazu předchozí věty, jen máme o proměnnou (parametr j funkce g) navíc, tj. platí $\varphi_{f(s_2(z,z,j))}(x) \simeq \Psi_3(e,z,j,x) \simeq \varphi_{s_2(e,z,j)}(x)$. Zvolme $g(j) = s_2(e,e,j)$.

Věta (O rekurzi pro více proměnných)

Nechť f je ČRF $n+1$ proměnných. Potom existuje číslo a takové, že platí $\varphi_a(x_1,\ldots,x_n) \simeq f(a,x_1,\ldots,x_n)$ (tj. a je indexem funkce $\lambda x_1,\ldots,x_n f(a,x_1,\ldots,x_n)$).

Důkaz

$$f(y,x_1,\ldots,x_n) \simeq \Psi_{n+1}(e,y,x_1,\ldots,x_n) \simeq \varphi_{s_1(e,y)}(x_1,\ldots,x_n)$$

Následně aplikujeme větu o rekurzi na $s_1(e,y)$ v proměnné y a dostáváme hledané a (podle VR platí: $\exists a : \varphi_{s_1(e,a)} \simeq \varphi_a$).

Věta (O rekurzi v závislosti na parametrech)

Jestliže f je ČRF $n+1$ proměnných, potom existuje PRF g o n proměnných taková, že platí:

$$\varphi_{f(g(y_1,\ldots,y_n),y_1,\ldots,y_n)}(x) \simeq \varphi_{g(y_1,\ldots,y_n)}(x)$$

Důkaz

Pro $n = 0$ je to totéž jako verze bez parametrů. g nachází pevné body v závislosti na parametrech. Podobně jako v předchozích větách platí: $\varphi_{f(s_{n+1}(z,z,y_1,\ldots,y_n),y_1,\ldots,y_n)}(x) \simeq \Psi_{n+2}(e,z,y_1,\ldots,y_n,x) \simeq \varphi_{s_{n+1}(e,z,y_1,\ldots,y_n)}(x)$. Zvolme $g(y_1,\ldots,y_n) = s_{n+1}(e,e,y_1,\ldots,y_n)$.

9.2 Riceova věta

Věta (Rice)

Jestliže $\mathcal{A}$ je třída ČRF (jedné proměnné), která je netriviální (nejsou to všechny funkce a není prázdná), potom indexová množina $A_{\mathcal{A}} = \{x : \varphi_x \in \mathcal{A}\}$ (indexy programů, které vyčíslují funkce z $\mathcal{A}$) je nerekurzivní.

Důkaz

Sporem. Nechť $A_{\mathcal{A}}$ je rekurzivní. Potom lze vytvořit ORF f takovou, že všechny prvky z $A_{\mathcal{A}}$ zobrazí na nějaký prvek $b \notin A_{\mathcal{A}}$ a všechny prvky mimo $A_{\mathcal{A}}$ zobrazí na nějaký prvek $a \in A_{\mathcal{A}}$. Podle věty o rekurzi existuje pevný bod $f - u_0$, tedy platí:

$$\varphi_{u_0} = \varphi_{f(u_0)}$$

Takže:

$$u_0 \in A_{\mathcal{A}} \Rightarrow f(u_0) = b \notin A_{\mathcal{A}}$$
$$u_0 \notin A_{\mathcal{A}} \Rightarrow f(u_0) = a \in A_{\mathcal{A}}$$

To je ovšem spor, protože u_0 a $f(u_0)$ jsou indexy stejné funkce, a tedy buď obě čísla v $A_{\mathcal{A}}$ leží, nebo obě neleží.

Důsledky

Pozor, nejedná se o třídu programů, ale třídu funkcí. Tedy i pro jednoprvkovou $\mathcal{A}$ bude $A_{\mathcal{A}}$ nekonečná a nerekurzivní (každá funkce je vyčíslovaná nekonečně mnoha programy a rozhodnout o jejich ekvivalenci nelze efektivně).

Proto platí:

- Nechť $\mathcal{A} = \{\varphi_e\}$, potom $A_{\mathcal{A}} = \{x : \varphi_x = \varphi_e\}$ je nerekurzivní.
- Rozhodnout o rovnosti funkcí vyčíslovaných dvěma programy nelze algoritmicky.

Kapitola 10

Státnice - Stromové vyhledávací struktury

10.1 Binární vyhledávací stromy

Definice

Binární vyhledávací strom T reprezentující množinu prvků S z (uspořádaného) univerza U je úplný strom (tj. všechny vnitřní vrcholy mají 2 syny), ve kterém existuje bijekce mezi množinou S a vnitřními vrcholy taková, že pro v vnitřní vrchol stromu platí:

- všechny vrcholy podstromů levého syna jsou $\leq v$
- všechny vrcholy podstromů pravého syna jsou $> v$.

Listy reprezentují jednotlivé intervaly mezi vnitřními vrcholy. Můžeme je vynechat, ale s nimi je to (jak pro koho) logičtější.

Základní operace na stromech

- **MEMBER** – test, zda prvek x je obsažen ve stromě (vyhledání, zpravidla s využitím invariantu)
- **INSERT** – vložení prvku x do stromu
- **DELETE** – odebrání prvku x ze stromu
- **MIN**, **MAX**, **ORD** – nalezení prvního, posledního, k-tého největšího prvku
- **SPLIT** – rozdělení stromu podle x, které vyhodí, je-li ve stromě
- **JOIN** – spojení dvou stromů (jsou dvě verze, s přidáním prvku navíc, nebo bez něho)

Obecná (nevyvážená) implementace

- **INSERT**: najít list reprezentující interval, kam vkládám, udělat z něj normální uzel se vkládanou hodnotou a dát mu dva listy s podintervaly.
- **DELETE**: najdu vrchol, má-li jednoho syna-lista, pak druhý syn ho nahradí na jeho místě, jinak najdeme a dáme na jeho místo nejmenší větší vnitřní vrchol, jehož levý syn je list, a pravého syna tohoto vrcholu dáme na jeho místo.
- **SPLIT**: procházím stromem a hledám x, ost. prvky házím cestou do dvou stromů T_1, T_2, ve kterých si vždy uchovávám ukazatel na list, místo kterého vkládám (odkrojím syna, ve kterém hledám dál, místo něj vložím list, na který si pamatuju ukazatel).
- **JOIN**: s prvkem navíc, triviální – spojím stromy jako 2 syny nového prvku.

Tato struktura sama nepodporuje efektivní **ORD**, je nutné přidat navíc položky, které určují počet listů v podstromu každého vrcholu. ORD je pak jen jde do pravých synů a přičítá levé podstromy když může, jinak jde do levého syna (a nepřičte nic).

Analýza algoritmů

Definujme si pomocné hodnoty λ, π jako hodnoty nejbližšího menšího (levějšího), resp. většího (pravějšího) prvku na vyšší úrovni, nebo $-\infty$, resp. $+\infty$, pokud tyto prvky neexistují.

Korektnost vyhledávání: Je-li T' podstrom $t \in T$, pak T' reprezentuje $S \cap (\lambda(t), \pi(t))$ (a je to největší interval nezastoupený mimo T'). Pak pro vyhledání vrcholu x platí $\lambda(t) < x < \pi(t)$, vyšetřuji-li vrchol t.

Díky tomu je korektní **MEMBER** a **INSERT**. U **DELETE** musím dokázat korektnost případu s přehazováním vrcholů (dostávám bin. strom reprezentující $S \setminus \{x\}$).

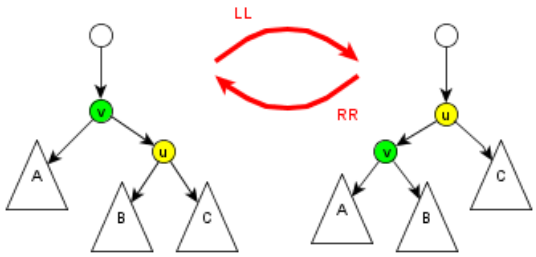
Korektnost **MIN**, **MAX**, **JOIN** je zřejmá, u **SPLIT** plyne z korektnosti hledání a toho, že moje označené listy jsou nejlevější, resp. nejpravější.

Korektnost **ORD** plyne z toho, že v každém kroku je k-tý prvek představován tolikátým v pořadí vrcholů akt. vyšetřovaného podstromu, kolik mi zbývá přičíst.

Složitost: Zpracování 1 vrcholu je vždy $O(1)$ a alg. se pohybuje po nějaké cestě od kořene k listu, která má $O(h)$, kde h je výška stromu.

Vyvažování

Chceme-li pro zachování efektivity operací zajistit, že výška bude $O(\log |S|)$, přidáme pro strom další podmínky, které bude muset splňovat a operace je zachovávat.

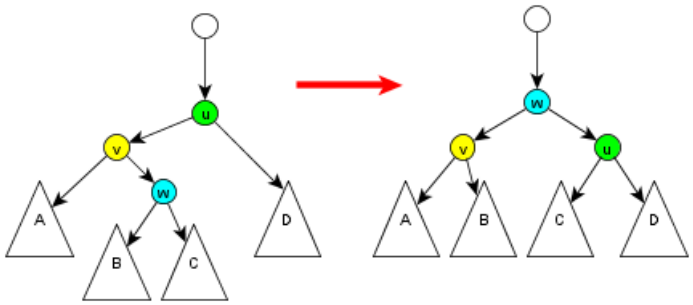


Obrázek 10.1: Rotace

Pro vyvažovací operace, které se snaží zachovat logaritmickou výšku, se používá pomocný algoritmus **ROTACE**(u, v):

1. Vezmu v, jeho pravého syna u a podstromy (zleva) A, B, C.
2. Přehodím v pod u, upravím ukazatel v otci a přeházím podstromy.

Existuje i symetrický případ, kdy se postupuje přesně opačným směrem. Někdy se této dvojici operací říká **LL-ROTACE** a **RR-ROTACE**.



Obrázek 10.2: Dvojrotace

Další potřebný algoritmus je **DVOJROTACE**(u, v, w):

1. Vezmu u, jeho levého syna v a pravého syna $v \quad - w$
2. Seřadím je tak, že w je otec obou, u vpravo a v vlevo.
3. Přitom opět upravím ukazatele v nadřízeném uzlu a přepojím podstromy.

Taky existuje symetrický případ. Jiné označení je **LR-ROTACE** a **RL-ROTACE**.
U obou operací lze aktualizovat i počty listů v podstromě a obě pracují v $O(1)$.

Alternativy k vyvažování

Je velká pravděpodobnost, že i bez vyvažování strom zůstane $O(\log |S|)$ vysoký a operace na něm můžou tak (bez vyvažování) běhat i rychleji. Proto existují i pravděpodobnostní postupy, nahrazující vyvažování znáhodněním posloupností operací. Další možnost jsou samoopravující struktury – operace samy bez dalších uchovávaných dat obstarávají vyvažování, existuje strategie, která zajistí dobré chování bez ohledu na data. Nebo se sleduje chování struktury, a když začne být příliš pomalá, vytvoří se nová – vyvážená. Poslední možnost je upravit dat. strukturu podle známého pravděpodobnostního rozdělení dat.

AVL-stromy

AVL-stromy (Adel'son, Velskii, Landis) jsou nejstarší vyvážené stromy, dodnes oblíbené, jednoduše definované, ale detailně technicky složité.

Podmínka AVL pro vyvažování: Výška pravého a levého podstromu lib. vrcholu se liší max. o 1.

Definice: $\eta(v)$ – výška vrcholu (délka nejdelší cesty z vrcholu do listů), $\omega(v)$ – rozdíl výšek levého a pravého podstromu ($\in \{-1, 0, 1\}$). Uchovávat potřebuji jenom ω.

Logaritmická výška

Výška celého stromu (η(kořen)) vychází z toho, že podstrom AVL stromu je vždy AVL strom. Vezmeme rekurzivní vztahy pro největší a nejmenší množinu uzlů v AVL stromu výšky i:

Nejmenší: $mn(i) = mn(i-1) + mn(i-2) + 1$

Největší: $mx(i) = 2mx(i-1) + 1$

Indukcí dokážeme, že $mx(i) = 2^i - 1$ a $mn(i) = F_{i+2} - 1$, kde F_i je i-té Fibonacciho číslo (pro ty platí vzorec $F_{i+2} = F_{i+1} + F_i$). Víme, že $\lim_{i\to\infty} F_i = \sqrt{5}(\frac{1+\sqrt{5}}{2})^{-i}$ a z toho zlogaritmováním plyne pro AVL-strom o výšce i s n prvky:

$$\log\left(\frac{c_1}{\sqrt{5}}\right) + (i+2)\log\left(\frac{1+\sqrt{5}}{2}\right) < \log(n+1) < i$$

A tedy $0.69i < \log(n+1) < i$, takže $i = \Theta(\log n)$.

Operace na AVL stromech

Operace **MEMBER** je stejná jako pro nevyvážené.

INSERT se musí po běžném vložení zabývat vyvažováním. Jde zpět ke kořeni a hledá, který nejnižší vrchol x nemá po vložení vyvážené ω, přičemž cestou upravuje ω. Na vrcholu x se provede vhodná ROTACE nebo DVOJROTACE, což zajistí vyváženost (existuje několik podpřípadů).

Operace **DELETE** odstraní vrchol a pak vyvažuje podobně jako INSERT, ale potřebuje víc operací (až $O(\log |S|)$ rotací). Asymptotická složitost je ale stejná – logaritmická.

Červeno-černé stromy

Červeno-černý strom má tyto čtyři povinné vlastnosti:

1. Každý uzel má definovanou barvu, a to černou nebo červenou.
2. Každý list je černý.
3. Každý červený vrchol musí mít oba syny černé.
4. Každá cesta od libovolného vrcholu k listům v jeho podstromě musí obsahovat stejný počet černých uzlů. Pro červeno-černé stromy se definuje černá výška uzlu ($\mathbf{bh}(x)$) jako počet černých uzlů na nejdelší cestě od uzlu k listu.

Garantování výšky

Podstrom libovolného uzlu x obsahuje alespoň $2^{\mathbf{bh}(x)} - 1$ interních uzlů. Díky tomu má červeno-černý strom výšku vždy nejvýše $2\log(n+1)$ (kde n je počet uzlů). (Důkaz prvního tvrzení indukcí podle $\mathbf{h}(x)$, druhého z prvního a třetí vlastnosti červeno-černých stromů)

Algoritmy

U algoritmů **INSERT** a **DELETE** jde také o vložení a následné vyvažování. Bez porušení vlastností červeno-černých stromů lze kořen vždy přebarvit načerno, můžeme pro ně předpokládat, že kořen stromu je vždy černý.

INSERT vypadá následovně:

- Vložený prvek se přebarví načerveno.
- Pokud je jeho otec černý, můžeme skončit – vlastnosti stromů jsou splněné. Pokud je červený, musíme strom upravovat (předpokládejme, že otec přidávaného uzlu je levým synem, opačný připad je symetrický):
  - Je-li i strýc červený, přebarvit otce a strýce načerno a přenést chybu o patro výš (je-li děd černý, končím, jinak můžu pokračovat až do kořene, který už lze přebarvovat beztrestně).
  - Je-li strýc černý a přidaný uzel je levým synem, udělat pravou rotaci na dědovi a přebarvit uzly tak, aby odpovídaly vlastnostem stromů.
  - Je-li strýc černý a přidaný uzel je pravým synem, udělat levou rotaci na otci a převést tak na předchozí případ.

DELETE se provádí takto:

- Skutečně odstraněný uzel (z přepojování – viz obecné vyhledávácí stromy) má max. jednoho syna. Pokud odstraňovaný uzel byl červený, neporuším vlastnosti stromů, stejně tak pokud jeho syn byl červený – to řeším přebarvením toho syna načerno.
- V opačném případě (tj. syn odebíraného – x – je černý) musím udělat násl. úpravy (předp., že x je levým synem svého nového otce, v opačném případě postupuji symetricky):
  - x prohlásím za "dvojitě černý" ("porucha") a této vlastnosti se pokouším zbavit.
  - Pokud je (nový) bratr x (buď w) červený, pak má 2 černé syny – provedu levou rotaci na rodiči x, prohodím barvy rodiče x a uzlu w a převedu tak situaci na jeden z násl. případů:
    - Je-li w černý a má-li 2 černé syny, prohlásím x za černý a přebarvím w načerveno, rodiče přebarvím buď na černo (a končím) nebo na "dvojitě černou" a propaguji chybu (mohu dojít až do kořene, který lze přebarovat beztrestně).
    - Je-li w černý, jeho levý syn červený a pravý černý, vyměním barvy w s jeho levým synem a na w použiji pravou rotaci, čímž dostanu poslední případ:
    - Je-li w černý a jeho pravý syn červený, přebarvím pravého syna načerno, odstraním dvojitě černou z x, provedu levou rotaci na w a pokud měl původně w (a x) červeného otce, přebarvím w načerveno a tohoto (teď už levého syna w) přebarvím načerno.

MIN a **MAX** jsou stejné jako pro nevyvážené.

JOIN (s prvkem navíc): mám-li černou výšku u obou stejnou, není co řešit, pokud ne, projdu po tom s větší $\mathbf{bh}(x)$ do patra, kde se výšky rovnají, půjčím si přísl. podstrom a slepím s ním, vrátím celek zpátky a aplikuji vyvažování, jako kdybych vložil 1 prvek (poruším výšku max. o 1).

SPLIT: rozhazuji podstromy do zásobníků, odkud je pak slepuji operací **JOIN**.

Každý algoritmus pracuje jen s vrcholy na jedné cestě od kořene k listům a s každým dělá konstantně činností, takže všechny algoritmy mají logaritmickou složitost. DELETE volá max. 2 rotace nebo 1 rotaci a 1 dvojrotaci, INSERT zase max. 1 rotaci nebo dvojrotaci (i když přebarvovat můžou rekurzivně až do kořene).

Váhově vyvážené stromy (BB-α)

Dnes jsou už na ústupu, ale občas se ještě používají. Mějme $1/4 < \alpha < 1 - \sqrt{2}/2$, označme $p(T)$ počet listů ve stromě T. Pak strom je BB-α, když

$$\alpha \leq \frac{p(T_{\text{levý}(v)})}{p(T_v)} \leq 1 - \alpha$$

pro T_v jako podstrom určený (každým) vrcholem v. O BB-α stromech platí, že:

$$\text{výška}(T) \leq 1 + \frac{\log(n+1)-1}{\log \frac{1}{1-\alpha}}$$

Takže jsou také vyvážené a operace mají zaručenou logaritmickou hloubku, vyvažuje se na nich také rotacemi a dvojrotacemi. Vždy totiž existuje $\alpha \leq d \leq 1 - \alpha$ takové, že když mám strom, jehož oba podstromy splňují vlastnosti a navíc $p(T_l)/p(T) \leq \alpha$ a $p(T_l)/p(T) - 1$ nebo $p(T_l) + 1/p(T) + 1$ vyhovuje, vezmu $\rho = \frac{p(T')}{p(T_r)}$, kde T' je určen levým synem T_r, a pro $\rho \leq d$ provedu ROTACE(T, T_r), jinak DVOJROTACE(T, T_r, T') a dostanu BB-α strom (bez důkazu). Opačný případ je popsaný symetricky.

Mají pěknou vlastnost, kvůli které se používaly: pro $\forall\alpha$ existuje $c > 0$ takové, že každá posloupnost k operací INSERT a DELETE volá max. $c \cdot k$ rotací a dvojrotací.

10.2 B-Stromy a jejich varianty

(a,b)-stromy

(a, b)-strom pro $a \leq b$ přirozená je strom T , který splňuje následující podmínky:

- každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvíc b synů
- všechny cesty od kořene k listům mají stejnou délku

Tato definice je ale pro praktické účely příliš obecná – budeme chtít navíc podmínky:

- $a \geq 2$ a $b \geq 2a - 1$
- kořen je buď list nebo má alespoň 2 a nejvíc b synů

Takový (a, b)-strom existuje pro každý přirozený počet listů, jeho výška je mezi $\log_b n$ a $1 + \log_a(\frac{n}{2})$, tedy $O(\log n)$. Indukcí: strom o výšce h má $2a^{h-1} \leq n \leq b^h$ listů (přidáním h-té hladiny do stromu s k listy dostaneme strom s $ka \leq n \leq kb$ listy.

Reprezentace množiny

Strom reprezentuje nějakou množinu S (prvků z univerza U), když mám bijekci mezi uspořádáním S a lexikografickým uspořádáním listů.

Každý vnitřní vrchol v obsahuje informaci o počtu synů $\rho(v)$, pole ukazatelů na syny S_v a pole H_v prvků z U takových, že i-tý je největší v S reprezentovaný v podstromě i-tého syna. Listy mají jen svůj prvek.

Pro každý prvek kromě největšího existuje vnitřní vrchol, který obsahuje jeho klíč, proto lze listy i vynechat a ukládat data ve vnitřních vrcholech (což ale není moc přehledné). Vrcholy můžou mít i odkaz na otce, nebo si otce můžu pamatovat při průchodech dolů (na vrcholech, ke kterým jsem nedošel od kořene, otce nepotřebuju).

Algoritmy

Máme pomocnou funkci **VYHLEDEJ**, který do hloubky projde stromem a vrátí nejbližší větší k nějakému prvku (nebo prvek sám, je-li ve stromě). Základní operace:

- **MEMBER**: přímo použije onu pomocnou operaci a pak zjistí, jestli našel, co hledal
- **INSERT**: vyhledám místo kam, pokud tam prvek není, vytvořím nový list, připojím na správné místo do S_v a postupně nahoru štěpím, je-li potřeba, extrémně rozštěpím kořen.
- **DELETE**: najdu prvek, najdu, kam je pověšený a to jedno políčko v S_v a H_v zruším, opravím ρ, pokud dostanu méně než a synů v uzlu, spojím s bezprostředním bratrem (má-li ten právě a synů), nebo přesunu nějaký list z bratra do mého uzlu.

Oba algoritmy pracují v $O(\log |S|)$ v nejhorším případě.

JOIN (bez prvku navíc): Přepokládá $\max S_1 < \min S_2$. Je-li $h(T_1) \geq h(T_2)$, najde v T_1 hladinu o 1 nad připojení a v ní největší prvek / vytvoří nadkořen T_1 v případě rovnosti, slije do něj prvky obou kořenů a případně provede štěpení. Jinak hledá a připojuje v T_2. Potřebuje čas $O(|h(T_1) - h(T_2)|)$.

SPLIT: Prochází postupně dolů, rozděluje uzly (podstromy s prvky $< x$, resp. $\geq x$) a hází výsledky do 2 zásobníků. Pokud oddělí více než 1 krajní prvek, hodí na zásobník strom, jehož kořen je právě oddělená část uzlu, jinak na zásobník dává podstrom onoho krajního prvku. Tak pokračuje až k listům, pokud tam najde přímo x, tak ho vyhodí. Stromy ze zásobníků spojí postupným voláním **JOIN** – v 1 zásobníku jsou max. 2 stromy stejné výšky, celkem jich je $k \leq 2 \log_a |S|$ a jejich zpracování trvá $O(\sum_{i=1}^{k-1}(h(T_i) - h(T_{i+1}) + 1)) = O(h(T_1) + k)$.

ORD: S takovouto reprezentací efektivně nejde, proto musíme navíc $\forall$ uzel udržovat pole P_v s počty vrcholů v jeho jednotlivých podstromech a při vkládání a odebírání ho průběžně aktualizovat. Pak v ORD procházím do hloubky a postupně přičítám velikosti přeskočených podstromů (pokud bych se přičtením dalšího dostal k 0, jdu na nižší hladinu).

Implementace

- Pro vnitřní paměť se doporučuje $a = 2 - 3$ a $b = 2a$, pro vnější $a \approx 100$ a $b = 2a$ (tj. v obou případech vlastně B-stromy).
- Při přístupu více uživatelů ke struktuře je problém s aktualizačními operacemi – zamykání celého stromu není efektivní: použije se vyvažování shora dolů: algoritmus INSERT zamkne uzel, jeho otce a syny. Pak pokud je počet synů $= b$, rozštěpí ho (předem), nebude se pak už štěpit zpátky.
  - Aby tohle fungovalo (abych měl $\geq a$ synů všude), je nutné $b \geq 2a$.
  - Podobně funguje DELETE – najde-li uzel s a syny, provede "preventivně" slití nebo přesun.
  - Provádí se tak víc slití a štěpení než v původní variantě, ale asymptoticky je to furt stejné.
  - Pro takovéto struktury na externí paměti se doporučuje $a \approx 100$ a $b = 2a + 2$.

B-Stromy

B-strom řádu m je vlastně (a,b) strom pro $a = \frac{m}{2}$ a $b = m$. V určitých implementacích se ovšem data nacházejí už ve vnitřních vrcholech, potom má každý uzel vždy o 1 méně datových záznamů než potomků. Pokud jsou data uložena až v listech, jedná se o **Redundantní B-strom**.

Implementační detaily:

- Někdy jdou z uzlů na data jen pointery, listy můžou mít jinou (jednodušší) dat. strukturu než vnitřní uzly.
- Pro implementaci je vhodné pamatovat si celou aktuálně procházenou větev v nějakém bufferu.
- V redundantních stromech nemusím při odstranění dat odstraňovat klíč ve vnitrnich uzlech (lze podle toho hledat i kdyz to tam neni).
- Vylepšení – **vyvažování stránek**– při přetečení stránky se nejdřív dívám, jestli není volno v sousedních. Pokud ano, přerozdělím a upravím klíče – zaručuje lepší zaplnění, ale je pomalejší. Podobně je možné vyvažovat počty se sousedy v případě vyhazování (i když nemerguju).

Další varianty

B* stromy – Na základě vyvažování stránek zpřísníme podmínky na počet uzlů: Kořen ma min. 2 potomky, ost. uzly minimálně $\lceil (2m-1)/3 \rceil$ potomků, všechny větve jsou stejné dlouhé. Štěpení se odkládá, dokud nejsou sourozenci plní, potom se štěpí buď 2 do 3 (jen s jedním sourozencem), nebo 3 do 4 (s oběma) uzlů. Při odebírání se slévají 3 uzly do 2 (nebo 4 do 3). Štěpení a slévání jde zesložitit ještě na víc stránek.

Odložené štěpení – používá stránku přetečení, vkládá znova, až když se naplní. Stránka přetečení může být jedna pro jeden listový uzel, nebo ji může sdílet nějaká skupina listů – štěpí se, až když jsou všechny listy i přetečení zaplněné. Tedy pokud má strom víc než 1 úroveň, má všechny listy zaplněné (za předpokladu nepoužití **DELETE**). S odebíráním musím i slévat a štěpit skupiny – jejich velikost není pevná.

Prefixové stromy – pro redundantní B-stromy; klíče jsou co nejkratší řetězce nutné k odlišení listů, nikoliv celé hodnoty, které se nacházejí až v listech. Při vkládání a štěpení stránek se nějakou heuristikou hledá nejkratší prefix, který by dvě vznikající stránky oddělil. Mazaní a slévání – žádná změna. Další zkrácení – u potomků se neopakuje předpona klíče, kterou ma rodič – to ale hodně zvýší nároky na CPU.

B+ stromy – pro intervalové dotazy: zrychlení tím, že zřetězíme vždy uzly v jedné hladině (a nebo jenom listy), tj. přidáme do uzlů ukazatele na levého a pravého souseda.

Hladinově propojené (a,b)-stromy s prstem (Finger trees) – pro vyhledávání navíc ještě přidáme odkaz na otce do každého uzlu a pro celou strukturu jeden "prst" – odkaz na nějaký list. Vyhledávání začíná od prstu a postupuje nahoru, dokud nenajde podstrom, v němž by měl být hledaný prvek; potom se spustí dolů. Pokud je prvek poblíž prstu, je to rychlejší než klasická varianta. Typicky máme funkci nastevní prstu na nějaký prvek a pokud se motáme v jeho okolí, vyjde to lépe.

Proměnná délka zaznamů – modifikace pro záznamy různé délky: neštěpit podle počtu záznamů, ale na zhruba 1/2 podle velikosti. Podmínka existence uzlu: součet délek záznamů v něm je $\geq B/2$ kde B je délka uzlu(stránky) (pro B* stromy $2B/3$). Problémy: dlouhé klíče mají tendenci propadávat ke kořeni, tím se zmenšuje arita stromu; může se 1 stránka štěpit i na 3 (pokud vkládám záznam delší nez 1/2 stránky); vložením záznamu může dojít ke zmenšení stromu (jak, to se ve skriptech nepíše :( ) Nejde vyrobit nezávislé **INSERT** a **DELETE**, řešení: univerzální alg. nahrazování řetězce řetězcem, **INSERT** a **DELETE** jsou jeho spec. příp. Řešeni snižování arity stromu: minimalizace délky klíčů (nalezeni klíče min. délky, která navíc splňuje min. naplněni) — pro B* stromy docela složité.

Amortizované odhady počtů štěpení a slití a vyvážení

Obecně může **INSERT** volat až $\log(|S|)$-krát štěpení a **DELETE** $\log(|S|)$-krát slití a jedno vyvážení (přesun). Začínáme-li s prázdným stromem a měříme na nějaké posloupnosti n operací, zjistíme, že jde amortizovaně o $O(1)$.

Důkaz pro $(2,4)$-stromy:

- Použijeme bankovní metodu, kdy **INSERT** bude stát dvě jednotky a **DELETE** jednu.
- Za štěpení a slití pak budeme platit vždy jednu jednotku, vyvážení nebude stát nic, protože je v každém **DELETE** voláno max. jednou a asymptoticky nic nezkazí.
- V jednotlivých uzlech stromu budeme udržovat následující počty jednotek podle stupně uzlů (přidáváme i stupně 1 a 5, které mají uzly těsně před štěpením a sléváním):

$\rho(v)$	1	2	3	4	5
jednotek	2	1	0	2	4

- Potom **INSERT** a **DELETE** bez štěpení díky své ceně udržují (občas i přeplácí) správné stavy jednotek v uzlech – snížení stupně stojí max. 1 a zvýšení max. 2 jednotky.

- Dojde-li ke štěpení uzlu stupně 5 do uzlů stupňů 3 a 2, čtyři jednotky se zaplatí: jedna za rozdělení, jedna na účet nového uzlu stupně 2 a (max.) dvě za zvýšení stupně rodiče.

- Dojde-li k vyvažování (přesunu), máme 2 jednotky, což nám stačí k vytvoření dvou uzlů stupně 2 ze stupňů 1 a 3 a přebývá nám při vyvážení uzlů stupňů 1 a 4.

- Sléváme-li uzly stupně 1 a 2, máme 3 jednotky celkem: jednou zaplatíme vlastní slití, jednou snížení stupně rodiče a jedna zbyde.

Protože všechny možnosti zachovávají invariant, je vidět, že celkem bude max. $2n$ operací slití a štěpení. Důkaz (prý) jde rozšířit i na libovolné a a $b \geq 2a$ (podle mě by mělo stačit zachovat ceny za operace a počty jednotek v krajních případech).

Pro $b = 2a - 1$ lze bohužel jednoduše nalézt takové posloupnosti operací, kde počet slití a štěpení je $O(n \log n)$, to samé, máme-li paralelní operace při $b = 2a + 1$. Proto se doporučuje $2a$, resp. $2a + 2$.

V hladinově propojeném stromě platí, že posloupnost n operací MEMBER, INSERT, DELETE a PRST vyžaduje $O(\log n +$ čas na vyhledání prvků).

10.3 Trie

Trie je vlastně stromová reprezentace slovníku. Její označení zřejmě pochází od slova "retrieval". Jejím úkolem je reprezentovat množinu $S \subseteq U$, kde U je tvořeno všemy slovy nad abecedou $\Sigma, |\Sigma| = k$ o délce l. Na této množině budeme provádět operace **MEMBER**, **INSERT** a **DELETE**.

Požadavek na délku se použije pouze u odhadů na složitost algoritmů. Ve skutečnosti ale není omezující – slova můžeme vždycky doplnit nějakými znaky mezery nebo lehce algoritmy upravit, aby s kratšími slovy počítaly.

Základní varianta

Definice

Trie je strom takový, že každý vnitřní vrchol má k synů, odpovídajících všem znakům abecedy. Každému vrcholu lze rekurzivně přiřadit slovo nad abecedou Σ následujícím způsobem:

- Kořeni patří prázdné slovo λ.

- a-tému synu patří slovo otce doplněné o a (kde a je libovolné písmeno).

Pro každý vnitřní vrchol trie musí platit, že tento vrchol je prefixem nějakého slova z reprezentované množiny S. Každý list obsahuje jeden bit, který udává přítomnost nebo nepřítomnost slova, které představuje, v množině S.

Je vidět, že taková struktura je dost paměťově náročná – každý vrchol potřebuje paměť $O(k)$ a celkem máme aspoň tolik vrcholů, co bodů množiny, násobeno délkou cesty k nim, tedy $O(kl|S|)$.

Operace

MEMBER je velice jednoduchý – postupně sestupuje stromem podél syna, který odpovídá i-tému písmenu hledaného slova v i-tém kroku. Pokud se dostane do listu dřív než dojde na konec slova, skončí neúspěchem. Jinak vrátí informaci o přítomnosti slova z listu, do kterého se dostal.

INSERT dojde do listu podobně jako **MEMBER**. Potom (je-li to potřeba) mění listy na vnitřní vrcholy a vkládá pokračování cesty až do dosažení délky slova. V posledním kroku upraví indikaci v listu.

DELETE vyhledá prvek a nastaví indikaci v jeho listu na FALSE. Pak se postupně vrací a dokud nalézá jen samé listy s FALSE, zruší celý vrchol a změní ho na list s FALSE.

Algoritmus **MEMBER** projde až l vrcholů a každý v konstantním čase (vrcholy se indexují přímo písmeny abecedy), tedy je $O(l)$. Algoritmy **INSERT** a **DELETE** vyžadují čas $O(kl)$, protože úprava jednoho vrcholu vyžaduje až k operací.

Komprimované Trie

Trie upravíme tak, že vyházíme vrcholy, jejichž slovo je prefixem stejné množiny slov jako slovo jejich otců (tj. vrcholy, kde nedochází k žádnému větvení a je jen jedna možnost pokračování). Ve vrcholech si teď ale místo toho musíme udržovat informaci o aktuální hloubce $\kappa(v)$ a navíc v listech musíme držet celé slovo, které reprezentují (abychom neztratili písmena z vrcholů, které jsme vyházeli).

Operace pak bude třeba upravit, ale protože takto upravené trie má jen $S-1$ vnitřních vrcholů, paměťová náročnost klesla na $O(k|S|)$

Operace

MEMBER pracuje podobně, ale ve vrcholu v vždy testuje $\kappa(v)+1$-ní písmeno hledaného slova. Nakonec (protože neotestoval všechna písmena a mohlo by tedy dojít ke kolizi) navíc porovná slovo uložené ve vrcholu se slovem, které jsme hledali.

INSERT pro nějaké slovo x opět vyhledá místo pro vložení a dojde do listu, kde najde jiné slovo y. Pak vezme největší společný prefix slov x, y a v jeho místě strom rozdělí – pokud ve správné hloubce už je vnitřní vrchol, pokračuje z něj, jinak nový dělící vrchol přidá. Potom upraví hodnoty v listech.

DELETE zruší informaci o mazaném slově stejně jako předtím. Navíc ale pokud zjistí, že otec "vyčištěného" listu v hierarchii stromu má jen jednoho dalšího potomka – vnitřní uzel, nacpe tohoto potomka na jeho místo.

Je vidět, že **INSERT** a **DELETE** změní maximálně jeden vnitřní vrchol a pracují tak v $O(k+l)$.

Očekávaná hloubka

Odhady časů pro operace **MEMBER**, **INSERT** a **DELETE** závisí na (maximální) délce slov l, ale takové hloubky komprimované trie dosáhne jen v nejhorším případě. Chceme proto odhad očekávané hloubky, za předpokladu, že reprezentovaná množina S je vzorkem dat z rovnoměrného rozdělení (což bývá často přibližně splněno).

Označíme q_d pravděpodobnost, že komprimovaný trie nad množinou $S, |S| = n$ má hloubku aspoň $d_n = d$. Pak je naše očekávaná (střední) hodnota hloubky:

$$E(d_n) = \sum_{i=1}^{\infty} i(q_i - q_{i+1}) = \sum_{i=1}^{\infty} q_i$$

Odhadneme proto velikost q_d. Víme, že hloubka trie je menší než d, když prefixy o délce d slov z naší množiny rozlišují tato slova jednoznačně. Pravděpodobnost, že toto nastane, je (počet jednoznačných prefixů a k nim počet libovolných doplnění do délky l, děleno počtem jednoznačných slov délky l, vše nad abecedou o velikosti k):

$$P(\text{jednoznačné rozlišení o délce } d) = \frac{\binom{k^d}{n} k^{n(l-d)}}{\binom{k^l}{n}}$$

Z toho:

$$q_d \leq 1 - P(\text{jednoznačné rozlišení o délce } d-1) =$$
$$1 - \frac{\binom{k^{(d-1)}}{n} k^{n(l-d+1)}}{\binom{k^l}{n}} \leq 1 - \frac{\left(\prod_{i=0}^{n-1}(k^{d-1}-i)\right)k^{n(l-d+1)}}{k^{nl}} = 1 - \prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right) \leq 1 - \exp\left(\frac{-n^2}{k^{d-1}}\right) \leq \frac{n^2}{k^{d-1}}$$

Poslední kroky jsme mohli udělat, protože platí (integrál s nerovností můžeme použít, protože daný logaritmus je klesající, při výpočtu integrálu použijeme substituci za $1 - \frac{x}{k^{d-1}}$):

$$\prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right) = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{k^{d-1}}\right)\right) \geq$$
$$\geq \exp\left(\int_0^n \ln\left(1 - \frac{x}{k^{d-1}}\right) \mathrm{d}x\right) = \exp\left((n - k^{d-1})\ln\left(1 - \frac{n}{k^{d-i}}\right) - n\right) \leq \exp\left(\frac{-n^2}{k^{d-1}}\right)$$

Očekávaná výška stromu pak vyjde, položíme-li $c = 2\lceil \log_k n \rceil$:

$$\sum_{i=1}^{\infty} q_i = \sum_{i=1}^{c} q_i + \sum_{i=c+1}^{\infty} q_i \leq \sum_{i=1}^{c} 1 + \sum_{i=c+1}^{\infty} \frac{n^2}{k^{i-1}} = c + \frac{n^2}{k^c}\left(\sum_{i=0}^{\infty} \frac{1}{k^i}\right) \leq 2\lceil \log_k n \rceil + \frac{1}{1 - \frac{1}{k}}$$

Trie v tabulce

Pokud se vzdáme operací **INSERT** a **DELETE**, můžeme trie reprezentovat na extrémně zcvrklém prostoru. Nejdříve si ho představme jako matici M dimenze $r \times s$, kde každý vnitřní vrchol odpovídá jednomu řádku a sloupce jsou písmena abecedy. Potom na pozici $M(v, a)$ je a-tý syn vrcholu v. Pole matice může obsahovat buď odkazy na další vrcholy (identifikátor řádku), nebo přímo slova, která jsou obsažena v reprezentované množině, nebo prázdnou hodnotu **null**. Ve vedlejším poli si musíme uchovat i hloubky vrcholů odpovídající nekomprimovanému trie – $\kappa(v)$.

Komprese matic – uložení do pole

Hodnoty **null** ale nepřinášejí novou informaci – stačí při průchodu maticí na další vrcholy testovat, zda nám stoupá hodnota $\kappa(v)$ a nakonec provést test shody s nalezeným prvkem. Díky tomu můžeme na místa, kde je **null**, v klidu ukládat něco jiného.

Matici M tak můžeme reprezentovat dvěma poli

- VAL – v něm budou hodnoty z různých řádků matice
- RD ("row displacement", "posunutí řádku") – bude udávat, kde začíná který řádek původní matice M ve VAL

Jednotlivé datové řádky původní matice se v poli VAL v klidu můžou překrývat, pokud překryté hodnoty jsou jen **null**. Formálně musíme zachovat, že když $M(i,j)$ je definováno, pak $M(i,j) = VAL(RD(i)+j)$ a že když $M(i,j)$ a $M(i',j')$ jsou definovány pro $(i,j) \neq (i',j')$, pak $RD(i)+j \neq RD(i')+j'$.

Pro nalezení "dobrého" rozložení řádků původní matice do pole VAL se používá algoritmus **First-Fit Decreasing**:

- Pro každý řádek původní matice M spočteme, kolik míst je non-**null** a setřídíme řádky matice podle této hodnoty v klesajícím pořadí
- Bereme řádky podle setřídění a vkládáme je na první místo od začátku pole VAL tak, že neporušují výše uvedené podmínky.

Označíme počet všech non-**null** hodnot jako m a počet non-**null** hodnot v řádcích s alespoň l non-**null** hodnotami jako m_l. Pokud řádky matice M splňují pravidlo harmonického rozpadu, tj. $\forall l : m_l \leq \frac{m}{l+1}$ (tj. např. více než polovina řádků obsahuje jen jednu skutečnou hodnotu), pak pro každý řádek i platí $RD(i) < m$ a algoritmus stavby polí potřebuje $O(rs + m^2)$ času (důkaz je hnusný).

Posouvání sloupců

Protože podmínku harmonického rozpadu splňuje jen málo matic, upravíme si obecné matice tak, aby ji splňovaly taky. Využijeme toho, že matici trochu "natáhneme" do počtu řádků (tím se, pravda, zvětší pole RD) a jednotlivé sloupce v ní rozstrkáme tak, aby v jednom řádku nevyšlo moc zaplněných míst najednou. Kde začíná který sloupec si zapamatujeme v dalším pomocném poli CD ("column displacement", "posunutí sloupce").

"Dobré" posunutí sloupců nalezneme obyčejným přístupem **First-Fit**, když pro každý sloupec j nalezneme nejmenší číslo $CD(j)$ splňující:

$$m(j+1)_l \leq \frac{m}{f(l, m(j+1))} \; \forall l = 0, 1, \ldots$$

Hodnota $m(j)$ je počet všech zaplněných míst v prvních j sloupcích právě konstruované matice a $m(j)_l$ je počet zaplněných míst v řádcích s alespoň l zaplněnými místy.

Pozorování:

- Je vidět, že každá funkce f musí splňovat $f(0, m(j)) \leq \frac{m}{m(j)} \; \forall j$, protože jinak by v algoritmu nemohla být splněna testovaná podmínka pro $l = 0$ (protože $m_j = m(j)_0$).
- Dále musí funkce f splňovat nerovnost $f(l,m) \leq l+1 \; \forall l$, aby výsledná matice splňovala podmínku harmonického rozpadu.

Dá se ukázat (a je to hnusný důkaz), že vhodná funkce je třeba $f(x,y) = 2^{x(2-\frac{y}{m})}$, protože splňuje obě podmínky a navíc výsledný vektor CD má délku s, vektor RD má délku menší než $4m \log\log m + 15.3m + r$ a vektor VAL má délku menší než $m + s$. Protože hodnoty CD indexují RD a hodnoty RD indexují VAL, plynou z toho omezení i na hodnoty v nich uložené.

Čas celého algoritmu vytváření matic je $O(s(r + m \log\log m)^2)$.

Další komprese vektoru RD

Protože M má jen m definovaných mít, z algoritmu pro výpočet RD plyne, že jen max. m míst v tomto vektoru bude různých od nuly. Proto můžeme použít následující kompresi (řekněme, že nenulových míst je t):

1. Vektor RD rozdělíme na n bloků o délce d.
2. Vytvoříme nový vektor CRD o délce t, který obsahuje jen nenulové prvky původního vektoru. Označme jejich původní pozice $i_j, j = 0 \ldots t-1$ a jejich pozice ve vektoru CRD jako $v(i_j)$.
3. Vytvoříme vektor $BASE$ o délce n, kde $BASE(x) = \begin{cases} -1 & i_j \div d \neq x \; \forall j = 0, \ldots t-1 \\ \min\{l; i_l \div d = x\} & \text{jinak} \end{cases}$
4. Vytvoříme matici $OFFSET$ typu $n \times d$, kde $OFFSET(x,y) = \begin{cases} -1 & x \cdot d + y \neq i_j \; \forall j \\ j - BASE(x) & x \cdot d + y = i_j \end{cases}$
5. Uložíme matici $OFFSET$ do vektoru OFF dimenze n tak, že z každého řádku vytvoříme číslo v soustavě o základu $d+1$: $OFF(x) = \sum_{k=0}^{d-1} (OFFSET(x,k)+1)(d+1)^k$.

Potom platí, že:

- $v(h) = 0 \Leftrightarrow OFFSET(h \div d, h \mod d) = -1$
- $v(h) = 1 \Rightarrow h = BASE(h \div d) + OFFSET(h \div d, h \mod d)$
- $OFFSET(i,j) = ((OFF(i) \div (d+1)^j) \mod (d+1)) - 1$

Celá tahle legrace má smysl, jen pokud $d \ll n$, a $t < n$. Když $d \leq \lceil \log\log n \rceil$, pak lze celé trie uložit pomocí pěti vektorů dimenze n s hodnotami menšími než $4n \log\log n$.

10.4 Haldy

Haldy se používají pro měnící se uspořádané množiny. Nevyžaduje se efektivní operace **MEMBER** (často se předpokládá s argumentem operace informace o uložení prvku). Požadují se malé nároky na paměť a rychlost ostatních operací.

Definice, operace

Halda je stromová struktura nad množinou (dat) S, jejíž uspořádání je dáno funkcí $f : S \to \mathbb{R}$, splňující lokální podmínku haldy:

$$\forall v \in S : f(\text{otec}(v)) \leq f(v), \text{ případně v duální podobě.}$$

Množina je reprezentovaná haldou, když přiřazení prvků vrcholům haldy je bijekce, splňující podmínku haldy. Různé druhy hald se liší podle dalších podmínek, které musí splňovat stromové struktury.

- Krom běžných operací můžu měnit uspořádání: operace **INCREASE** a **DECREASE** změní velikost f na nějakém daném prvku s se známým uložením o $+a$, $-a$.
- Další operace: **DELETEMIN** – smazání prvku s nejmenší hodnotou f.
- Pro operaci **DELETE** budeme požadovat přímé zadání uložení prvku.
- Navíc definujeme operaci **MAKEHEAP** – vytvoření haldy při známé množině a f,
- a **MERGE** – slití dvou hald do jedné, reprezentující $S_1 \cup S_2$ a $f_1 \cup f_2$, aniž by se ověřovala disjunktnost.

Regulární haldy

Pro d-regulární strom ($d \in \mathbb{N}$) s kořenem r platí, že existuje pořadí synů vnitřních vrcholů takové, že očíslování prohledáváním z r do šířky splňuje:

1. každý vrchol má nejvýše d synů
2. když vrchol není list, tak všechny vrcholy s menším číslem mají právě d synů
3. má-li vrchol méně než d synů, pak všechny vrcholy s většími čísly jsou listy

Potom takový strom s n vrcholy má max. jeden ne-list, který nemá právě d synů, jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$. Čísla synů vrcholu s číslem k jsou $(k-1)d+2, \ldots, kd+1$, číslo otce je $1 + \lfloor \frac{k-2}{d} \rfloor$. Takto vytvořená halda umožňuje i efektivní reprezentaci v poli.

Operace na regulárních haldách

- Není známa efektivní operace **MERGE**.
- Máme pomocné operace **UP**, **DOWN**, posunující prvek níž/výš ve struktuře, dokud není splněna podmínka haldy ("probublávání").
- **INSERT** jen vloží nový prvek za poslední a spustí **UP**
- **DELETE** nahradí odstraněný prvek posledním listem a volá **UP** nebo **DOWN** podle potřeby
- **DELETEMIN** odstraní kořen, nahradí ho posl. listem a volá **DOWN**
- **MIN** jen vrátí kořen
- **INCREASE** a **DECREASE** změní hodnotu f nějakého prvku a zavolají **DOWN**, resp. **UP** (pozor, je to naopak, než názvy napovídají).
- Operace **MAKEHEAP** vytvoří libovolný strom a pak postupně od posledního ne-listu ke kořeni volá na všechno **DOWN**.

U všech operací je korektnost zajištěna podmínkou haldy (a tím, že **UP** a **DOWN** zaručí její splnění).

Složitost operací

Běh **DOWN** vyžaduje $O(d)$ a **UP** $O(1)$ v každém cyklu, takže celkem jde o $O(d \log |S|)$ a $O(\log |S|)$.

Haldu lze vytvořit opakovaným **INSERT**em v čase $|S| \log |S|$, ale pro větší množiny je rychlejší **MAKEHEAP** – uvažujeme-li, že operace **DOWN** vyžaduje čas odpovídající výšce vrcholu. Ve výšce $k-i$ je d^i vrcholů. Tím dostávám celkový čas $O(\sum_{i=0}^{k-1} d^i(k-i)d)$, což se dá odhadnout jako $O(d^2|S|)$.

Aplikace

Heapsort – vytvoření haldy a postupné volání **MIN** a **DELETEMIN**. Lze ukázat, že pro $d = 3, d = 4$ je výhodnější než $d = 2$, empiricky je do cca 1000000 prvků $d = 6$ nebo $d = 7$ nejlepší. Pro delší posloupnosti je možné d zmenšit.

Dijkstra – normální Dijkstrův algoritmus, jen vrcholy grafu uchovávám v haldě, tříděné podle aktuálního d (horního odhadu vzdálenosti). Složitost $O((m+n)\log n)$, pro $d = \max\{2, \frac{m}{n}\}$ je to $O(m \log_d n)$ a pro husté grafy ($m > n^{1+\varepsilon}$) je lineární v m.

Leftist haldy

Leftist halda je binární strom (T, r). Označme npl(v) délku nejkratší cesty z v do vrcholu s max. 1 synem. Leftist halda musí splňovat následující podmínky:

1. má-li vrchol 1 syna, pak je vždy levý
2. má-li 2 syny l, p , pak npl(p)≤npl(l)
3. podmínka haldy na klíče prvků (ex. přiřazení prvků vrcholům stromu)

Pro leftist haldu se definuje pravá cesta (posl. pravých synů) a pokud máme takovou cestu délky k z vrcholu v, víme, že podstrom v do hloubky k je úplný binární strom. Délka pravé cesty z každého vrcholu je tedy logaritmická ve velikosti podstromu.

Operace jsou založeny na algoritmech **MERGE** a **DECREASE**.

- **MERGE** testuje prázdnost jednoho ze stromů (a pokud je jeden prázdný, vrátí ten druhý jako výsledek). Pokud ne, volá se rekurzivně na podstrom pravého syna kořene s menším klíčem dohromady s celým druhým a výsledek připojí místo onoho pravého syna. Pokud neplatí podmínka na npl, syny vymění.
- **INSERT** je to samé co vytvoření jednoprvkové haldy a zavolání **MERGE**.
- **DELETEMIN** je z**MERGE**ování synů kořene (a jeho zahození).
- **MAKEHEAP** je vytvoření hald z jednotl. prvků. Nacpu je do fronty a potom v cyklu vyberu dva první, zmerguju a hodím výsledek na konec, dokud mám ve frontě víc než 1 haldu.

INCREASE a **DECREASE** se dělají jinak.

- Mám pomocnou operaci **OPRAV**, která odtrhne podstrom a dopočítá všem vrcholům správné npl. Po odtržení vrcholu a příp. přehození pravého syna doleva jde nahoru, dokud provádí změny npl (možno až do kořene), vztahuje npl odspoda a příp. prohazuje syny.
- **DECREASE** se pak udělá snížením hodnoty ve vrcholu, zavoláním **OPRAV**, tj. jeho odříznutím od zbytku haldy, a **MERGE** podstromu a zbytku.

INCREASE: zapamatuju si levý a pravý podstrom vrcholu s mým prvkem a provedu na něj **OPRAV** (vyhodím ho), potom vyrobím nový vrchol s mým prvkem se zvednutou hodnotou a jako samostatnou haldu ho z**MERGE**uju s levým podstromem. Pravý podstrom z**MERGE**uju se zbytkem haldy a nakonec s tím z**MERGE**uju výsledek **MERGE** levého podstromu a zvednutého prvku.

Složitost

1 běh **MERGE** bez rekurze je $O(1)$, hloubka rekurze je omezena pravými cestami, takže je to $O(\log(|S_1| + |S_2|))$. Z toho plyne logaritmovost **INSERT** a **DELETEMIN**.

Pro **MAKEHEAP** se uvažuje, kolikrát projdou haldy frontou: po k projití frontou mají velikost 2^{k-1} a tedy fronta obsahuje $\lceil \frac{|S|}{2^{k-1}} \rceil$ hald. Jeden **MERGE** je $O(k)$ a jedno projití frontou pro všechny haldy tedy trvá $O(k\lceil \frac{|S|}{2^{k-1}} \rceil)$. Celkem dostávám $O(|S| \sum_{k=1}^{\infty} \frac{k}{2^{k-1}}) = O(|S|)$ (součet řady je 4).

OPRAV chodí jen po pravé cestě, takže má logaritmickou složitost. **INSERT**, **INCREASE** a **DECREASE** se díky ní dostanou taky na $O(\log |S|)$, protože jejich části kromě **MERGE** a **OPRAV** mají konstantní složitost.

Binomiální haldy

Binomiální stromy se definují rekurentně jako H_i, kde H_0 je jednoprvkový a H_{i+1} vznikne z dvou H_i, kdy se kořen jednoho stane dalším (krajním) synem kořenu druhého. Pak strom H_i má 2^i prvků, jeho kořen má i synů, jeho výška je i a podstromy určené syny kořene jsou právě $H_{i-1}, \ldots, H_0$.

Binomiální halda reprezentující S je seznam stromů T_i takový, že celkový počet vrcholů v těchto stromech je $|S|$ a je dáno jednoznačné přiřazení prvků vrcholům, respektující podmínku haldy. Každý strom je přitom izomorfní s nějakým H_i a dva $T_i, T_j, i \neq j$ nejsou izomorfní.

Existence binomiální haldy pro každé přirozené $|S|$ plyne z existence dvojkového zápisu čísla.

Operace

Operace na binomiálních haldách jsou založené na MERGE.

- **MERGE** pracuje stejně jako binární sčítání – za pomoci operace **SPOJ** (slepení dvou stromů, přilepím jako syna toho, který má v kořeni vyšší klíč) slepí stromy stejného řádu, přenáší výsledky do dalšího spojování (přenos + obě haldy mající strom daného řádu = vyplivnutí 1 stromu na výsledek a spojení zbývajících dvou).
- **INSERT** je MERGE s jednoprvkovou haldou.
- **MIN** je projití kořenů a vypsání nejmenšího.
- **DELETEMIN** je **MIN**, odebrání stromu s nejmenším prvkem v kořeni a přidání (**MERGE**) podstromů jeho kořene do haldy.
- **INCREASE** a **DECREASE** se dělají úplně stejně jako u regulárních hald.
- Přímo není podporováno **DELETE**, jen jako **DECREASE** + **DELETEMIN**.
- **MAKEHEAP** se provádí opakováním **INSERT**.

Složitost **MERGE** je $O(\log|S_1| + \log|S_2|)$, protože 1 krok **SPOJ** je konstantní. Halda má nejvýše $\log|S|$ stromů, takže **MIN** a **DELETEMIN** mají tuto složitost. Výška všech stromů je $\leq \log|S|$, což dává složitost **INCREASE** $O(\log^2|S|)$ a **DECREASE** $O(\log|S|)$. Pro odhad složitosti **MAKEHEAP** se použije amortizovaná složitost přičítání jedničky k binárnímu číslu, což je $O(1)$, tedy celkem $O(|S|)$.

Líná implementace

Vynecháme předpoklad neexistence dvou izomorfních stromů v haldě a budeme "vyvažování" provádět jen u operací **MIN** a **DELETEMIN**, kdy se stejně musí projít všechny stromy. **MERGE** je pak prosté slepení seznamů hald. Vyvažování se provádí operací **VYVAZ**, která sloučí izomorfní stromy (podobně jako **MERGE** z pilné implementace).

Složitost **INSERT** a **MERGE** je $O(1)$, ale **DELETEMIN** a **MIN** v nejhorším případě $O(|S|)$.

Amortizovaná složitost vychází ale líp: použijeme potenciálovou metodu, když za hodnocení konfigurace $w(H)$ zvolíme počet stromů v haldě $|\mathcal{H}|$. **INSERT** a **MERGE** ho nemění, resp. mění o 1, takže jsou stále $O(1)$.

Operace **VYVAZ** potřebuje $O(|H|)$, protože slití dvou stromů trvá konstantně dlouho a nelze slévat víc stromů, než kolik jich je v haldě. Kromě operace **VYVAZ** potřebuje **MIN** $O(|\mathcal{H}|)$ a **DELETEMIN** $O(|\mathcal{H}| + \log|S|)$ (max. stupeň stromu je logaritmický).

Dohromady vychází amortizovaná složitost pro **MIN**: $am(o) = t(o) - w(H) + w(H') = O(|\mathcal{H}| - |\mathcal{H}| + \log|S|)$, protože výsledný počet stromů $|H'|$ odpovídá pilné implementaci. Pro **DELETEMIN** podobně dostanu $O(|\mathcal{H}| + \log|S| - |\mathcal{H}| + \log|S|) = O(\log|S|)$.

Fibonacciho haldy

Definují se jako množiny stromů, které splňují podmínku haldy a musely vzniknout posloupností operací z prázdné haldy. Všechny operace zachovávají podmínku, že jednomu vrcholu lze odříznout max. dva syny. Strom má rank i, má-li jeho kořen i synů (podobné jako izomorfismus s H_i u binomiálních hald).

Podmínka odříznutí max. dvou synů se zachovává pomocnou operací **VYVAZ2**. Když vrchol není kořen a byl mu předtím někdy odříznut syn, je speciálně označený. **VYVAZ2** prochází od daného vrcholu ke kořeni a dokud nalézá označené vrcholy, odtrhává je i s jejich podstromy, ruší jejich označení a vkládá do haldy jako zvláštní stromy. Když se vrchol stane kořenem, označení se zapomene.

Operace

MERGE, INSERT, MIN a **DELETEMIN** jsou stejné jako v líné implementaci binomiálních hald, jen požadavek na isomorfismus s H_i je nahrazen požadavkem na daný rank. Pomocné operace z binomiálních hald **VYVAZ** a **SPOJ** jsou také stejné.

DECREASE, INCREASE a **DELETE** vycházejí z leftist hald. Používají pomocnou operaci **VYVAZ2**

- **DECREASE** odtrhne podstrom určený snižovaným vrcholem (není-li to už kořen), zruší u něj případné označení a vloží ho zvlášť do haldy, na odtržené místo zavolá **VYVAZ2**.
- **INCREASE** provede to samé, jen ještě roztrhá podstrom zvedaného vrcholu (odtrhne všechny syny, zruší jejich příp. označení a vloží jako samostatné stromy do haldy) a vloží zvednutý vrchol do haldy zvlášť.
- **DELETE** je to samé co **INCREASE**, bez přidání vrcholu zpět do haldy.

Korektnost a složitost

Operace **SPOJ** podobně jako u binomiálních hald vyrobí ze dvou stromů ranku i jeden strom ranku $i+1$. Operace **VYVAZ2** zajistí, že od každého vrcholu kromě kořenů byl odtržen max. 1 syn – když odtrhnu dalšího, odtrhu i tento vrchol a propaguju operaci nahoru.

Složitost operací:

- **MERGE** a **INSERT** je $O(1)$ (stejně jako u binomiálních hald)
- **MIN** má $O(|\mathcal{H}|)$ (nemění označení vrcholů)
- **DELETEMIN** $O(|\mathcal{H}|+\text{maxrank}(\mathcal{H}))$, kde *maxrank* udává maximální rank stromu v haldě (může navíc odznačit některé vrcholy)
- **DECREASE** je $O(1+c)$, kde c je počet odznačených vrcholů (navíc označí max. 1 vrchol)
- **INCREASE** a **DELETE** jsou $O(1+c+d)$, kde navíc d je počet synů zvedaného nebo odstraňovaného vrcholu (také označí navíc max. 1 vrchol).

Pro výpočet amortizované složitosti použijeme potenciálovou metodu a zvolíme hodnotící funkci w jako počet stromů v haldě + 2× počet označených vrcholů. Můžeme říct, že amortizovaná složitost **MERGE**, **INSERT** a **DECREASE** je $O(1)$.

Označme max. rank stromů v lib. haldě reprezentující n-prvkovou množinu jako $\rho(n)$. Amortizovaná složitost **MIN**, **DELETEMIN**, **INCREASE** a **DELETE** pak je $O(\rho(n))$ (pro **MIN** a **DELETEMIN** je vzorec amortizované složitosti podobný jako u binomiálních hald, pro **INCREASE** a **DELETE** je to vidět přímo ze vzorců).

Pro odhad $\rho(n)$ je potřeba znát fakt, že i-tý nejstarší syn libovolného vrcholu má aspoň $i-2$ synů (plyne z toho, že se slévají jen stromy stejného řádu a odtrhnout lze max. jednoho syna).

Vezmeme tedy nejmenší strom T_j ranku j, který toto splňuje. Ten musí být složením T_{j-1} a T_{j-2} (vzniká tak, že se slijí dva T_{j-1} a potom se na tom, který je pověšený jako syn nového kořenu, provede **DECREASE** a tím se z něj stane T_{j-2}). Z minimálního počtu synů se dá odvodit i rekurence $|T_k| \geq 1+1+|T_0|+\cdots+|T_{k-2}|$, která dá indukcí to samé.

Potom $|T_{k+1}| = F_k$, kde F_k je k-té Fibonacciho číslo. Pro Fibonacciho čísla platí, že $\lim_{k\to\infty} F_k = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k$. Proto je $\rho(n) = O(\log(n))$, což dává logaritmickou amortizovanou složitost pro **MIN**, **DELETEMIN**, **INCREASE** a **DELETE**. Z toho pochází i název Fibonacciho haldy.

Aplikace

Fibonacciho haldy se díky své rychlosti **INSERT**, **DECREASE** a **DELETEMIN** často používají v grafových algoritmech. Praktické porovnání rychlosti s jinými haldami však není dosud přesně prostudováno.

Motivací pro vývoj Fibonacciho hald byla možnost **aplikace v Dijkstrově algoritmu**. Dává totiž složitost celého algoritmu $O(m+n\log n)$, což by mělo být lepší pro velké, ale řídké grafy proti d-regulárním haldám. O prakticky zjištěném "zlomu" ale nevíme.

Kapitola 11

Státnice - Hašování

11.1 Hashování

Základní motivací pro hashování je slovníkový problém, kdy máme za úkol reprezentovat množinu S prvků z nějakého univerza U a provádět na ní následující operace:

- **MEMBER** (je třeba, aby tato operace probíhala velmi rychle)
- **INSERT**
- **DELETE**

Aby byl **MEMBER** rychlý, bylo by nejlepší mít v paměti pole bitů o velikosti U. V případě, že $|S| << |U|$ (a navíc U může být neúnosně velké), použiji hashovací funkci $h : U \to \{0, \ldots, m-1\}$ a množinu S reprezentuji polem s m políčky tak, že $x \in S$ je uložen na indexu $h(x)$. Předpokládejme, že funkce h se dá spočítat v čase $O(1)$ – jiné funkce vlastně nemají smysl, protože nepřináší dostatečné zrychlení.

Problém je, když nastane kolize: $x \neq y$, $h(x) = h(y)$. Jednotlivé druhy hashování, které následují, se liší strategiemi předcházení a řešení kolizí.

Pro následující analýzy si označíme:

- $|S| = n$
- $|U| = N$
- Load factor (faktor zaplnění) – $\alpha = \frac{n}{m}$.

Hashování se separovanými řetězci

V tomto typu hashování se kolize řeší řetězením ve spojácích: pro každé políčko založíme zvlášť spoják všech prvků, které se do něj hashují. Všechny algoritmy je musí projít. Předpokládejme, že řetězce jsou prosté – nic se v nich neopakuje. V nejhorším případě mají všechny prvky stejný hash a máme jen jeden seznam.

Paměťová náročnost je pro každý seznam $O(1 + l(i))$, kde $l(i)$ je délka toho seznamu.

Existují dvě varianty – neuspořádaná a s s uspořádanými prvky v řetězcích. Liší se jedině v očekávaném počtu testů pro neúspěšné hledání (když dojdu v řetězci za místo, kde by byl hledaný prvek, můžu skončit).

Pro odhad složitosti alogritmů předpokládáme, že:

- Hashovací funkce h rozděluje data rovnoměrně
- Sama reprezentovaná množina S je náhodný výběr z U s rovnoměrným rozdělením

Tyto předpoklady v praxi ale splněny být nemusí.

Očekávaná průměrná délka řetězců

Pro odhad složitosti se počítá **očekávaná délka řetězců**. Označme délku i-tého řetězce jako $\mathbf{l}(i)$. Potom pravděpodobnost, že tento řetězec má délku l, odpovídá binomickému rozdělení:

$$P(\mathbf{l}(i) = l) = p_{n,l} = \binom{n}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{n-l}$$

Toto je jen aproximace (pro nekonečnou velikost univerza i seznamů), pro případ, že $N >> n^2 m$, ale lze použít. Očekávaná délka řetězce pak vychází jako (rozepíšu faktoriál a vytknu $\frac{n}{m}$, pak změním rozsah sumace $1 \ldots n$ (protože násobení $l = 0$ mi nic nedá), pak můžu z $l - 1$ udělat l a sumovat $0 \ldots n - 1$):

$$E(l) = \sum_{l=0}^{n} l p_{n,l} = \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-1-l} = \frac{n}{m}(\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m} = \alpha$$

Vlastně tu ale objevujeme Ameriku tím, že počítáme střední hodnotu binomicky rozdělené veličiny s parametrem $\frac{1}{m}$ – ze vzorce nám vyjde to samé. Stejně tak rozptyl ze vzorce vyjde $\frac{n}{m}\left(1-\frac{1}{m}\right)$.

Očekávaná délka nejdelšího řetězce

Tento údaj však sám o sobě nestačí, počítá se i očekávaný nejhorší případ (očekávaná **délka nejdelšího řetězce**). Ta se definuje následovně:

$$EMS = \sum_j j P(\max_i \mathbf{l}(i) = j) = \sum_j P(\max_i \mathbf{l}(i) \geq j)$$

Z toho (pravděpodobnost disjunkce jevů je $\leq$ součet jednotlivých pravděpodobností; vyčíslení: počet podmnožin správné velikosti a pravděpodobnost, že mají stejný hash):

$$P(\max_i \mathbf{l}(j) \geq j) \leq \sum_i P(\mathbf{l}(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j = \frac{\prod_{k=0}^{j-1}(n-k)}{j!}(\frac{1}{m})^{j-1} \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}$$

Najdeme mezní hodnotu j_0, pro které $n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1$. Označme $k_0 = \min\{k | n \leq k!\}$. Potom $j_0 \leq k_0$. Ze Stirlingovy formule plyne, že $\log x! = \Theta(x \log x)$. Z toho odvodíme (hodně neformálně, asymptoticky):

$$\log k_0! = k_0 \log k_0 = O(\log n)$$

$$\log k_0 + \log\log k_0 \approx \log k_0 = O(\log\log n)$$

$$k_0 = \frac{k_0 \log k_0}{\log k_0} = O(\frac{\log n}{\log\log n})$$

A $j_0 = O(k_0)$. Pro $\alpha \leq 1$ platí, že $EMS = O(j_0)$:

$$EMS = \sum_j P(\max_i \mathbf{l}(i) \geq j) \leq \sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty}(n(\frac{n}{m})^{j-1}\frac{1}{j!}) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = \cdots \leq j_0 + \frac{1}{j_0}$$

A tedy očekávaná délka nejdelšího řetězce je $O(\frac{\log n}{\log\log n})$.

Očekávaný počet testů

Testy jsou porovnání toho, co hledáme, s nějakým prvkem, nebo zjištění, že řetězec je prázdný. Jejich očekávaný počet je další odhad efektivity struktury. Rozlišujeme úspěšné a neúspěšné hledání.

Neúspěšné hledání (Je-li délka řetězce 0, jeden test stejně provedu, jinak otestuji celý řetězec):

$$E(T) = p_{n,0} + \sum_l l p_{n,l} = (1-\frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

S uspořádanými řetězci končím dřív ($e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1-e^{-\alpha})$).

Počet testů pro úspěšné vyhledávání je roven průměru počtu testů provedených při vložení každého z prvků, tj. 1 + očekávaná délka řetězce při každém vkládání: $\frac{1}{n}\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$.

Hashování s přemísťováním

Nevýhodou separovaných řetězců je nutnost alokovat další paměť, to je neefektivní. Proto zavedeme do hashovací tabulky pomocné ukazatele a celé řetězce nacpeme přímo do ní (a zřetězené prvky prostě rozházíme na jiné adresy). Pro hashování s přemísťováním se v tabulce uchovává navíc jednoduše odkaz na předchozí a následující prvek řetězce. Pokud vkládáme na místo, kde už je nějaký prvek z jiného řetězce, přehodíme tento cizí prvek jinam.

Algoritmy jsou téměř stejné jako pro separované řetězce, jen při **DELETE** prvního prvku řetězce je nutné na jeho místo přesunout druhý (pokud existuje).

Očekávaný počet testů je stejný jako pro hashování se separovanými řetězci. Přemísťování v tabulce je ale náročnější než 1 test, proto jsou **INSERT** a **DELETE** pomalejší.

Hashování se dvěma ukazateli

Od předchozího se liší tím, že místo ukazatele na předchozí prvek používá odkaz na začátek řetězce BEGIN. Řetězec tak už nemusí začínat na indexu svého hashe.

Místo přesouvání prvků algoritmy mění BEGIN (ten je na j-tém políčku vyplněn, právě když existuje řetězec prvků s hashem j).

- **INSERT** všechno vkládá na konec řetězce, zakládá-li nový, do BEGIN (na místě určené hashem) píše, kde se ve skutečnosti nachází
- **DELETE** jen upravuje odkazy na následující, nebo BEGIN (pokud maže poslední prvek řetězce).

Kvůli tomu, že řetězce začínají jinde než na svém místě, je počet testů o něco větší:

- Úspěšné hedání: $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$
- Neúspěšné hledání přibližně $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2+\alpha) - 2$.

Srůstající (coalesced) hashování

Srůstající hashování používá jen jeden ukazatel v hashovací tabulce navíc – odkaz na další prvek NEXT. Řetězce tak obsahují hodnoty s různými hashi. Prvek s vkládáme vždy do řetězce, obsahujícího $h(s)$-té políčko v tabulce.

Existují různé varianty:

- Standardní (bez pomocné paměti, "late" a "early" insertion) – LISCH, EISCH
- Bezpřívlastkové (s pomocnou pamětí, "late", "early" a "varied" insertion) — LICH, VICH, EICH.

Bez pomocné paměti – LISCH a EISCH

LISCH je "late insertion", tedy přidává se za poslední prvek řetězce. EISCH ("early insertion") přidává za první prvek řetězce.

- Algoritmus **MEMBER** je stejný pro oba (jen projití řetězce po odkazech NEXT).
- Alg. **INSERT**:
  - U LISCH projití celého řetězce (v případě že není prázdný, jinak jednoduše vložím na správné políčko) s testy na přítomnost prvku, potom vložení na libovolné volné místo v tabulce a připojení na konec řetězce.
  - Pro EISCH vložení na nějaké volné místo v tabulce a jen přepojení ukazatelů NEXT – připojení do řetězce za první prvek (pokud je řetězec neprázdný).

Algoritmy **DELETE** nejsou známy, kromě primitivních. Problémem je u nich zachování náhodného uspořádání prvků v řetězcích, které se předpokládá pro dodržení očekávaných časů operací. Je ale možné také prvky jen označit jako odstraněné a jejich místa použít při vkládání dalších (to ale zpomaluje hledání).

EISCH je kupodivu o něco rychlejší na úspěšné vyhledání (je větší pravděpodobnost práce s novým prvkem), očekávaný počet testů je stejný.

Počet testů v neúspěšném případě: Spočteme průměr přes všechny posloupnosti délky $n+1$ (kde hledáme $n+1$. prvek v množině ostatních n). Označme $c_{n,l}$ počet řetězců délky l, které přispívají celkem $1+2+\cdots+l = l + \binom{l}{2}$ porovnáními k sumě:

$$\frac{c_{n,0} + \sum_{l=1}^{n} l c_{n,l} + \sum_{l=1}^{n} \binom{l}{2} c_{n,l}}{m^{n+1}}$$

Tady $c_{n,0}$ představuje počet prázdných řádků, tedy $c_{n,0} = (m-n)m^n$, $lc_{n,l}$ je součet délek všech řetězců v reprezentacích n-prvk. množin, tedy $\sum_{l=1}^{n} l c_{n,l} = nm^n$.

Poslední člen označíme S_n. Při **INSERT**u do $n-1$-prvkové množiny jsou 2 možnosti vzniku řetězce délky l: buď přidávám do řetězce (délky $l-1$), nebo řetězec (pův. délky l) nezměním. Z toho vyjádříme rekurentní vztah pro S_n (úpravy: rozepsání rozdílů, vykrácení, rozpis l^2 jako $l^2 - l + l = \binom{l}{2} + l$):

$$S_n = \sum_l \binom{l}{2}(m-l)c_{n-1,l} + \sum_l \binom{l}{2}(l-1)c_{n-1,l-1} = mS_{n-1} - \sum_l l^2 c_{n-1,l} = (m+2)S_{n-1} + (n-1)m^{n-1})$$

Pak pomocí vztahu $T_n^c = \sum_{i=1}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}$ spočítaného z $(c-1)T_n^c = nc^{n+1} + (\sum_{i=2}^{n} - c^i) - c$ získáme nerekurentní vztah ($S_0 = 0$, obrácení sumace a vytknutí $m + 2^{n-1}$):

$$S_n = (m+2)^{n-1}S_0 + \sum_{i=0}^{n-1}(m+2)^i(n-1-i)m^{n-1-i} = (m+2)^{n-1}\sum_{i=1}^{n-1} i\left(\frac{m}{m+2}\right)^i = \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n$$

A tedy očekávaný počet testů vyjde:

$$1 + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$$

Počet testů pro úspěšný případ spočteme pro LISCH jako počet testů při vkládání prvku. Metoda EISCH pro tento postup nesplňuje předpoklady. Porovnání klíčů při neúspěšném vyhledávání je stejně při přístupu na obsazené políčko, neporovnávám ale nic při přístupu na neobsazené políčko, takže dostávám:

$$\frac{n}{m} + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) = \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 + \frac{2n}{m}\right)$$

Průměr pro postupné vkládání všech prvků pak dává:

$$1 + \sum_{i=0}^{n-1}\frac{1}{4}\left(\left(1+\frac{2}{m}\right)^i - 1 + \frac{2i}{m}\right) = 1 + \frac{m}{8n}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) + \frac{n-1}{4m} \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

Pro metodu EISCH vychází (bez důkazu):

$$\frac{m}{n}\left(\left(1+\frac{1}{m}\right)n - 1\right) \approx \frac{1}{\alpha}(e^{\alpha} - 1)$$

Všechny odhady mají odchylku $O(\frac{1}{m})$.

S pomocnou pamětí – LICH, VICH, EICH

V této variantě rozdělíme paměť na dvě části:

- (hash-funkcí) přímo adresovatelná
- pomocná část (bez přístupu hash-funkcí)

Při kolizích nejdříve ukládáme do řádků z pomocné části, pak teprve do přímo adresovatelné, tedy oddalujeme srůstání řetězců. Chování se tak až do určitého okamžiku podobá separovaným.

Existují tři varianty podle chování algoritmu **INSERT**:

- LICH vždy přidává na konec řetězce
- EICH v případě neprázdného řetězce vždy za 1. prvek
- VICH vždy za poslední prvek v pomocné paměti nebo (pokud žádné v pomocné paměti nejsou) za 1. prvek řetězce (tj. chová se na pomocné paměti jako LICH a v přímo adresovatelné části jako EICH).

Algoritmy až na VICH se chovají stejně jako ve standardním srůstajícím hashování, rozhodující je výběr volného řádku pro vložení: např. "vždy vyber z nejvyšší adresy" může zaručit používání pomocné paměti. Také tu není přirozené efektivní **DELETE**.

Odhad složitosti: definujeme si následující hodnoty:

- n – počet uložených prvků
- m – velikost přímo adresovatelné paměti
- m' – celková velikost paměti
- $\alpha = \frac{n}{m'}$ – faktor zaplnění
- $\beta = \frac{m}{m'}$ – adresovací faktor
- λ – jediné nezáp. řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$

Pokud je $\alpha \leq \lambda\beta$, pak pro všechny verze vychází očekávaný počet testů $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$ v neúspěšném případě a $1 + \frac{\alpha}{2\beta}$ v úspěšném (chyba: $O(\log \frac{m'}{\sqrt{m'}})$.

V případě, že $\alpha \geq \lambda\beta$ (začínají srůstat řetězce), se metody liší a vychází divnosti. V neúspěšném případě je VICH a LICH lepší než EICH, v úspěšném vede VICH před EICH a LICH (vždy o jednotky procent). Doporučená hodnota $\beta = 0.86$. Na hledání volného řádku se v praxi hodí např. spojový seznam volných řádků.

Lineární přidávání

Tato a následující metoda nepoužívá žádné dodatečné položky v hashovací tabulce a zároveň řetězce kolidujících prvků ukládá přímo do ní. Nalezení dalšího prvku z řetězce je přímo v algoritmu.

Lineární přidávání je nejjednodušším řešením takové situace: v případě kolize při **INSERT**u nalezne nejbližší vyšší volné políčko a vloží nový prvek tam. Předpokládáme "cyklickou" paměť, tj. když dojdeme na konec, vkládáme od začátku.

Problémem je tvoření shluků – při velkém zaplnění se operace dost zpomalují. Také nepodporuje efektivní DELETE (a ani primitivní způsoby nejsou moc rychlé). V praxi je dobré uchovávat počet uložených prvků nebo mít zarážku (nikdy neobsazované pole), abychom věděli, kdy dojde k přeplnění.

Očekávaný počet testů je $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ v neúspěšném a $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)\right)$ v úspěšném případě (bez důkazu).

Dvojité hashování

Dvojité hashování je vylepšení předchozí metody tak, aby nevznikaly shluky. Výběr následujícího řádku bude závislý na předchozím, ale s rovnoměrným rozložením. Na to použiji druhou hashovací funkci h_2.

Při operacích **INSERT** pak hledám nejmenší i od 0, že $(h_1(x)+i\cdot h_2(x)) \mod m$ je volné políčko, tj. postupně přičítám $h_2(x)$ a modulím. Stejný postup je i pro operaci **MEMBER**.

Je nutné, aby $h_2(x) \not| m$, tj. abych měl prosté posloupnosti (a z každého políčka tak mohl vést řetězec po celé tabulce). Idea, že iterace h_2 tvoří pro každé x náhodnou permutaci paměťových míst, není úplně přesná, ale v praxi stačí, aby z $h_1(x)=h_1(y)$ plynulo, že $h_2(x)$ a $h_2(y)$ budou odlišné.

Funkce navíc musíme volit "chytře" (i lineární přidávání je spec. příp. dvojitého hashování, kdy $h_2\equiv 1$). Pak tato metoda je znatelně rychlejší než lin. přidávání. Předpoklad náhodnosti použitý v teoretické analýze sice splnit nelze, ale přiblížit se mu ano.

Očekávaný počet testů

Předpokládáme, že iterování funkce h_2 tvoří náhodné permutace (což, jak bylo řečeno, není úplně přesné).

Pro neúspěšný případ: označme $q_i(n,m)$ pravděpodobnost, že při zaplnění $\frac{n}{m}$ je pro nějaké x prvních $i-1$ políček, kam bych ho mohl vložit, plných. Potom $q_i(n,m)=\frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}$ a tedy $q_i(n,m)=\frac{n}{m}q_{i-1}(n-1,m-1)$.

Očekávaný počet testů je (předposlední rovnost plyne z rekurentního vztahu pro q_j, poslední krok dokázat indukcí):

$$C(n,m)=\sum_{j=0}^{n}(j+1)(q_j(n,m)-q_{j+1}(n,m))=\sum_{j=0}^{n}(q_j(n,m))=1+\frac{n}{m}C(n-1,m-1)=\frac{m+1}{m-n+1}$$

Počet testů v úspěšném případě – stejná metoda jako u dřívějších analýz, takže vychází:

$$\frac{1}{n}\sum_{i=0}^{n-1}C(i,m)=\frac{1}{n}\sum_{i=0}^{n}n-1\frac{m+1}{m-i+1}\approx\frac{1}{\alpha}\ln\left(\frac{m+1}{m-n+1}\right)\approx\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

Srovnání

Podle počtu testů:

	neúspěšné	**úspěšné**
1.	separované uspořádané řetězce	separované (usp. i neusp.) řetězce, přemísťování
2.	separované řetězce, přemísťování	dva ukazatele
3.	dva ukazatele	VICH
4.	VICH, LICH	LICH
5.	EICH	EICH
6.	LISCH, EISCH	EISCH
7.	dvojité hashování	LISCH
8.	lineární přidávání	dvojité hashování
9.		lineární přidávání

- VICH je při vhodném α lepší než hashování se dvěma ukazateli.
- Lineární přidávání se nedá použít pro $\alpha>0.7$, dvojité hashování pro $\alpha>0.9$.
- Separované řetězce a obecné srůstající hashování používají víc paměti, přemísťování a dvojité hashování zas víc času, tj. nelze říct, které je jednoznačně lepší.

Implementační dodatky

- Pro hledání volných řádků se většinou používá seznam (zásobník).
- Přeplnění se většinou řeší držením α v rozumném intervalu ($\langle 1/4, 1\rangle$) a přehashováním do jinak velké tabulky ($2^i \cdot m$) při pře- nebo podtečení
- V praxi se doporučuje přehashování odkládat (např. pomocnými tabulkami) a provádět při nečinnosti systému.

DELETE se ve strukturách, které ho nepodporují, řeší označením políčka jako smazaného s možností využití při vkládání. V případě, že polovina polí je blokovaná tímto způsobem, se vše přehashuje. Pro srůstající hashování se toto používat nemusí, máme metody na zachování náhodnosti rozdělení dat.

V praxi je výhodné, známe-li něco o rozdělení vstupních dat, aby ho hashovací funkce kopírovala (většinou to ale nejde), jinak musíme předpokládat rovnoměrnost, což zaručeno zdaleka není. Nutnost rovnoměrného rozdělení vstupních dat lze obejít (viz níže).

11.2 Univerzální hashování

Abychom nemuseli předpokládat rovnoměrné rozložení vstupních hashovaných dat (což zdaleka není vždy zaručeno), budeme mít místo pevné hash-funkce nějakou množinu H, z níž funkci náhodně rovnoměrně vyberu. Analýza složitostí se pak dělá přes všechny $h \in H$ a platí pro všechny $S \subset U$ – i nerovnoměrné (S je daná pevně a h se k ní volí; $|U| = N$).

Pro formalizaci analýz je nutné mít analytické zadání funkcí h a znát přesnou velikost množiny $|H|$. To obejdeme očíslováním funkcí $H = \{h_i; i \in I\}$ a počítáním s indexovou množinou (očekávaná hodnota je průměr přes I). Při použití skutečné velikosti H v odhadech bychom dostali horší výsledky, protože některé funkce s různými indexy se můžou ve skutečnosti ukázat jako identické, a to tu zanedbáváme.

Definice: Systém funkcí $H = \{h_i; i \in I\} : U \to \{0, \ldots, m-1\}$ je c-univerzální, pokud:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \tfrac{c|I|}{m}.$$

Tj. zaručuje se, že pro každé dva různé prvky má maximálně c funkcí kolizi.

Existence c-univerzálních systémů

Předpokládejme, že universum má tvar $U = \{0, 1, \ldots, N-1\}$ kde N je nějaké prvočíslo a vezmeme funkce typu

$$h_{a,b}(x) = ((ax + b) \mod N) \mod m$$

Jsou dobře použitelné, protože se dají počítat rychle. Protože N je prvočíslo, můžeme pracovat v $\mathbb{Z}_N$, coz je těleso. Rovnice $h_{a,b}(x) = h_{a,b}(y)$ je ekvivalentní s:

$$\exists i \in \{0, \ldots, m-1\} \wedge \exists r, s \in \{0, \ldots, \lceil \frac{N}{m} \rceil - 1\} : (ax + b \equiv i + rm) \mod N \wedge (ay + b \equiv i + sm) \mod N$$

Z Frobeniovy věty o jednoznačnosti řešení lineárních rovnic plyne, že pro každé r, s, i existuje jen jedna dvojice a, b , které vyhovuje. Počet řešení soustavy je tedy omezený číslem $m \cdot \lceil \frac{N}{m} \rceil^2$ (i – m hodnot, r, s – $\lceil \frac{N}{m} \rceil$ hodnot pro daná x, y).

Pak je systém c-univerzální pro $c = \left(\lceil \frac{N}{m} \rceil\right)^2 / \left(\frac{N}{m}\right)^2$ a jeho velikost opdovídá N^2.

Vlastnosti

Vyrobíme si pomocnou funkci

$$\delta_i(x, y) = \begin{cases} 1 \text{ pro } h_i(x) = h_i(y), x \neq y \\ 0 \text{ jinak} \end{cases}$$

Chceme potom spočítat součet $\delta_i(x, S) = \sum_{y \in S} \delta_i(x, y)$. Z výsledku vidíme očekávanou délku řetězce pro libovolnou (jednu) množinu dat. Tohle pak sečtu přes všechny mé hash-funkce a z c-univerzality dostanu

$$\sum_{i \in I} \delta_i(x, S) = \sum_{y \in S} \sum_{i \in I} \delta_i(x, y) \leq \sum_{y \in S, x \neq y} c\frac{|I|}{m} = \begin{cases} (|S| - 1)c\frac{|I|}{m} & \text{pro } x \in S \\ |S|c\frac{|I|}{m} & \text{jinak} \end{cases}$$

Z toho dopočítám (podělením $|I|$) horní odhad očekávaného $\delta_i(x, S)$.

Výsledek: očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v c-univerzálním hashování je $O(1 + c\alpha)$ (kde faktor naplnění $\alpha = \frac{|S|}{m}$). Čas n po sobě jdoucích operací na původně prázdné tabulce je $O(n(1 + \frac{c}{2}\alpha))$. To není lepší hodnota než mají separované řetězce ($O(1 + \alpha)$), ale u nich předpokládám rovnoměrné rozdělení dat.

Výběr vhodné funkce není úplně jednoduchý, protože funkcí může celkem být např až N^2, tj. nelze ho provést jednoduchým zavoláním generátoru náhodných čísel, nýbrž např. náhodným vybráním každého bitu indexu funkce. Proto je výhodné najít co nejmenší c-univerzální systémy (viz dále).

Dolní odhady velikosti univ. systémů

Očíslujme hash-funkce z I a induktivně definujme množiny U_i jako největší podmnožiny U_{i-1} takové, že $h_{i-1}(U_i)$ je jednoprvková. Platí $|U_i| \geq \lceil \frac{U_{i-1}}{m} \rceil$, tedy $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ – velikost těchto množin klesá s logaritmem a $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$. Takže velikost univ. systému roste alespoň úměrně logaritmu velikosti univerza.

Dolní odhad c

5-univerzální systém: Zvolme $t \in \mathbb{N}$ a k němu vezměme t-té prvočíslo p_t tak, že $t \ln p_t \geq m \ln N$. Definujme systém funkcí $H = \{g_{c,d,l}(x) | t < l \leq 2t, c, d \in \{0, 1, \ldots, p_{2t}-1\}\}$ kde $((c(x \mod p_l)+d) \mod p_{2t}) \mod m$. Zřejmě $|H| = tp_{2t}^2$.

Odhadem $|G = \{(c,d,l); h_{c,d,l}(x) = h_{c,d,l}(y)\}|$, když si množinu rozdělíme na $G_1 = \{(c,d,l) \in G; x \mod p_l \neq y \mod p_l\}$ a $G_2 = \{(c,d,l) \in G; x \mod p_l = y \mod p_l\}$, se dá dokázat, že systém je 5-univerzální, za dalších podmínek i 3.25-univerzální.

Dolní odhad c: Platí: $c > 1 - \frac{m}{N}$. Spočítáme $\sum_{h \in H} \sum_{x,y \in U} \delta_h(x,y)$ – pro pevnou h máme (z Cauchy-Schwarzovy nerovnosti, $u_{i,h} = |\{x \in U, h(x) = i\}|$):

$$\sum_{x,y \in U} \delta_h(x,y) = \sum_{i=0}^{m-1} u_{i,h}(u_{i,h} - 1) \geq \frac{(\sum_{i=0}^{m-1} u_{i,h})^2}{m} - N = \frac{N^2}{m} - N$$

Tedy $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \geq \frac{|H|N(N-m)}{m}$. Zároveň platí $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \leq \sum_{x,y \in U} c\frac{|H|}{m} = N^2 c\frac{|H|}{m}$, což mi dává výsledek.

11.3 Perfektní hashování

Základní ideou perfektního hashování je nalézt hash-funkci, která pro danou množinu S nedělá žádné kolize, takže operace **MEMBER** bude velice rychlá. Potom je nevýhoda, že nelze přirozeným způsobem realizovat operaci **INSERT**, tj. v praxi se nesmí moc často vyskytovat.

Tabulka by neměla být o mnoho větší než množina S a funkce h rychle spočitatelná a její realizace nezabírat moc paměti (tedy žádná zadávání tabulkou).

Definice

- Hashovací funkce h je perfektní pro množinu S, pokud pro $\forall x, y \in S, x \neq y : h(x) \neq h(y)$
- Soubor funkcí $H : U \to \{0, \ldots, m-1\}$ je (N, m, n)-perfektní, pokud $\forall S \subseteq U$ takové, že $|S| = n$ existuje $h \in H$ perfektní pro S.

Odhady velikosti

Každá funkce h je perfektní pro $\sum\{\prod_{j=0}^{n-1} |h^{-1}(i_j)|; 0 \leq i_0 < \cdots < i_{n-1} < m\}$ množin (sčítáme přes všechny množiny hashů $h(S)$ a pro každou z nich uvažujeme všechny možnosti, jak mohla vzniknout). Z Cauchy-Schwarzovy nerovnosti plyne, že tento výraz nabývá maxima, když $\forall i : |h^{-1}(i)| = \frac{N}{m}$. Každá funkce je tedy perfektní pro max. $\binom{m}{n}(\frac{N}{m})^n$ množin. Z toho plyne:

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}\left(\frac{N}{m}\right)^n}$$

Jiný odhad lze provést jako u c-univ. systémů s očíslovanými funkcemi $|H| = \{h_1, \ldots, h_t\}$. Používám induktivně definované množiny U_i, kde $U_0 = U$ a U_i je největší podmnožina U_{i-1}, kde je zrovna funkce h_i konstantní. Dostáváme $|U_i| \geq \frac{|U_{i-1}|}{m}$, tj. $|U_t| \geq \frac{N}{m^t}$, ale z perfektnosti plyne $|U_t| \leq 1$. Dostáváme $t \geq \frac{logN}{logm}$.

Existence

Reprezentujme soubor funkcí $H = \{h_1, \ldots, h_t\}$ na univerzu velikosti N pomocí matice $M(H)$ typu $N \times t$, takže $M(H)_{x,i} = h_i(x)$, tj. v jednom sloupci jsou výsledky jedné hashovací funkce pro všechny prvky univerza.

Pak žádná funkce z H není perfektní pro množinu $S \subset U$, když podmatice $M(H)$ tvořená řádky příslušejícími prvkům S nemá prostý sloupec. Takových matic je maximálně (počet všech funkcí minus počet prostých, to celé krát libovolné doplnění na N řádek):

$$\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t}$$

Podmnožin U velikosti n je pak $\binom{N}{n}$, čímž vynásobeno mám počet matic neodpovídajících (N,m,n)-perfektnímu systému. Všech matic je m^{Nt}. Potom existuje (N,m,n)-perfektní systém, když:

$$\binom{N}{n}\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t} < m^{Nt}$$

Přišernými kejklemi dostaneme podmínku existence $t \geq n(\ln N)e^{\frac{n^2}{m}}$.

Konstrukce funkce

Chceme splnit rychlou spočitatelnost a paměťovou nenáročnost. Předpokládáme univerzum prvočíselné velikosti a funkce typu:

$$h_k(x) = (kx \mod N) \mod m$$

Označme $b_i^k = |\{x \in S; (kx \mod N) \mod m = i\}|$. Potom pokud $h_k(x)$ není perfektní, pak nějaké $b_i^k = 2$ a mám $\sum_{i=0}^{m-1}(b_i^k)^2 \geq n+2$.

Odhadnu výraz $\sum_{k=1}^{N-1}((\sum_{i=0}^{m-1}(b_i^k)^2) - n) = \sum_{x\neq y\in S}\{k; 1 \leq k < N, h_k(x) = h_k(y)\}$. Z vlastností modula mám takových k pro daná x,y nejvýše $2\lfloor\frac{N}{m}\rfloor = 2\lfloor\frac{N-1}{m}\rfloor$. Dostávám tedy odhad $2(N-1)\frac{n(n-1)}{m}$ a z něj vidím, že existuje takové k, že $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$, tedy pro tabulku velikosti $m > n(n-1)$ mám perfektní funkci.

Dá se dokázat trochu slabší předpoklad, že $P(k; \sum_{i=0}^{m-1}(b_i^k)^2 < \frac{3n(n-1)}{m} + n) \geq 1/4$, který je základem pravděpodobnostního algoritmu.

Pak mám deterministický algoritmus, který pro $m = n(n-1)+1$ nalezne perfektní h_k v čase $O(nN)$ a pravděpodobnostní, který pro $m = 2n(n-1)$ najde perfektní h_k v čase $O(n)$. Mám tedy konstrukci perfektní hash-funkce, ta ale nesplňuje požadavek na malou tabulku ($m = \Theta(n^2)$).

Menší tabulka

Zmenšíme-li velikost tabulky na $m = n$, bude výše uvedený algoritmus schopný nalézt funkci, pro kterou platí $\sum(b_i^k)^2 < 3n$ ($\sum(b_i^k)^2 < 4n$ v pravděpodobnostní variantě). Každou kolizi pak můžeme "rozstrkat" perfektní funkcí nad miniaturní tabulkou a celková velikost všech tabulek bude mnohem menší:

- Vezmeme nalezenou funkci a najdeme všechny neprázdné množiny $S_i = \{s \in S; h_k(s) = i\}$
- Pro jim odpovídající $c_i = |S_i|(|S_i|-1)+1$ (dvojnásobek v pravděp. metodě) najdeme k_i takové, že h_{k_i} je perfektní funkce pro S_i do tabulky velikosti c_i.
- Definujme $d_i = \sum_{j=0}^{i-1} c_j$, potom pokud $h_k(x) = l$, pak $g(x) = d_l + h_{kl}(x)$ je perfektní, její hodnota spočitatelná v čase $O(1)$ a hashuje do tabulky velikosti $O(3n)$ ($O(6n)$ s pravděp. případě), je naleznutelná v čase $O(nN)$ ($O(n)$) a pro její uložení do paměti jsou potřeba hodnoty k a k_i, vyžadující $O(n \log N)$ paměti.

Pro výpočet $g(x)$ potřebuji 2 násobení, 2 modulo a 1 sčítání (pro d_i v paměti), tabulka má velikost $\sum c_i \leq \sum(b_i^k)^2 < 3n$. Taková funkce ale stále nesplňuje požadavek na málo paměti pro uložení.

"Malá" funkce

Víme, že pro $m \in \mathbb{N}$ je počet prvočísel, která ho dělí $O(\frac{\log m}{\log\log m})$. Z toho úvahou o dělitelích čísla $D = \prod_{1\leq i<j\leq n}(s_j - s_i) \leq N^{n^2}$ na n-prvkové S a vzorce hustoty prvočísel $p_t \leq 2t\ln t$ dostanu, že existuje p o velikosti $O(\ln D) = O(n^2 \ln N)$ takové, že $\phi_p(x) = x \mod p$ je perfektní pro S.

Deterministické nalezení trvá $O(n^3 \log n \log N)$ (test perfektnosti každého systému je $O(n\log n)$). Proto použijeme pravděpodobnostní algoritmus (mezi $4cn^2\ln N$ přir. čísly je aspoň 1/2 prvočísel, která vyhovují): nejdřív najde prvočíslo a pak testuje perfektnost. Očekávaný počet testů je $O(\ln(4cn^2\ln N))$, celková složitost algoritmu je pak $O(n\log n(\log n + \log\log N))$. Najde zhruba až 2× větší prvočíslo než deterministický.

Tuto funkci použijeme ke konstrukci výsledné hash-funkce:

1. Nalezneme prvočíslo q_0, aby $\phi_{q_0}(x) = x \mod q_0$ byla perfektní pro S
2. Položíme $S_1 = \{\phi_{q_0}(s) | s \in S\}$, pak najdeme prvočíslo $q_1 \in \langle n(n-1)+1, 2n(n-1)+2\rangle$
3. K němu existuje $l \in \langle 1, q_0 - 1\rangle$ takové, že $h_l(x) = ((lx \mod q_0) \mod q_1)$ je perfektní pro S_1
4. Položíme $S_2 = \{h_l(s) | s \in S_1\}$ a najdeme perfektní g pro S_2 do tabulky s méně než $3n$ řádky (viz výše, počítá se ale do univerza o velikosti q_1).
5. Pak $f(x) = g(h_l(\phi_{q_0}(x)))$ je perfektní.

Funkce q_0 je určena 1 číslem o velikosti $O(n^2\log N)$, h_l 2 čísly o velikosti $O(n^2)$ a $O(q_0)$, g je určená $n+1$ čísly o $O(q_1)$, tj. celkem zadání vyžaduje $O(n\log n + \log n + \log\log N)$ paměti.

Kapitola 12

Státnice - Dynamizace datových struktur

12.1 Úvod

Mnoho datových struktur podporuje velice efektivní operaci **MEMBER**, ale nemá operace **INSERT** a **DELETE**. Úkolem dynamizace je právě tyto operace do jakékoliv obecné struktury co nejefektivněji doplnit.

Zobecněný vyhledávací problém

Zobecněný vyhledávací problém je libovolná funkce $f : U_1 \times 2^{U_2} \to U_3$ – pro prvek a nějakou množinu vrací nějakou hodnotu.

Struktura $\mathcal{S}$ je statická struktura řešící vyhledávací problém (neboli S-struktura) f, je-li dán algoritmus, který pro $A \subseteq U_2$ zkonstruuje datovou strukturu $\mathcal{S}(A)$ a algoritmus, který pro $x \in U_1$ a $\mathcal{S}(A)$ spočte $f(x, A)$.

Struktura je semidynamická, pokud navíc existuje algoritmus, který pro $x \in U_2$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \cup \{x\})$.

Struktura je dynamická, pokud navíc existuje algoritmus, který pro $x \in A$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \backslash \{x\})$.

Ve všech následujících operacích budeme uvažovat jen rozložitelný vyhledávací problém:

- Vyhledávací problém je rozložitelný, pokud existuje binární operace $\circ$ na univerzu U_3 taková, že pro disjunktní $A, B \subseteq U_2$ a $x \in U_1$ platí $f(x, A) \circ f(x, B) = f(x, A \cup B)$.

To platí pro situaci, která nás nejvíc zajímá – pokud funkce f provádí operaci **MEMBER** nad nějakou datovou strukturou. Jiné problémy (patří x do konvexního obalu množiny?) ale rozložitelné nejsou.

Navíc budeme předpokládat, že funkce $\circ$ lze spočítat v konstantním čase. Označme

- $Q_{\mathcal{S}}(n)$ čas na vyčíslení $f(x, A)$, pokud $|A| = n$
- $S_{\mathcal{S}}(n)$ paměťový prostor potřebný k reprezentaci n-prvkové množiny
- $P_{\mathcal{S}}(n)$ čas na vytvoření struktury $\mathcal{S}(A)$ pro $|A| = n$

Pro asymptotické odhady budeme předpokládat, že funkce $Q_{\mathcal{S}}(n)$, $\frac{S_{\mathcal{S}}(n)}{n}$ a $\frac{P_{\mathcal{S}}(n)}{n}$ jsou neklesající.

12.2 Semidynamizace

Vytvoříme semidynamickou strukturu, jejíž čas operace **INSERT** bude velmi rychlý a navíc se nám moc nepokazí doba pro provedení operace **MEMBER**. Základní idea je podobná jako v binomiálních haldách – roztrháme reprezentovanou množinu A na sadu disjunktních podmnožin A_i o velikosti 2^i (takové množiny jsou neprázdné pro i odpovídající jedničkám v binárním zápisu čísla $|A|$.

Formálně budeme mít spojový seznam S-struktur $\mathcal{S}(A_i)$ odpovídajících množinám A_i, potom seznam spojových seznamů $s(A_i)$, které obsahují prvky jednotlivých množin, a nakonec pomocnou dynamickou strukturu T (např. binární vyhledávací strom). To všechno dohromady zjevně vyžaduje jen $O(S_{\mathcal{S}}(n))$ paměti (celkem reprezentujeme pořád stejně prvků a vyhledávací stromy i spoják jsou $O(n)$).

Struktura T se používá k "rychlému" testování před **INSERT**em, zda nechceme vkládat prvek, který už v množině máme, případně před **DELETEM**, jestli nechceme mazat neexistující prvek. Pro jednoduchost to v popisech operací vynecháme.

Operace MEMBER

Operace **MEMBER** potom projde všechny neprázdné množiny A_i a zavolá **MEMBER** na nich. Výsledek spojí za pomoci funkce $\circ$.

To celkem dává složitost **MEMBER**u $O(Q_{\mathcal{S}}(n)\log(n))$, protože hledáme v max. $\log_2$ dílčích množinách.

Protože jakékoliv mocniny rostou rychleji než logaritmus, platí, že pokud $Q_{\mathcal{S}} = n^\epsilon$ pro nějaké $\epsilon > 0$, pak je operace **MEMBER** $O(n^\epsilon)$.

Operace INSERT

Operace **INSERT** najde nejmenší i takové, že $A_i = \emptyset$. Potom vezme všechny $A_j, j < i$ (ty jsou neprázdné) a spolu s vkládaným prvkem je slije do jedné množiny A_i (zahodí S-struktury pro všechna $A_j, j < i$, zkonkatenuje spojáky $s(A_j)$ a na jejich základě zkonstruuje novou S-strukturu $\mathcal{S}(A_i)$). Nová množina má správný počet prvků, protože $\sum_{j=0}^{i-1} 2^j + 1 = 2^i$.

V odhadu amortizované složitosti využijeme fakt, že S-struktura pro A_i se tvoří jen tehdy, když existují S-struktury pro všechny $A_j, j < i$. Tj. S-struktura o velikosti 2^i prvků se konstruuje znovu až po dalších $2^{i+1} - 1$ **INSERT**ech. Amortizovaně tak vychází :

$$\sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(2^i)}{2^i} \leq \sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(n)}{n} = \frac{P_{\mathcal{S}}(n)}{n} \log n = O(\frac{P_{\mathcal{S}}(n)}{n} \log n).$$

Podobně jako pro **MEMBER** platí, že pokud $P_{\mathcal{S}} = n^\epsilon$ pro nějaké $\epsilon > 1$, pak je složitost operace **INSERT** $O(n^{\epsilon-1})$.

Operace INSERT se zaručeným nejhorším časem

Protože někdy nestačí mít **INSERT** rychlý amortizovaně, byla vytvořena jeho varianta, která zaručuje rozumnou rychlost i v nejhorším případě. Dělá se to prostým rozložením potřebných operací do všech volání. Musíme ale povolit i požadavky na naši strukturu – místo max. jedné množiny pro každou velikost odpovídající mocnině dvou budeme mít max. 4 takové množiny $A_{i,0}, \ldots A_{i,3}$, z nichž poslední navíc bývá zkonstruovaná jen částečně ("rozpracovaná").

Označme počet množin A_i jako k_i, přičemž $k_i = -1$, pokud neexistuje žádná množina velikosti 2^i. Algoritmus potom vypadá následovně:

1. Vytvoří se jednoprvková $A_{0,k_0+1} = \{x\}$ a zvýší k_0 (tj. i odpovídající S-struktura)
2. Pro první souvislý úsek, kde $k_i \geq 0$:
   (a) Provedeme dalších $\frac{P_{\mathcal{S}}(2^i)}{2^i}$ konstrukce S-struktury $\mathcal{S}(A_i, k_i)$
   (b) Pokud jsme tím tuto S-strukturu dotvořili, vezmeme první dvě struktury "o patro níž" ($A_{i-1,0}$ a $A_{i-1,1}$) a připravíme je prvním krokem k sloučení. Zároveň je odtud odebereme a přečíslujeme zbylé. Jejich započaté sloučení přidáme na úroveň i.
3. Pokud jsme v posledním kroku (i, t.ž. $k_{i+1} = 0$) vytvořili druhou strukturu stejné velikosti na nejvyšším dosaženém "patře", připravíme první krok sloučení těchto dvou největších struktur, přemístíme jejich započaté sloučení "o patro výš" a odebereme původní struktury.

Díky tomu, že počet kroků volíme tak šikovně, vždycky dotvoříme strukturu právě včas na to, abychom mohli vytvářet další stejné velikosti. Navíc, protože se provede jen $O(\frac{P_{\mathcal{S}}(2^i)}{2^i})$ kroků pro každé i, odpovídá složitost v nejhorším případě amortizované složitosti původní operace **INSERT**.

12.3 Dynamizace

Při úplné dynamizaci navíc požadujeme efektivní algoritmus **DELETE**. Přidáme další předpoklad, a to, že na naší původní S-struktuře existuje operace **FAKE-DELETE** (ta spočívá v označení nějakého prvku jako "smazaného", aniž by se smazal skutečně – dál tedy zabírá místo a prodlužuje operace).

Základní ideou oproti semidynamizaci navíc je, že se budeme snažit za každou cenu zajistit, aby všechny naše pomocné S-struktury stále obsahovaly alespoň osminu "nesmazaných" prvků a přestavovat je až poté, co počet "smazaných" překročí tuto mez. Formálně naše S-struktury reprezentují vlastně množiny B_i, kde $B_i = A_i \backslash \{x_i; i = 1, 2 \ldots k\}$ (a $2^{i-3} < |A_i| \leq 2^j$).

Asymptoticky tak zaručíme, že množin (a pomocných S-struktur) bude stále jen logaritmicky mnoho a paměťové nároky taky zůstávají asymptoticky stejné. Struktura je tedy stejná jako pro (základní) dynamizaci. Navíc jsou nyní spojové seznamy prvků přímo provázané s S-strukturami (protože už nepoznáme podle velikosti seznamu, ke které S-struktuře vlastně patří).

Operace **MEMBER** pak pracuje úplně stejně, jen nakonec zkontroluje, zda nalezený prvek nebyl označen jako "smazaný" (a případně ho pak nevrátí).

Operace INSERT

Algoritmus operace **INSERT**(x) je následující:

1. Najdi nejmenší j takové, že $|\cup_{i\leq j} A_i| < 2^j$.
2. Vytvoříme $A_j = \cup_{i\leq j} A_i \cup \{x\}$ (včetně S-struktury a spojáku). To splňuje požadavky na velikost $2^{j-1} < |A_j| \leq 2^j$, jinak by pro j nebylo nejmenší.
3. Položíme $A_i = \emptyset$ pro $i < j$.

Díky zachovávané minimální velikosti vytvářené množiny máme složitost amortizovaně stejnou jako v případě semidynamizace.

Operace DELETE

Odebrání prvku x z naší dynamické struktury vypadá následovně:

1. Odstraníme x ze spojáku
2. Vyřešíme čtyři různé případy podle velikosti množiny A_i, která obsahuje prvek x:
   (a) Je-li A_i jen jednoprvková, prostě zahodím struktury s ní spojené.
   (b) Je-li $|A_i| > 2^{i-3}$ (tj. ještě mám víc než osminu prvků "platných"), provedeme pouze **FAKE-DELETE**.
   (c) Je-li A_{i-1} buď prázdná, nebo dost velká ($|A_{i-1}| > 2^{i-2}$), můžeme ji s A_i prohodit. Pro "nové" A_{i-1} pak vytvoříme novou S-strukturu.
   (d) Je-li A_{i-1} neprázdná, ale moc malá na prohození s A_i, potom je určitě můžeme sloučit a získáme novou množinu, která splňuje velikostní omezení pro A_i. Pro ni vyrobíme novou S-strukturu.

Amortizovaná složitost celé operace **DELETE** je $O(D_{\mathcal{S}}(n) + \log n + \frac{P_{\mathcal{S}}(n)}{n})$, kde $D_{\mathcal{S}}(n)$ je čas potřebný na operaci **FAKE-DELETE** a $\log n$ je potřeba na vyhledání prvku. Odhadneme, kolikrát se dělá skutečné přestavění S-struktur. Pokud $x \in A_i$, pak **DELETE** může nově vytvořit jen $S(A_i)$ nebo $S(A_{i-1})$. Z jejich podmínek velikosti (v algoritmu) vidíme, že mezi dvěma **DELETE** pro stejné A_j (přičemž $|A_j| \leq 2^{i-2}$) se musí provést aspoň 2^{j-3}-krát **FAKE-DELETE** (a libovolný počet **DELETE** na jiných množinách). Amortizovaná složitost vytváření S-struktur tak vychází:

$$\frac{P_{\mathcal{S}}(2^{j-2})}{2^{j-3}} = O(\frac{P_{\mathcal{S}}(n)}{n})$$

I se současně prováděnými operacemi **DELETE** se amortizovaná složitost operací **INSERT** zachovává. Aby **INSERT** vytvořil podruhé strukturu pro A_i, musí opět nastat $1 + \sum_{j\leq i} |A_j| > 2^{j-1}$. Protože **DELETE** nikdy nevytvoří strukturu s víc než polovinou skutečně zaplněnou, musí se do té doby provést aspoň 2^{i-2} **INSERT**ů, čímž získáme stejnou situaci jako u semidynamizace.

Kapitola 13

Státnice - Datové struktury ve vnější paměti

13.1 Základní pojmy

- Logickou jednotkou dat je záznam, který má atributy se jmény a doménami (uvažujeme max. jednu hodnotu pro každý atribut).
- (Homogenní) soubor je kolekce (multimnožina) záznamů. Na souboru jsou definovány operace **INSERT**, **DELETE**, **UPDATE** a **FETCH**.
- Pro soubory s neopakujícími se záznamy je klíč množina atributů, které záznam jednoznačně identifikují v souboru. Jeden nich z klíčů se označuje jako primární.
- Vyhledávací klíč je něco jiného – k jeho jedné hodnotě se dá najít množina odpovídajících záznamů. Jsou tři druhy vyhledávacích klíčů: hodnotový, hashovaný a relativní (přímo pozice v souboru).
- Logickému záznamu odpovídá fyzický záznam délky R, BÚNO na magnetickém disku; ten může obsahovat další data — oddělovače, ukazatele, hlavičky. Záznamy mají buď pevnou, nebo proměnnou délku.
- Fyzické záznamy jsou organizovány do bloků délky B – hlavní jednotky přenosu mezi RAM a HDD.
- $\frac{B}{R}$ se nazývá blokovací faktor ($\lfloor b \rfloor$).
- Schéma organizace souborů (SOS) je popis logické paměťové struktury, ve které se soubor nachází. Může obsahovat více logických souborů, ten který nese uživatelská data je primární, jeho délka v počtu záznamů se značí N. Další operace nad SOS jsou **BUILD, REORGANIZATION, OPEN, CLOSE**.
- Dotaz je každá funkce, která pro zadaný argument vrátí "odpověď" – množinu záznamů. Dotazy mohou být buď na úplnou, nebo jen částečnou shodu, popř. intervalovou shodu.
- Vyváženost struktury je zajištění omezení délky cesty při vyhledávání nějakým výrazem ($O(\log N)$ atp.), popř. rovnoměrnosti naplnění bloků (popisuje ji faktor naplnění stránek). Splnění se dosahuje štěpením a sléváním bloku.
- SOS, které splňuje vyváženost, se nazývá dynamické, jinak statické.

Fyzická média

Soubory se fyzicky ukládají hlavně na magnetické pásky nebo HDD.

- Pásky umožňují jen sekvenční přístup, bloky jsou při vyhledávání čteny a kontrolovány sekvenčně (vhodné setřídění dat podle klíče). Pro kapacitu pásky je důležitá hustota, jsou nutné meziblokové mezery. Sekvenční čtení trvá $t' = R \cdot t/(R + W)$, kde W jsou meziblokové mezery, t je transfer rate.
- HDD umožňuje přímý přístup k datům. Uvažují se hlavy, válce (cylindry) a stopy. Vždy jen jedna hlava může číst. Většinou se HDD dělí na sektory. Pro rychlost přístupu jsou důležité následující proměnné:
  - seek (přesun na jiný válec) — s
  - rotate (doba 1 půlotáčky) — r
  - block transfer time — propustnost sběrnice — btt

Nové disky mají různý počet sektorů na 1 stopu a tím pádem složitý výpočet cylindru a hlavy z čísla bloku, který dělá řídící elektronika. Udávaná adresa cylindr-hlava-sektor tak nemusí mít nic společného se skutečností.

Př. nechť 1 stopa = 512 K. Načtení 1 MB pak trvá minimálně $s + r$ pro nalezení prvního sektoru a $2 \times 2r$ za načtení stop. Při čtení z náhodně rozmístěných 4 K bloků ale vyjde $256 \times (s + r + btt)$ — tj. až 100x pomalejší.

Časový odhad doby přístupu k záznamu T_F je složitější:

- $s + r + btt$ obecně
- $r + btt$ pro další záznam ve stejném cylindru

Máme i operaci **REWRITE** — přepis (během 1 otáčky disků): $T_{RW} = 2r$. Další časy se značí T_I (**INSERT**), T_D (**DELETE**), T_U (**REWRITE**), T_Y (**REORG**), T_X (načtení celého souboru).

13.2 Statické metody organizace souborů

Sekvenční soubor

Jsou dvě varianty:

- Neuspořádaný (hromada) – jenom fyzicky za sebe naplácané záznamy. Složitost nalezení $O(N)$.
- Uspořádaný – záznamy řazeny podle klíče. Aktualizované záznamy se umisťují do zvláštního neuspořádaného souboru, **REORGANIZATION** celek znova setřídí. Nalezení je opět $O(N)$ nebo $O(N/\lfloor b \rfloor)$. U médií s přímým přístupem ale $O(\log_2 N)$.

Index-sekvenční soubor

Primární soubor je sekvenční soubor setříděný podle primárního klíče a nad ním (i víceúrovňový) index: číslo bloku a minimální hodnota klíče v něm; pro vyšší úroveň to samé, ale na blocích indexu nižší úrovně.

Nejvyšší úroveň indexu je obvykle jen jeden blok (master). Počet úrovní se dá spočítat: $x = \lceil \log_p N/\lfloor b \rfloor \rceil$, kde $p = \lfloor B/(V + P) \rfloor$ a V je velikost klíče a P velikost pointeru na blok nebo záznam.

Zařazení nového záznamu – problémy: přidávání do oblasti přetečení a v ní řetězení záznamu za sebe (každý záznam tam má pointer na další záznam v oblasti přetečení, nejenom blok). Pro oddálení nutnosti vkládat do oblasti přetečení lze bloky plnit na míň než 100 %.

Indexovaný soubor

Máme primární soubor a indexy pro různé vyhledávací klíče. Indexují se přímo záznamy, ne jen bloky. Primární soubor už nemusí být setříděný. Varianta: clusterovaný index – záznamy se společnou hodnotou 1 atributů jsou blízko sebe (jde jen pro jeden atribut).

Index může být podobný jako u index-sekvenčního souboru, ale lepší je, když pro záznamy se stejným klíčem je ve všech úrovních až na první (nejnižší) jen jeden klíč, který pak odkazuje na seznam záznamů. Pro aktualizace se nepředpokládá oblast přetečení, ale změny v indexu. $T_F = (1 + x)(s + r + btt)$.

Lze použít i dotazy na kombinovanou částečnou shodu, ale trvají dlouho. Alternativa: kombinovaný index pro více atributů; to ale vyžaduje analýzu, na co jsou dotazy nejčastější.

Bitové mapy

Bitové mapy jsou efektivní způsob indexace pro atributy s malou doménou (max. stovky). Pro každou hodnotu této domény vyrobíme vektor bitů délky N, kde jednička na i-té pozici indikuje, že i-tý záznam má právě tuto hodnotu atributů.

Booleovské dotazy potom fungují jako bitové operace nad těmito vektory. Vektory bitů lze navíc komprimovat — nejsou tak velké, vhodné pro databáze s hodně záznamy.

Indexy v DIS (document information system)

Jedná se o tzv. invertované soubory pro fulltextové hledání – pro každé slovo máme uložen počet souboru, kde se vyskytuje, a pointer na soubor souřadnic, tj. dvojic [dokument, pozice]. Dotazy na shodu pro více slov pak jsou množinové průniky (začínající od slova s nejmenším počtem výskytu v databázi).

Získání invertovaného souboru: rozparsování na slova, setřídění, odstranění duplicit, výpočet frekvencí slov. Zipfův zákon distribuce slov $f = k/r$, kde k je konstanta, r je pořadí četnosti, f je četnost. Taky lze využít kompresi a nebo zmenšit indexy vyhozením nevýznamných slov.

Soubor s přímým přístupem

Záznamy v primárním souboru ("adresový prostor") jsou rozptýleny pomocí hashovací funkce.

Implementace: buď hash = číslo stránky; nebo hash = číslo stránky i relativní pozice v ní. Pro kolize se snažím umístit záznamy pokud možno do stejné stránky.

13.3 Statické hashovací metody

Jako perfektní hashování se označuje nejen stav, kdy funkce nedává pro danou množinu žádné kolize, ale i takový systém, kde máme zaručený čas $O(1)$ pro přístup k záznamu.

Cormackovo perfektní hashování

Hashování odpovídá "velké funkci a malé tabulce" ze sekce o hashování. Máme soubor a adresář. Krom primární hashovací funkce h jsou potřeba další hashovací funkce $h_i(k,r) = (k \mod (2i + 100r + 1)) \mod r$, kde:

- r je velikost oboru hodnot (počet kolizí)
- i je nějaký vhodný parametr — číslo hash fce (hledá se postupným zkoušením, dokud není výsledek bez kolizí).

V adresáři uchováváme pro každý záznam odkaz do primárního souboru, r a i. V primárním souboru máme klíč a data. Při hledání:

1. Zahashujeme klíč pomocí h, dostaneme se do adresáře, pokud je tam $r = 0$, nenašli jsme.
2. Pokud $r \geq 1$, spočítám podle uloženého i hash-fci h_i, vezmu odkaz do primárního souboru, přičtu výsledek h_i a v primárním souboru zkontroluju, jestli jsem našel co jsem hledal.

Pro **INSERT** spočítám h a:

1. Pokud na daném místě v adresáři nic není, najdu volné místo v primárním souboru délky 1 a vložím
2. Pokud tam už něco je, najdu volné místo délky $r + 1$, zvětším r a najdu i, aby kolizní od sebe rozházela a přeházím je na nové místo.

r nemusím zvětšovat o 1, ale rychleji, takže najdu požadované i lépe (pokud mám blbý h_i, někdy r prostě musím zvětšit o víc).

Toto hashování lze použít i pro dynamický paměťový prostor, když z primárního souboru uděláme nesouvislý prostor, kam lze přidávat stránky.

Perfektní hashování Larson-Kalja

Uchováváme menší pomocnou tabulku než předchozím případě – pro každou stránku adresového prostoru máme jen jednu hodnotu – separátor. Máme:

- Množinu M hashovacích funkcí h_i, které vytvářejí pro každý záznam posloupnost stránek, do kterých ho můžeme zkoušet vkládat
- Navíc druhou sadu jim jednozn. odpovídajících funkcí s_i, které počítají signaturu.

Pokud je separátor stránky větší než signatura záznamu, bude záznam v této stránce, jinak ho tam nemůžu vložit. Po přeplnění vyhodím záznamy s největší signaturou a zmenším separátor. Prvek se nemusí povést vložit (když mi dojdou obě sady funkcí).

Pokud vkládám do plné stránky, nemůžu prvek vložit, i když má prvek menší signaturu než separátor — separátor pak musím zmenšit a ze stránky vyhodit i něco dalšího, čímž dojde ke kaskádování.

Toto hashování je vhodné pro málo vkládání a hodně čtení – mám zaručený jen jeden přístup na disk, protože malý adresář se do paměti vejde.

Dotazy na částečnou shodu

Máme-li záznamy s n atributy, použijeme pro každý z atributů zvláštní hashovací funkci, která generuje d_i-bitový výsledek (signaturu atributu). Signatura (hash) celého záznamu je pak konkatenací signatur atributů. Pro adresový prostor o velikosti $M = 2^d$ stránek volíme délky signatur atributů tak, aby $\sum d_i = d$.

Dotaz na částečnou shodu je potom dotaz, kde bity některých atributů jsou nahrazeny "?". Nazveme s-bitovou signaturu dotazu, má-li zadaných s bitů. Takový dotaz je 2^{d-s}-krát dražší než přímý konkrétní.

Je-li pravděpodobnost dotazů na i-tý atribut rovna P_i, vyjde průměrná cena dotazů:

$$\sum_{q \subseteq \{1,..n\}} (P_q \cdot \Pi_{i \notin q} 2^{d_i})$$

Ideální rozvržení d_i proto vychází (Lagrangovými multiplikátory):

$$d_i = (d - \textstyle\sum_{j=1}^{n} \log_2 P_j)/n + \log_2 P_i.$$

Je nutné upravit případné extrémy a přepočítat.
Pro urychlení dotazů můžeme použít deskriptory stránek:

- Máme deskriptor záznamů: $w = w_1 + .. + w_n$-bitový řetězec, kde pro každý atribut $A_1, ..A_n$ máme právě jednu jedničku (může být zadáno jinak, ex. vzorce pro ideální w_i i počet jedniček k nim).
- Deskriptor stránek je bitový OR deskriptorů všech záznamů ve stránce.
- Při hledání vyrobím deskriptor dotazů: místo "?" dám samé nuly. Pokud má dotaz někde v deskriptoru "1" a stránka "0", nemusím tam hledat.

Lze udělat i dvouúrovňově – s deskriptory segmentů.
Další možnost urychlení jsou Grayovy kódy, které se snaží řešit nesouvislost oblasti stránek se záznamy vyhovujícími dotazům (např. 10???1010). Jde o jiný způsob kódování binárních čísel — dvě po sobě jdoucí čísla se liší vždy jen v jednom bitu. Max. zisk — o 50% lepší než binární. Konverze čísla do Grayova kódování z pozičního vypadá následovně:

$$G(x) = B(x) \text{ xor } B(\lfloor \frac{x}{2} \rfloor)$$

Zpětně (g_i je i-tý bit Grayova kódu, n-tý bit je nejvyšší):

$$b_n = g_n,\ b_i = g_i \text{ xor } b_{i+1}$$

13.4 Dynamické hashování

Rozšiřitelné hashování – Fagin (Koubkovo "externí hashování")

Kromě primárního souboru (nebo souborů, protože jde o dynamickou strukturu) mám pomocnou strukturu – adresář s pointery na stránky o velikosti 2^d (přičemž některé stránky mohou být v adresáři uvedeny víckrát) a používám hashovací funkci s rovnoměrným rozdělením.

Vždy použiju jen prvních d bitů hashe, minimum kolik potřebuju. Podle nich určím záznam v adresáři a z něj najdu stránku. Pro stránku se může používat míň bitů, proto na ní může ukazovat víc záznamů v adresáři. Když se stránka naplní, rozštěpím ji na 2 podle dalšího bitu hash-funkce.

Pokud už stránka používá stejně bitů jako adresář, musím zvětšit o 1 bit i adresář (a zdvojnásobit jeho velikost), s ostatními stránkami nic nedělám. Při vypouštění záznamů lze stránky slévat, pokud si pamatuju jejich zaplněnost (můžu slévat jen stránky, které se liší jen v posledním bitu).

Adresář můžeme ukládat na jedné stránce externí paměti. Potom **FETCH** jsou max. 3 operace a **INSERT** nebo **DELETE** max. 6 operací s externí pamětí. Očekávaný počet použitých stránek je $\frac{n}{b \ln 2}$, kde b je počet prvků v jedné stránce, a velikost adresáře $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$ – to je víc než lineární růst, tj. nelze používat donekonečna (bez důkazů).

Lineární hashování – Litwin

V tomto případě nemám adresář, jen oblast přetečení, přístupnou pointry z primárních stránek. Data vkládám do primárních stránek, po každých L **INSERT**ech se vynuceně štěpí určená stránka (v pořadí podle čísel stránek $0, 1, 00, 01, 10, 11, 000$). Podle hashe (jeho konec musí odpovídá číslu stránky) se rozdělují při štěpení prvky.

FETCH: když mám aktuálně n stránek, vezmu posledních $\lfloor \log_2 n \rfloor + 1$ bitů hashe (pokud je to $> n$, zanedbám horní bit) a tím dostanu číslo stránky. Když v ní prvek není a stránka je přetečená, podívám se ještě do oblasti přetečení.

Při štěpení je nutné vrátit do rozštěpených stránek i záznamy z oblasti přetečení. Problémem jsou více zaplněné stránky na konci, tedy ještě nerozštěpené. Řešením je buď nerovnoměrné rozdělení hash-fce, nebo skupinové štěpení stránek.

Skupinové štěpení stránek

Rozšíření Litwina pro lepší využití stránek:

- Stránky jsou vždy organizovány vždy v s skupinách po g stránkách (začínáme s s_0 skupinami, postupně se s zvětšuje, g zůstává).
- Mám inicializační hash-funkci h do $\{0 \dots (g \cdot s_0) - 1\}$.
- Štěpím také po L vložení, vždy g stránek do $g + 1$, na to mám nezávislé hashovací funkce $h_0, ...h_\infty$ do $\{0 \dots g\}$.
- Až dostanu s skupin po $g + 1$ stránkách, přeorganizuju stránky zpátky do skupin po g (můžu přidat nějaké prázdné, aby to vycházelo).

Při hledání se počítají všechna prehashování úplně od začátku — je to docela HW náročné, ale proti rychlosti disků se vyplatí.

Spirálová paměť

Tohle je další rozšíření Litwinova hashování, tentokrát pro exponenciální rozdělení klíčů. Místo rozdělení jedné stránky na starou a novou tu starou zahodí a přidá dvě nové.

Klíče jsou nejprve rovnoměrně rozděleny hash-funkcí $G(c,k) \to \langle c, c+1\rangle$ $(c \in \mathbb{R}_0^+)$, pak přepočteny do $\langle b^c, b^{c+1}\rangle$ a zaokrouhleny na konkrétní čísla stránek. Při změně c se mění velikost prostoru.

Při expanzi se nové c volí tak, aby zničilo stránku, která se štěpí (nejnižší číslo):

$$c_{n+1} := log_b(f+1) \text{ (kde } f \text{ je číslo 1. stránky).}$$

Aby se mi neposouvaly všechny stránky pořád dál od 0, první zahozené stránce dám nové číslo a znova ji použiju – přepočítávání log. stránek na fyzické se pak dělá rekurzivně:

1. Je-li $\lfloor$logická stránka$/b\rfloor < \lfloor$(1+logická stránka)$/b\rfloor$, pak fyzická stránka := fyzická stránka($\lfloor$logická stránka$/b\rfloor$)
2. Jinak fyzická stránka = $\lfloor$log. stránka$/b\rfloor$.)

Hashovací funkce zachovávající uspořádání

Kvůli urychlení intervalových dotazů se objevily hashovaci metody, zachovávající uspořádání klíčů:

- Lineární hashování pro intervalové dotazy – vychází z Litwina, využívá nejvýznamnější bity k určení stránky.
- Částečně lineární hashování zachovávající uspořádání – vychází ze znalosti rozdělení klíčů, dělí doménu klíčů na nestejně dlouhé intervaly, aby vyvažovalo nerovnoměrnost rozdělení. Nelze ale pořád reorganizovat, takže se upravují jenom přilehlé intervaly při vkládání a odebírání. Nehodí se pro dynamicky rostoucí doménu klíčů. Může být provedeno víceúrovňově.

13.5 Třídění na vnější paměti

Pro třídění něčeho, co se nevejde do operační paměti, se používá **MERGESORT**:

1. Ze dvou souborů vezmu dva prvky
2. Vyberu menší z nich, zapíšu ho na výstup
3. Ze souboru, odkud ten menší pocházel, načtu další.
4. Konec setříděného úseku poznám tak, že další načtený prvek je menší než ten vypsaný. Pokud dojdu na konec setříděného úseku na jednom ze vstupů, dokopíruju zbytek setříděného úseku i z druhého vstupu na výstup.

Postupně dostávám delší setříděné kusy. Původní použití — setříděni kousků, které se vešly do paměti, a slévání na páskách. Dnes je použitelné i na HDD.

N-cestný **MERGESORT** na vnější paměti: Mám-li $n+1$ stránek v paměti:

1. Vytvořit setříděné běhy velikosti n stránek (použít **HEAPSORT** nebo **QUICKSORT**).
2. Pak v každém kroku slévat (maximálně) n nejkratších běhů, výsledek ukládat jako 1 běh.

Pro M stránek v celém souboru je složitost $O(2M\lceil \log_n M/n\rceil)$ — celé projde $\log_n(M/n)$-krát procesem slévání.
HEAPSORT může vytvořit i delší běhy, než co se vejde do paměti:

1. Průběžně odebírám a vypisuju minimální prvky a načítám další.
2. Pokud je načtený prvek větší než minimum, hodím ho na haldu.
3. Pokud je menší, dám ho do aktuálně vznikající haldy na druhém konci pole.

Až se vyčerpá halda, začnu nový běh. V nejhorším případě dopadne stejně jako třídění v paměti, průměrně je 2x lepší, ideálně setřídí všechno.

Kapitola 14

Státnice - Třídění

14.1 Třídění založené na porovnávání

Existuje mnoho algoritmů, známe i (za určitých podmínek) dolní odhad složitosti.

Heapsort

HEAPSORT je třídění pomocí (např.) d-regulárních hald, jen lokální podmínku haldy používáme v duální podobě a máme funkci **DELETEMAX** (která ale funguje stejně jako **DELETEMIN**). Postupně se odebírají maxima a setříděná posloupnost se staví na jejich místě od konce pole. Iterace končí, když v haldě zbývá jen jeden prvek. Celkem $O(n \log n)$ operací.

Mergesort

MERGESORT je snad nejstarší "chytrý" třídící algoritmus. Pracuje s frontou, do které na počátku nahází "předtříděné" rostoucí úseky, potom je v cyklu vybírá a jejich slití vrací zpátky, dokud nemá jen jednu posloupnost. Slití je vybrání vždy menšího na výstup a na konci dokopírování zbytků.

Jedna operace slití vyžaduje $O(n + m)$, jsou-li 2 posloupnosti dlouhé n a m prvků. První rozházení vyžaduje lineární čas, potom každé projití všech prvků slitím vyrábí posloupnosti délky $\geq 2^{i-1}$, tedy počet projití všech prvků je $\lceil \log n \rceil$ a celková složitost $O(n \log n)$. Je adaptivní na předtříděné posloupnosti a při omezeném počtu rostoucích úseků dosahuje lineární složitosti.

Quicksort

QUICKSORT je asi nejpoužívanější třídící algoritmus, v průměru má při rovnoměrném rozložení nejnižší očekávaný čas. Využívá techniky rozděl a panuj.

1. Vybere prvek a (pivot).
2. Vytvoří posloupnosti $a_1 \dots a_{k-1}$ prvků menších než a a $a_{k+1} \dots a_n$ větších než k.
3. Na ty sám sebe zavolá rekurzivně, do výsledku zapíše za sebe obě setříděné poloviny.

Procedura (bez rekurze) vyžaduje čas k nebo $n - k$ v jednom běhu, tj. pro a medián, kdyby $n - k = k$, by měl celý algoritmus složitost $O(n \log n)$. Medián lze nalézt v lineárním čase, ale pak by byly **MERGESORT** i **HEAPSORT** rychlejší, proto se jako a bere např. první prvek posloupnosti.

Potom má procedura očekávaný čas $O(n \log n)$, ale nejhorší případ $O(n^2)$, což je pro neznámé rozdělení dat nevhodné. Náhodná volba by celý běh také mohla dost zpomalit, proto se v praxi bere medián tří až pěti pevně zvolených prvků.

Očekávaný čas Quicksortu

Dva libovolné prvky a_i, a_j jsou porovnávány maximálně jednou, a to když a_i nebo a_j je pivot a předtím ani jeden z nich pivot nebyl. Vezmeme si náhodnou veličinu $X_{i,j}$, která má hodnotu 1, když **QUICKSORT** porovnal během výpočtu b_i a b_j z nějaké výsledné setříděné posloupnosti $(b_i)_{1 \leq i \leq n}$, a 0 jinak. Potom $\mathbf{E}X_{i,j} = p_{i,j}$, kde $p_{i,j}$ je pravděpodobnost porovnání. Potom celk. počet porovnání v celém běhu je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}X_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}$$

Pro výpočet $p_{i,j}$ uvažujeme "strom rekurze", kde každý vrchol odpovídá jednomu rekurzivnímu volání procedury a s tím i nějaké podposloupnosti $a_i, \dots, a_j$ (do té už se sahá jen v jeho podstromě). V jeho levém podstromě jsou

operace na úsecích prvků menších než pivot $a_i, \dots a_{l-1}$ této posloupnosti a v pravém na větších $a_{l+1}, \dots a_j$. Přiřadíme každému vrcholu jeho pivot a očíslujeme vrcholy prohledáváním do šířky. Pak $X_{i,j} = 1$, když b_j nebo b_i je první pivot z množiny $\{b_i, \dots b_j\}$ v tomto očíslování (protože kdyby to byl jiný prvek z této množiny, rozdělím b_i a b_j, aniž bych je porovnal; naopak prvky mimo tuto množinu coby pivoty od sebe b_i a b_j neoddělí). Množina $\{b_i, \dots, b_j\}$ má $j-i+1$ prvků, takže $p_{i,j} = \frac{2}{j-i+1}$. Počet porovnání pak vyjde

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \int_1^n \frac{1}{x} \mathrm{d}x = 2n \ln n$$

Jednoduché třídění

- **SELECTIONSORT** třídí vybíráním nejmenšího prvku a jeho prohozením s levým krajním v nesetříděném úseku.
- **INSERTSORT** vkládá do setříděného úseku další prvek a postupným vyměňováním ho řadí na správné místo.
- **SHELLSORT** je jeho vylepšení – postupně **INSERTSORT**em třídí sekvence složené z každého k-tého prvku pro klesající $k \to 1$ (sekvence klesajících k musí být zvolena "šikovně").
- **BUBBLESORT** – iterativně prochází posloupností a prohazuje inverze
- Jeho varianta **SHAKESORT**, která posloupnosti prochází tam a zpátky.

A-sort

Tento algoritmus je aplikací (a, b) stromů v třídících algoritmech, vhodnou pro částečně předtříděné posloupnosti. Jinak proti klasickým algoritmům nemá žádné výhody. Pro algoritmus je nutné znát list s nejmenším prvekm FIRST, cestu k němu od kořene a pro každý list ukazatele na následující v uspořádání NEXT.

Postup: Odzadu (od "předtříděně největšího") vkládat prvky do stromu modifikovaným A-INSERTem a pak přečíst posloupnost listů (jít po NEXT). A-INSERT pracuje tak, že místo pro vložení prvku hledá od FIRST (jde postupně nahoru po otcích a hledá, kde nejdřív může slézt zas k listům).

Složitost: Pomalejší než běžné třídění na libovolná data (asymptoticky stejné), ale rychlejší na částečně předtříděná. Vezmeme F – počet inverzí v posloupnosti. Celk. potřebuju $O(n)$ pro načtení prvků, $O(n)$ pro všechna štěpení dohromady ze všech běhů A-INSERTu a na každé vložení $O(h)$ pro nalezení místa, kde h je výška, kam se z FIRST dostanu, přeskočím tak $f_i \geq a^{h-2}$ vrcholů (menších než vkládaný) a $h \in O(\log f_i)$. Součet f_i za $\forall i$ dává:

$$\sum_{i=1}^{n} \log f_i = n \log(\prod_{i=1}^{n} f_i)^{\frac{1}{n}} \leq n \log \frac{\sum_{i=1}^{n} f_i}{n} = n \log \frac{F}{n}$$

Protože se nepoužívá DELETE, hodí se na toto $(2,3)$ stromy. Pro míru $F \leq n \log n$ má složitost $O(n \log \log n)$, v urč. případech i rychlejší než Quicksort.

Porovnání algoritmů

Algoritmus	Čas nejhůř	Čas ∅	Porov. nejhůř	Porov. ∅	PP	Paměť	AD
QUICKSORT	$\Theta(n^2)$	$9n \log n$	$n^2/2$	$1.44n \log n$	A	$n + \log n + c$	N
HEAPSORT	$20n \log n$	$\leq 20n \log n$	$2n \log n$	$2n \log n$	A	$n + c$	N
MERGESORT	$12n \log n$	$\leq 12n \log n$	$n \log n$	$n \log n$	N	$2n + c$	A
A-SORT	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	A	$5n + c$	A
SELECTIONSORT	$2n^2$	$2n^2$	$n^2/2$	$n^2/2$	A	$n + c$	N
INSERTSORT	$O(n^2)$	$O(n^2)$	$n^2/2$	$n^2/4$	N	$n + c$	A

c je nějaká konstanta, F značí počet inverzí v posloupnosti, PP – přímý přístup, AD – adaptivní na předtříděné.

Pro krátké posloupnosti je do délky 22 vhodný **SELECTIONSORT**, do 15 **INSERTSORT**, jinak **QUICKSORT**, což vede k hybridnímu algoritmu. Pro **A-SORT** jsou nejvhodnější $(2,3)$-stromy. Poměr časů **QUICKSORT, MERGESORT, HEAPSORT** je v průměru $1 : 1.33 : 2.22$, platilo to ale v roce 1984 :-).

Vylepšení Mergesortu

Nedosahuji optimálních výsledků, pokud slévané posloupnosti ve frontách nejsou přibližně stejně dlouhé. Proto provedu úvahu: mějme algoritmus, který slévá rostoucí posloupnosti a uvažujme jeho "slévací" strom T (kde posloupnost $P(v)$ odp. vrcholu v (délky $l(P(v))$) vznikne slitím posloupností z jeho synů). Součet časů pro MERGESORT je pak $O(\sum\{l(P(v))|v \text{ vnitřní vrchol } T\}) = O(\sum\{d(t)l(P(t))|t \text{ list } T\}$. Dále pracujeme jen s délkami posloupností,

vytvoříme algoritmus **OPTIM**, který při slévání sumu minimalizuje – na začátku dá každé jednoprvkové posl. hodnotu c, která odpovídá hodnotě jejího prvku (?). Pro slévání vybírá posloupnosti (stromy) s nejmenším c a slitému stromu přiřadí $c_1 + c_2$. Nakonec zbyde v množině stromů jen jeden, a ten je optimální. Pro třídění fronty podle c se používá BUCKETSORT. Celkem pracuje v čase $O(\sum_{i=1}^{n} l(P_i))$ na posloupnosti rostoucích úseků $P_1, \ldots, P_n$.

14.2 Přihrádkové třídění

Bucketsort (Counting sort)

Algoritmus **BUCKETSORT** třídí jen přirozená čísla z intervalu $< 0, m >$ a to zavedením $m+1$ množin, do kterých je rozhází a nakonec tyto spojí do výsledku. Třídění je stabilní pro opakující se prvky, inicializace množin a projití při konkatenaci potřebují $O(m)$, rozházení prvků pak $O(n)$, takže celkem $O(n+m)$.

Varianta **RADIXSORT** umí třídit i ve větších intervalech, když používá BUCKETSORT na každou jednotlivou číslici. Protože BUCKETSORT je stabilní, bude to celé fungovat.

Hybrid Sort

Sofistikovanější verze **HYBRIDSORT** třídí reálná čísla z $\langle 0, 1\rangle$ (a obecně tedy jakékoliv klíče). Má dané α (počet přihrádek v poměru k n), rozhazuje do $k = \alpha n$ přihrádek a ty pak třídí haldou.

Nejhorší možný čas je $O(n \log n)$, protože nejhůře se může stát, že všechny prvky nacpu do 1 přihrádky. Očekávaný čas pro nezávislé rovnoměrně rozdělené prvky je $O(n)$ – pravděpodobnost velikostí množin se řídí binomickým rozdělením s parametrem $1/k$, střední hodnota pak je:

$$\mathbf{E}(\sum_{i=1}^{k}(1 + X_i \log X_i)) \leq \mathbf{E}(\sum_{i=1}^{k}(1 + X_i^2)) = k + k\left(\frac{n(n-1)}{k^2} + \frac{n}{k}\right) = O(n)$$

Jak je vidět z odhadu, i kdybychom použili algoritmus s kvadratickou složitostí ve třídění jednotlivých přihrádek, zůstane očekávaná složitost lineární.

Wordsort

WORDSORT je modifikace BUCKETSORTu pro třídění slov. Příprava:

1. Rozhodí slova do množin L_i podle jejich délky.
2. Vytvoří dvojice pozice-písmeno $\{(j, a_i[j])\}$, kterou setřídí podle druhé složky BUCKETSORTem, výsledek setřídí podle první složky (stejně), tj. má seznam setříděných dvojic pozice-písmeno, které se ale mohou opakovat.
3. Dvojice rozstrká do množin S_i pro každou pozici i a odstraní duplicity.

Pak v hlavním cyklu postupuje od největší možné délky a pro každé i pracuju jen s množinou slov délky $\geq i$:

1. Podle i-tého písmena rozhodím všechna aktuálně tříděná slova do množin T_x.
2. Potom podle množiny S_i vyberu všechna neprázdná T_x a sliju je za sebe.
3. Pro další krok přidávám kratší slova vždy na začátek množiny aktuálně tříděných.

Výpočet délek slov, inicializace L_i a zařazení slov do L_i vyžadují čas $O(L)$, kde L je součet délek všech řetězců. Vytvoření seznamu dvojic a jeho třídění vyžaduje také $O(L)$. Založení S_i a přeházení dvojic do nich také $O(L)$. Inicializace T_x je $O(|\Sigma|)$, kde $|\Sigma|$ je velikost abecedy. V hlavním cyklu v každém kroku potřebuji dvakrát čas $O(l_i)$ (součet délek slov dlouhých i nebo víc). Celkem tedy $O(L + |\Sigma|)$.

14.3 Pořadové statistiky (hledání mediánu)

Vstupem je (neuspořádáná) posloupnost n (navzájem různých) čísel. Výstup je $\lfloor \frac{n}{2} \rfloor$-té, nebo obecně k-té nejmenší číslo z nich. Složitost budeme měřit počtem porovnání.

Z dvou následujících **FIND** bývá rychlejší než **SELECT** pro většinu případů, ale nemá zaručenou asymptotickou složitost. Je známo, že medián lze nalézt na $3n$ porovnání a že dolní odhad počtu porovnání je $2n$.

Hledání mediánu technikou rozděl a panuj (algoritmus FIND)

Tento algoritmus používá techniku "rozděl a panuj". chová se podobně jako **QUICKSORT** a hledá obecně k-té nejmenší číslo:

1. Vybrat pivota a rozdělit posloupnost pomocí $n-1$ porovnání na 3 oddíly: čísla menší než pivot (a prvků), pivota samotného a čísla větší než pivot ($n-a-1$ prvků).
2. (a) Pokud je $a+1 < k$, hledám rekurzivně $k-a+1$-tý prvek v $n-a-1$ prvcích
   (b) Pokud je $a+1 = k$, vracím pivot
   (c) Pokud je $a+1 > k$, hledám rekurzivně k-tý prvek v a prvcích

Rekurzivní vzorec $T(n) = T(n-1) + (n-1)$ v nejhorším případě, což dává $\Theta(n^2)$. Očekávaný čas odhadneme na $4n$ a dokážeme indukcí podle n z rekurzivního vzorce $T(n,i) = n + \frac{1}{n}\left(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)\right)$.

Zaručeně lineární hledání mediánu (algoritmus SELECT)

Vylepšení, garantující dobrý výběr pivota a lineární složitost i v nejhorším čase:

1. Rozdělit posloupnost na pětice (poslední může být neúplná, mějme threshold např. $n = 100$, pod kterým množinu přímo třídíme)
2. V každé pětici najít medián
3. Rekurzivně najít medián těchto mediánů
4. Použít ho jako pivot pro dělení celé posloupnosti, protože je větší než alespoň 3 prvky z alespoň 1/2 pětic (až na zakrouhlení), je větší než alespoň $1/2 \cdot 3/5 = 3/10$ prvků (a také menší než alespoň 3/10 prvků)
5. Z toho mám vždy zaručeno, že zahodím alespoň 3/10 prvků pro následující krok.

Vzorec potom vychází: $T(n) = c \cdot \frac{n}{5} + T(\frac{n}{5}) + (n-1) + T(\frac{7}{10}n)$ a podle Master Theoremu nebo substituční metodou se dá vyčíslit jako $T(n) = \Theta(n)$.

Kapitola 15

Formální základy databázové technologie

Kapitola 16

Jednotlivé otázky

16.1 Relační kalkuly, relační algebry, deduktivní databáze

Relační kalkuly

- Relační model — dotazování — relační kalkuly (Skopal)
- neprocedurální jazyky
- relačně úplné
- využití aparátu predikátové logiky 1. řádu pro dotazování
- Doménový relační kalkul (DRK) — pracuje s daty na úrovni atributů
- N-ticový relační kalkul (NRK) — pracuje s daty na úrovni n-tic (řádků databáze)

Relační algebry

- neprocedurální jazyk
- obsahuje šest základních operátorů:
  - výběr (select): — vrací relaci jejíž hodnoty atributů splňují danou podmínku, která může obsahovat jména atributů, konstanty, operátory prorovnávání a logické spojky
  - projekce (project) — vrací relaci obsahující vybrané sloupce neobsahující shodné řádky
  - sjednocení (union) — vstupní relace musí mít shodnou aritu (stejný počet atributů) a domény atributů musí být stejného typu, výsledná ralace neobsahuje opakující se řádky
  - množinový rozdíl (set difference) — musí platit stejné předpoklady jako u sjednocení
  - kartézský součin (Cartesian product) — předpokládá se, že atributy vstupních relací jsou disjunktní (v opačném případě musí být provedeno přejmenování), v praxi mnohdy neproveditelné kvůli vysoké režii
  - přejmenování (rename)
- tyto operátory vytváří z jedné, dvou či více vstupních relací výsledek ve formě nové relace
- Pokorného slajdy
- Relační model — dotazování — relační algebra (Skopal)

Deduktivní databáze

- DJ II

viz wcs:Deduktivní databáze

16.2 Bezpečné výrazy, ekvivalence dotazovacích jazyků

16.3 Relační úplnost

Dotazovací jazyk, kterým lze vyjádřit všechny konstrukce relační algebry (tj. všechny dotazy, které lze popsat relační algebrou) se nazývá relačně úplný.

NRK i DRK jsou relačně úplné

16.4 Věta o tranzitivním uzávěru relace

- Věta:

For an arbitrary binary relation R, there is no expression E(R) in relational algebra equivalent to R+, the transitive closure of R.

Důkaz je na dvě strany v článku níže. Provádí se částečně sporem, částečně indukcí přes počet operací v hypotetickém výrazu (pro výpočet tranzitivního uzávěru) relační algebry.

- Zdroj (znění a důkaz věty):

Článek: Alfred V. Aho and Jeffrey D. Ullman: Universality of data retrieval languages. In POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, 1979, strany 110–119 link1 link2

V češtině u Prof. Pokorného na slajdech.

16.5 Datalog, sémantika Datalogu pomocí nejmenšího pevného bodu

16.6 Datalog s negací, stratifikace, předpoklad uzavřeného světa

16.7 Sémantika SQL

16.8 Logické problémy konstrukce informačního systému

Jaroslav Pokorný: "Jsou to problemy z DJ letni semestr — deduktivny databaze, korektnost a uplnost IS, obecne zavislosti a jejich zpracovani. Je to ve slajdech za DATALOGem." link-slajdy

Kapitola 17

Databázové modely a jazyky

17.1 Typy dotazovacích jazyků (procedurální, neprocedurální, jazyky pro výběr dokumentů)

Procedurální

(nebo také navigační, algebraické)

- dotaz jako posloupnost operací nad relacemi, jsou založeny na relační algebře.

Neprocedurální

(specifikační, deskriptivní, deklarativní)

- dotaz se zadává jako predikát charakterizující výslednou relaci, výsledkem výběru dat je relace, která splňuje podmínky formule. Jsou založeny na relačním kalkulu.

Jazyky pro výběr dokumentů

- boolský model
- vektorový model

17.2 SQL

- neprocedurální jazyk s mnoha procedurálními rozšířeními.

17.3 Vyhodnocování a optimalizace dotazů

17.4 Algoritmy vyhodnocení dotazů v Datalogu a Datalogu s negací

Slajdy Dotazovací jazyky 2

nerekurzivní program

Zavislostní graf, (U,V) je hrana pokud exituje pravidlog V:- ...U... Jedna se o nerekurzivní program, tedy graf je acyklický. Z toho plyne existence topologického uspořádání. Podle tohoto uspořádání zpracovávám virtuální relace.

- pravou stranu převeď na spojení a selekci
- proveď na výsledek projekci
- předchozí dvě pravidla proveď pro všechna pravidla se stejnou hlavou a výsledek sjednoť

rekurzivní program

naivní algoritumus, polonaivní

datalog s negací

stratifikace

17.5 Implementace relačních operací

Fyzická implementace relačních databází (Databázové systémy)

17.6 Indexace dokumentů

17.7 Modely a vlastnosti transakcí

viz wen:ACID

17.8 Izolace transakcí, alokace prostředků (zámky, granularita zamykání, dvoufázové uzamykání, deadlock)

viz wen:Isolation (database systems), wen:Lock (database), wen:Two phase locking, wen:Deadlock

- 2PL (dvoufázové zamykání)
  - pokud už transakce o nějaký zámek požádala a uvolnila ho, nesmí o žádný požádat znovu = 2 fáze — zamykání a odemykání
  - takovýto rozvrh je konfliktově uspořádatelný, ale negarantuje zotavitelnost rozvrhu (=> striktní 2PL)
- striktní 2PL
  - všechny zámky jsou uvolněny až při ukončení transakce
  - garantuje navíc zotavitelnost rozvrhu
  - zabezpečení proti kaskádovému rušení transakcí
- deadlock (uváznutí)
  - transakce čekají navzájem na nějaký zdroj a nechtějí si dát ten svuj
  - waits-for graf — zobrazují se tam závislosti, pokud je cyklus, je uváznutí
  - snížení frekvence uváznutí (ne její řešení)
    - lock downgrade — pokud už nepotřebuji výhradní zámek, snížím na sdílený
    - lock upgrade — pokud potřebuju výhradní a mám sdílený, mám větší právo, než ten co nemá nic a chce výhradní
  - řešení: časová razítka, konzervativní 2PL
- časová razítka (pro 2PL)
  - každá transakce dostane nějakou prioritu (např. časové razítko — čím starší, tím vyšší priorita)
  - dvě možnosti řešení za použití priorit
    - wait-die: pokud chce transakce T1 zámek a už ho má někdo jiný (T2), pokud má T1 vyšší prioritu, čeká, jinak zemře
    - wounds-wait: pokud chce transakce T1 zámek a už ho má někdo jiný (T2), pokud má T1 vyšší prioritu, T2 zemře, jinak T1 čeká
  - rušeným transakcím je třeba zvyšovat prioritu aby nebyly pořád utiskované
- konzervativní 2PL
  - všechny zámky které bude potřebovat si vyžádá hned na začátku a nebo alespoň požádá o jejich rezervaci
  - není používán, protože se nemusí vědět, které zámky bude potřebovat a navíc je zde vysoká režie uzamykání
- Optimistické řešení
  - používá se tam, kde dochází velmi zřídka k jakýmkoliv konfliktům, naopak tam kde je více konfliktů je použít nelze
  - skládá se ze tří fází
    - Read — transakce čte z databáze, provádí různé operace, ale vše zapisuje do svého prostoru (databázi nemění)

    * Validation — transakce předloží co vytvořila a SŘBD zkontroluje, zda to není v konfliktu s jinou transakcí, pokud ano, transakce je zrušena, v opačném případě potvrzena
    * Write — vše je zapsáno do db
- Časová razítka
  - každá transakce dostane časové razítko (TS(T1), TS(T2)) a následně během rozvrhování je kontrolováno pořadí konfliktních operací. Pokud TS(T1) < TS(T2) a A1 (akce T1) je v konfliktu s A2 (akce T2) musí A1 nastat před A2, jinak je T1 restartována.
  - pro dosažení zotavení je potřeba implementovat bufferování všech zápisů dokud transakce nepotvrdí.

17.9 Zotavení, žurnály

Transakce — uzamykací protokoly, zotavení (Databázové systémy)
Zotavení po havárii systému provádí recovery manager, zajišťuje:

- atomicitu (odvolání akcí, pokud byla transakce zrušena — undo)
- trvanlivost (zapsání potvrzených akcí i když systém havaroval)

Zotavení je jedna z nejsložitějších součástí SŘBD, používá se algoritmus ARIES — spustí se po restartu havarovaného systému.

- tři fáze ARIES
  - Analysis — identifikují se dirty pages — stránky modifikované, ale nezapsané v okamžiku havárie
  - Redo — zopakují se všechny akce od příslušného místa (zapsané v logu) tak aby se databáze dostala do stavu před havárií
  - Unod — odvolají se všechny akce těch transakcí, které nebyly potvrzeny (commitnuty), tedy databáze obsahuje pouze potvrzené transakce

Log algoritmu ARIES se označuje journal, každá změna databáze je nejdříve uložena zde (právě kvůli možné havárii). Po restartu algoritmus vystopuje všechny akce před havárií a znovu je provede tak, aby se db dostalo do stavu před havárií. Nepotvrzené akce jsou zrušeny. Při rušení je opět vše logováno pro případ opětovné havárie při provádění ABORT.

17.10 Databáze textů: modely (boolský, vektorový)

viz to samé v jiném okruhu

17.11 Vyhledávání v textech

KMP algoritmus

- testování jednoho vzorku oproti textu
- má pomocné pole délky vzorku
- o(m + n) (délka textu + vzorku)

Boyer-Moorův algoritmus

- testování jednoho vzorku oproti textu
- testuje odzadu dopředu ve vzorku
- má dané dvourozměrné pole kam skočit, když text a vzorek různé (pro znaky, které vzorek neobsahuje skáče vždy o celý vzorek)
- průměrná složitost je o(m*n) ale v praxi o(m/n)
- na běžném textu je mnohem efektivnější než KMP
- lze vylepšit tak, že dvourozměrné pole je nahrazeno dvěma jednorozměrnými
  - P1 — pro každé x z abecedy poslední pozice výskytu ve vzorku (pro znaky co nejsou ve vzoru délka celého vzorku)
  - P2 — pro každou zkontrolovanou část (nějaká podčást odzadu vzorku) je dán skok na stejný podřetězec vyskytující se dříve ve vzorku.
  - je vybrán větší skok z obou polí

Aho-Corasick

17.12 Rodina jazyků a nástrojů XML (XML schema, XPath, XQuery, XSLT)

`http://www.w3schools.com`

Kapitola 18

Implementace databázových systémů

18.1 Seznam otázek

Metody indexace relací, hashování, B-stromy, datové struktury na externí paměti. Vícerozměrné dotazy implementované pomocí hashovacích metod, vícerozměrné mřížky, vícerozměrných stromů. Přístupové metody k prostorovým objektům: R-stromy a jejich varianty. Databáze textů: modely (boolský, vektorový), vyhledávání v textech, signatury, metody implementace signatur (vrstvené kódování), uspořádání odpovědi. Komprese dat: predikce a modelování, reprezentace celých čísel, obecné metody komprese, komprese bitových map, řídkých matic, trie, textů. Huffmanovo kódování (statické, dynamické), aritmetické kódování, LZ algoritmy. Uzamykací protokoly, časová razítka. Distribuované transakce.

Vsechnu latku pokryva text "Zaklady implementace souboru a databazi" od Pokorneho a Zemlicky a slajdy "Dokumentografickych informacnich systemu" od Kopeckeho

18.2 Metody indexace relací

- relacie mozu byt chapane ako subory, zaznamy suboru ako n-tice relacie, samozrejme su tam rozdiely, napr.:
  - schemy vsetkych relacii sa natiahnu naraz s celou DB, subory musime otvarat/zatvarat jednotlivo
- indexsekvenční soubor (indexováno jen podle primárního klíče)
- invertovaný soubor — lze indexovat podle čehokoliv — indexují se přímo záznamy
- bitová mapa
  - pro atributy malých domén (pohlaví, typ karoserie vozu...) se pro každou doménu vytvoří bitový vektor a jednička tam kde hodnota pro daný řádek platí, tedy vždy jen jedna jednička na řádku => hodně nul, lze dobře komprimovat
  - vyhledávání je rychlé používají se operace AND, OR atd.
  - insertem se všechny vektroy prodlouží, přidáním další možnosti (rozšíření domény) se přidá další bitový vektor
  - lze využívat i pro GROUP BY, COUNT atd.

18.3 Hashování

- perfektni hasovani Cormacka
- perfektni hasovani Larsona a Kalji
- Deskriptory stranek, Grayovy kody (viz Vícerozměrné dotazy implementované pomocí hashovacích metod)
- hasovani Fagina
- dynamicke hashovani Litwina
- skupinove stepeni stranek

18.4 B-stromy

- redundantni, neredundantni B-strom
- redundantni, neredundantni B*-strom
- redundantni B+-stromy
- prefixove stromy
- (a, b)-stromy
- stromy s promenlivou delkou zaznamu
- vicerozmerne B-stromy

18.5 Datové struktury na externí paměti

- hromada — nehomogenni záznamy (ruzne delky) jsou ukládány za sebou (bez dalších podmínek) — vyhledání čehokoliv = projití celé hromady (v nejhorším případě)
- sekvenční soubor — podobný jako hromada, jen záznamy jsou pevné délky (homogenni)
  - uspořádaný sekvenční soubor — záznamy uspořádány podle klíče, nově přidávané se ukládají do "souboru aktualizací" a občas je provedena reorganizace, kdy je "soubor aktualizací" vyprázdněn a data z něj zatříděna do uspořádaného sekvenčního souboru. Nalézt záznam lze pak půlením intervalů v case O(log_2(n)).
  - indexsekvenční soubor — obdobně jako předchozí jsou záznamy setříděny podle klíče, navíc existují indexy do jednotlivých bloků (stránek). Indexy jsou tvořeny klíči některých záznamů. Celý index je pak jakýsi B-strom. Při přidávání není index reorganizován, ale je použita oblast přetečení, kam je nový záznam vložen. K záznamu, za který měl být nový záznam vložen se pak uloží číslo bloku i číslo záznamu, kde nový záznam leží. V oblasti přetečení jsou tak vytvářeny lineární seznamy.
  - invertovaný (indexovaný) soubor — indexují se přímo záznamy, ne bloky (jak tomu je v indexsekvenčním souboru), data nejsou v souboru nijak uspořádána, jednotlivé bloky nemusí být na disku za sebou. Indexů může být více různých. Při přidávání není použita další struktura, přímo se upravují indexy. Používá se v DIS (invertovany soubor).
- ukládání pomocí hashování
- b-stromy

18.6 Vícerozměrné dotazy implementované pomocí hashovacích metod

- viacrozmerne dotazy — obsahuju viac atributov (n>1)
  - na uplnu zhodu — hodnoty vsetkych n atributov su pevne dane (n=2: MENO='Jiri' AND MESTO='Praha')
  - na ciastocnu zhodu — hodnoty niektorych atributov (<n) su pevne dane
  - na uplnu intervalovu zhodu — pre vsetkych n atributov su dane intervaly hodnot (napr. VEK>22)
  - na ciastocnu intervalovu zhodu — intervaly hodnot su dane len pre niektore atributy
- implementacia
  - **signatury** — d bitove binarne retazce
    - signatura zaznamu (dlzky d) je zretazenim signatur jednotlivych atributov, pre kazdy atribut A_i je samostatna hasovacia funkcia h_i, ktora vytvori signaturu dlzky d_i. Plati, ze d_0+d_1+...+d_n=d; n-pocet atributov. Signatura dotazu sa tvori podobne, nezadane atributy su reprezentovane retazcom nul.
    - signatura dotazu Q sa porovnava so signaturami zaznamov S — na pozicii, kde je v Q jednicka, musi byt jednicka aj v S, inak zaznam do odpovede nepatri
  - **deskriptory stranok**
    - vrstvia sa deskriptory zaznamov v danej stranke pomocou logickeho OR. Deskriptor zaznamu je opat zretazenie deskriptorov atributov A_i, pre kazdy atribut je dana hasovacia funkcia h_i, ktora priradi kazdej hodnote atributu binarny retazec obsahujuci prave 1 jednicku.

    * subor deskriptorov stranok je pripojeny k primarnemu suboru, ak je velky, moze byt dvojurovnovy (prim. subor sa rozdeli do segmentov, kazdy ma svoj subor deskriptorov, tie tvoria 1. uroven; 2. uroven ma deskriptory ukazujuce do 1.)
    * rychlejsie ako signatury, lebo sa musi porovnat menej deskriptorov (ale na ukor pamate); lepsie ako indexovany subor, lebo nerastie cas vyhodnocovania s rastucim poctom atributov v dotaze, pamatove naroky su podobne
  - **grayove kody** — binarne retazce, hodnoty nasledujuce za sebou sa lisia vzdy len v jednom bite
    * adresa stranky dana signaturou je interpretovana ako grayov kod
    * zaznamy vyhovujuce signature dotazu tvoria zhluky, kde zaznamy su fyzicky za sebou v jednej stranke (alebo viacerych po sebe iducich strankach), nikdy nie je viac pristupov na disku ako u binarneho kodovania
    * dobre pre dotazy na ciastocnu zhodu pri malom pocte zadanych atributov a pre dotazy na intervalovu zhodu

18.7 Vícerozměrná mřížka

Slajdy Pokorný

18.8 Vícerozměrné stromy

Vícerozměrné B-stromy

slajdy Pokorný

- lze pomocí nich indexovat soubor podle několika atributů současně.
- celý strom má tolik hladin, kolik je atributů (hlavní strom není B-Strom!)
- pro každý atribut jsou vytvořeny stromy — pro první jeden, pro druhý tolik, kolik má první atribut různých hodnot atd. U každé hodnoty atributu (uzlu stromu) existuje index na strom s dalším atributem.
- jednotlivé vrstvy celého stromu jsou indexovány polem Level (vrstva = jeden atribut) — slouží pro přeskočení prvních i atributů, které nebyly v dotazu zadány
- stromy v jedné vrstvě jsou propojeny do spojáku pomocí NEXT v kořeni, poddoména jednoho stromu je určena pomocí LEFT a RIGHT (uloženy v kořeni), které ukazují do další hladiny na první a poslední strom, který se týká daného stromu. Tedy každý prvek z vrstvy i-1 má ve vrstvě i určeny všechny prvky ve vrstvě i+1, LEFT a RIGHT dává vše.
- podobná struktura by pravděpodobně šla definovat i pro obecný, nebo jakýkoliv jiný vyhledávací strom.

18.9 Přístupové metody k prostorovým objektům: R-stromy a jejich varianty

viz wen:R-Tree

18.10 Databáze textů: modely (boolský, vektorový)

Boolský model DIS

Každý dokument je popsán množinou termů, které obsahuje. Termy jsou uloženy v indexovém souboru a odtud potom probíhá vyhledávání.

- určení termů:
  - ručně — každý může určit jiné termy, dokonce i jeden člověk se může 2x rozhodnout jinak
  - automaticky — jednoznačné, ale bez porozumnění textu
  - při určování termů by se měla vynechávat nevýznamová slova (předložky, spojky...), ale také příliš specifická slova
- Indexy lze nad kolekcí dokumentů vytvářet dvojím způsobem. Buď je množina termů pevně daná a pro každý nový dokument se aktualizují jen indexy (řízená indexace), nebo se s každým novým dokumentem přidávají i nové termy (neřízená indexace).

- Vyhledávání:
  - probíhá klasicky přes termy pomocí “AND”, “OR”, “NOT”
  - lze také využít faktografické informace o dokumentu (autor, datum vytvoření...)
  - lze použít zástupné znaly ?, *
- Proximitní omezení
  - při vyhledávání lze také užít proximitních omezení, tedy informaci o tom, v jaké vztahu dva hledané termy jsou (o kolik slov jsou vzdáleny, zda jsou ve stejném odstavci, nebo větě.
  - pro práci s omezeními je třeba buď mít přítomny dokomenty a informaci zjišťovat přímo v nich a nebo rozšířit index -> namísto <term_id, doc_id> musí existovat index <term_id, doc_id, par_nr, sent_nr, word_nr>

Nevýhody boolského DIS:

- při zadávání dotazu jsou všechny termy stejně důležité
- ve výsledku disjunktního dotazu jsou promíchány dokumenty které obsahují všechny požadované termy s těmi, které obsahují pouze jeden
- ve výsledku konjunktivního dotazu nejsou dokumenty, které obsahují neobsahují žádný z termů stejně tak jako ty které neobsahují pouze jeden

Vektorový model DIS

Vektorový model je mladší než boolský (asi o 20 let) a vznikl aby odstranil chyby boolského modelu — především díky možnosti ohodnotit termy jak v dotazu, tak v indexu.

- Každý dokument je v indexu definován váhami pro jednotlivé termy.
- dotaz se zadává vektorem vah pro požadované termy <w1, w2, .. , wn>, w in <0,1>
- protože dlouhé dokumenty by byly zvýhodněny (obsahují jistě více termů), provádí se normalizace vektorů, to lze provádět:
  - během indexace, což nezvětšuje odezvu, je však zapotřebí občas vše znovu “přenormalizovat”
  - během vyhledávání — zpomaluje odezvu, je v definici podobnostní funkce
- Oproti boolskému modelu lze omezit velikost výstupu — výstup je seřazen dle relevance a tedy lze omezit jak počtem výstupů tak minimální relevancí.
- Podobnost mezi dotazem a dokumentem se určuje pomocí podobnostní funkce. Taková funkce není jednoznačně daná, podobnost může být tedy při různých implementacích různá. Základní podobnostní funkce je skalární součin přes dotaz a dokument (sim(q,d) = q x d), jsou ale i složitější:

Kosinová míra: $sim(q,d) = \frac{q \times d}{\sqrt{(\sum(q_i^2))} * \sqrt{(\sum(d_i^2))}}$
Jaccardova míra: $sim(q,d) = \frac{q \times d}{\sum(q_i) + \sum(d_i) - \sum(q_i * d_i)}$ a další.

- Pokud je po vektorovém modelu požadován i dotaz s negací, lze váhu w uvažovat z <-1, 1> a stejně tak i zadávat dotaz.
- Indexace:
  - frekvence termu v dokumentu TF = #výskytů termu / #všech termů v dokumentu
  - je potřeba vyřadit nevýznamová slova (stop list = {the, a, an, on ... })
  - ignorovat termy s velmi nízkým výskytem
  - normalizovat frekvenci ostatních termů $NTF_i = \frac{1}{2} + \frac{1}{2} * \frac{TF_i}{max(TF_j)}$, j = index jednotlivých termů
- Inverzní frekvence termu v dokumentech
  - Nasel jsem dvě definice. První je od Kopeckého druhá od Pokorného. Definice jsou v podstatě matematicky ekvivalentní.
  - n je počet všech dokumentů.
  - k je počet dokumentů ve kterých se term vyskytuje.

1. $ITF = -log\left(\frac{k}{n}\right)$
2. $IDF = log\left(\frac{n}{k}\right) + 1$

Váha termu i v j-tém dokumentu se urči jako $w_{ij} = NTF_{ij} * ITF_j$

- Dotazování (dotaz má stejnou reprezentaci jako index):
  - Zadáním vektoru dotazu
  - odkazem na zaindexovaný dokument (tedy "najdi mi dokumenty podobné tomuto")
  - odkazem na jiný dokument (není problém pro něj spočítat vektor termů)
  - přímo vložený text (obdoba předchozího)
- Lze uplatnit i zpětnou vazbu, kdy uživatel označuje relevantní dotazy a DIS na to reaguje přehodnocením výpočtu.

Vyhledávání v textech

Signatury, metody implementace signatur (vrstvené kódování)

Slajdy pokorny

- signatura bloku textu je d-bitový řetězec, který v sobě nese informaci o tom, které termy v bloku jsou
- **vrstvění signatur**
  - každý term má svou signaturu (vypočtenou pomocí hash funkce) dlouhou d bitů a signatura bloku = OR přes všechny signatury termů v bloku.
- **řetězení signatur**
  - signatury se spojují do řetězce — vhodné pro strukturovaná data (autor, rok vydání, nakladatelství...)
- **transponovaný soubor signatur**
  - namísto N řetězců délky d mám d řetězců délky N (pokud signatury bloků naskládám do matice, koukám se pak na ně místo jako na řádky jako na sloupce)
  - protože počet 1 v dotazu délky d je o mnoho menší než d a můžu porovnávat jen řetězce kde má dotaz jedničku, porovnávám toho daleko méně
  - složitější aktualizace
- **soubor indexovaných deskriptorů**
  - stejně jako z termů vytvářím vrstvením signaturu bloku, můžu pro více bloků vytvořit společnou signaturu
- **dvouúrovňové vrstvené kódování**
  - každý blok má dvě signatury — vnětší (délky ds) a vnitřní (délky dn) kde dn << ds, bloky jsou rozděleny do skupin a každá skupina má vrstvenou signaturu ds — podle ní se rozlišují skupiny a ve skupin se bloky rozlišují podle signatury dn (tedy najdu skupinu bloků a pak v ní blok)
- **rozdělený soubor signatur**
  - signaturu bloku délky d rozdělím na dva kusy (prefix a zbytek) a pak seskupím "zbytky" se stejnými prefixy — pokud testuju, otestuju nejdřív prefix a pokud se neshoduje, do skupiny vůbec nemusím chodit, čímž ušetřím celkem dost času

Metody je možné různě kombinovat tak, aby výsledná struktura byla co nejefektivnější.

Uspořádání odpovědi

18.11 Komprese dat: predikce a modelování

Reprezentace celých čísel

Slajdy k OZD II (od slajdu 21) (Fibonacciho, Eliasovy kódy, fázování)

Obecné metody komprese

- ztrátová (nazývá se komprese)
- bezeztrátová (nazývá se kompakce)
- statická
- semiadaptivní
- dynamická

Dvě fáze:

- modelování — přiřazení pravděpodobností jednotkám textu (znaky nebo m-tice znaků pevné délky)
- kódování — transformace do nového kódu

Komprese bitových map

Slajdy k OZD II (od slajdu 92)

- pomocí Huffmanova kódování
  - kódováním posloupností pevné délky
  - kódování běhů (posloupnost nul ukončených jedničkou apod.)

Komprese řídkých matic

Něco lze nalézt u Koubka v kapitole o reprezentaci Trie pomocí matic a jejich kompresi — viz materiály k přednášce Datové struktury II

Trie

Komprese textů

Huffmanovo kódování (statické, dynamické)

Slajdy k OZD II (od slajdu 40) (statické, adaptivní kódování)
Dobře popsáno také na stránkách k DIS

Aritmetické kódování

viz wen:Arithmetic coding

Slajdy k OZD II (od slajdu 47)

Dokumenty k DIS (myslím, že v posledním kroku příkladu je chyba)

LZ algoritmy

Slajdy k OZD II (od slajdu 56) (LZ77, LZSS, LZ78, LZW a další)

18.12 Uzamykací protokoly

viz wen:Concurrency control#Concurrency control mechanism

18.13 Časová razítka

viz wen:Timestamp-based concurrency control#Informal

18.14 Distribuované transakce

viz wen:Distributed transaction

viz wen:Two-phase commit protocol

Ve stručnosti:

- říká se jim také globální transakce
- musejí splňovat ACID
- jsou to transakce, které pracují s více uzly najednou (s více databázemi v síti apod.)
- celkem jednoduchý algoritmus, který problém řeší je "Dvoufázový commit"

Dvoufázový commit

1.fáze (commit-request phase)

- kordinátor pošle všem uzlům dotaz, který je zde potřeba vykonat
- uzly vykonají vše až do doby, kdy by měly commitovat/abortovat
- každý uzel pošle svůj výsledek koordinátorovi (YES/NO)

2.fáze (commit phase)

pokud koordinátor dostal od všech uzlů odpověď YES (success)

- koordinátor pošle commit zprávu všem uzlům
- všechny uzly uvolní své zdroje a zámky
- všechny uzly pošlou koordinátorovi výsledek dotazu
- koordinátor vše skompletuje

pokud koordinátor dostal alespoň jedno NO (failure)

- koordinátor všem uzlům pošle abort zprávu
- každý z uzlů udělá vlastní abort a uvolní zámky a zdroje
- všechny uzly pošlou koordinátorovi výsledek dotazu (??? nechápu ale k čemu to je ???)
- poté co koordinátor obdrží výsledky abortuje celou transakci

Stromový 2-fázový commit (Tree two-phase commit protocol)

Obdoba předchozího jen s rozdílem, že koordinátor je kořenem stromu a vše se děje na stromové struktuře — commity jsou sbírány do uzlu stromu a až jsou všechny, jsou propagovány víš až ke kořeni. Oproti tomu abort je propagován hned.

Problémem 2PC protokolu je, že každý z uzlů čeká, až dodělá práci poslední z uzlů, čímž dost dlouho blokuje.

Kapitola 19

Programovací jazyky a překladače

19.1 Otázky

Následující seznam otázek shnuje mou představu o tom, co je z okruhu Programovací jazyky a překladače ke státnicím potřeba umět. Tato představa nebyla konzultována s vyučujícím(i).

Struktura kompilátoru a navazujících nástrojů (linkery, loadery, debuggery, knihovny, preprocesory)

- schéma překladače
- popis linkeru
- popis loaderu (Unix, Windows)
- statické knihovny, dynamické knihovny, implmentace
- popis C preprocesoru

Konečné automaty a lexikální analýza

- úkol lexikální analýzy
- výstup lexikální analýzy, varianty, problémy
- typické způsoby implementace
- lex, flex

Syntaktická analýza — LL, LR techniky

- úkol syntaktické analýzy
- výstup syntaktické analýzy (vč. možností při analýze shora dolů a zdola nahoru)
- pojmy: gramatika, bezkontextová gramatika, derivace, levá/pravá derivace
- definice: FIRST a FOLLOW
- analýza shora dolů, vztah s LL, definice a popis LL, pojmy rozepsání a krácení
- rozdíly mezi slabou a silnou LL
- implementace analýzy shora dolů rekurzivním sestupem
- analýza zdola nahoru, vztah s LR, definice a popis LR, pojmy shift a reduction
- rozdíly mezi LR, SLR a LALR
- konstrukce LR, SLR a LALR automatu
- yacc, bison
- gramatiky s regulární pravou stranou

Syntaxí řízený překlad a atributové gramatiky

- derivační strom vs. AST
- definice: AG, Attr, Syn, Inh, normální forma, atributový výskyt, In, Out
- definice: acyklické AG, k-průchodové AG, jednoduše k-průchodové AG, zleva-doprava k-průchodové AG, jednoduše zleva-doprava k-průchodové AG
- algoritmus konstrukce průchodů jednoduše zleva-doprava k-průchodovou AG

Reprezentace programu mezikódem

- důvody reprezentace mezikódem
- pojem základní blok
- formáty mezikódu: sekvenční (2-adresový, 3-adresový (trojice, čtveřice), prefixová/postfixová notace), graf základních bloků, strom, DAG
- operandy mezikódu
- SSA
- funkce u trojic a SSA
- generování mezikódu (použití AG, zisk z proházení operací, problém LHS a RHS u přiřazení a jeho řešení, synchronizace u DAGů)

Překlad výrazů a programových struktur

- preklad programove struktur — control-flow, tedy if/when/for/goto pripadne try/catch
- vyrazy — zakladni princip vyhodnocovani, CSE, eventualne eliminace duplicitnich read/write operaci.

Rozsahy platnosti proměnných, aktivační záznamy, implementace vnořených procedur, volací konvence

- obsah aktivačního záznamu
- způsoby implementace vnořených procedur (odkaz na všechny záznamy, odkaz na předchozí záznam, displej) a jejich vlastnosti (režie volání, přístupu k proměnným a prostorová náročnost)
- implementace procedurálních parametrů v Pascalu
- volací konvence: co vše zahrnují (registry vs. zásobník, zachovávání registrů, pořadí předávání parametrů, návratová hodnota, úklid zásobníku)
- popis cdecl, stdcall, fastcall

Vliv architektury počítače na generování kódu a optimalizaci

Zdroje

- Wikipedia: Microarchitecture

Obsah otázky

- registry (počet, typy, způsob přístupu, (ne)ortogonalita)
- instrukční sada (RISC, CISC, (ne)ortogonalita)
- pipelining
- superskalarita
- out-of-order execution
- spekulativní vykonávání instrukcí
- SIMD
- práce s pamětí (heap, stack), zarovnávání

Metody generování kódu, přidělování registrů, optimalizace

Zdroje

- Wikipedia: Category: Compiler optimizations

Obsah otázky

- klasifikace základních bloků podle rozsahu platnosti, výpočet liveBegin a liveEnd, určení kolizí
- výběr a uspořádání instrukcí (1:1, 1:n, m:1, m:n), pojem generator-generator, řazení instrukcí
- objekty umisťované do registrů
- algoritmus obarvování grafu
- optimalizace: důvody, čas vs. prostor, úroveň, požadované vlastnosti
- druhy optimalizací: lokální, globální, v rámci programu
- lokální optimalizace: CSE, copy propagation, dead-code elimination, constant folding, algebraické operace
- globální optimalizace: CSE, copy propagation, dead-code elimination, optimalizace cyklů (přesun invariantního kódu, redukce síly operace, odstranění indukční proměnné)
- paralelizace a vektorizace
- profilem řízená optimalizace

Podpora kompilátorů pro synchronizační primitiva, vlákna

Zdroje

- Using wait(), notify() and notifyAll()

Obsah otázky

- Java: kritické sekce, monitory (synchronized), wait, notify
- Java: vytvoření vlákna, paměťový model, TLS, podpora v knihovnách (thread (un)safe)
- volatile proměnné
- podpora knihovnami při absenci podpory kompilátoru

Objektově orientované jazyky a principy jejich implementace

Zdroje

- Memory Layout for Multiple and Virtual Inheritance
- Compilation of object oriented languages

Obsah otázky

- principy OO: zapouzdření, dědičnost a polymorfismus
- class-based languages vs. prototype-based languages
- OO podle Smalltalk: zasílání zpráv
- implementace dědičnosti: tabulka virtuálních funkcí, volání virtuálních funkcí, princip pozdní vazby, layout objektů při jednonásobné a vícenásobné dědičnosti, diamond problem, virtuální dědičnost, RTTI a jeho implementace (typeid, dynamic_cast)

Překladače vs. interpretry, skriptovací jazyky

- rozdíly mezi překladačem a interpretrem
- výhody/nevýhody překladačů a interpretrů
- typické příklady použití překladačů a interpretrů
- bytecode, virtuální stroj (zásobník vs. registry), JIT, příklady (Java, Lua)
- optimalizační techniky u interpreterů: method caching, direct threading, value tagging
- typické vlastnosti skriptovacích jazyků (dynamické typování, reflexivita, dynamičnost)
- typické použití skriptovacích jazyků (skripty, web,...), důvody

19.2 Materiály

- Web předmětu Principy překladačů
- Web předmětu Konstrukce překladačů

Kapitola 20

Analýza a návrh softwarových systémů

Kapitola 21

Zdroje

- Materiály přednášky Software design and implementation na univerzitě v Severní Karolině
- Ian Sommerville — Software Engineering
- Podklady k přednášce Modelování a realizace programových systému Karla Richty
- Formální metody specifikace – příklady, další zdroje

Kapitola 22

Zpracování jednotlivých otázek

22.1 Algebraické specifikace, formální popis datových struktur

Příklad — množina INTů:

```
sorts: SET, INT, BOOLEAN          | vsechny vyuzivane typy
operations:                       | popisuji syntaxi daneho ADT
  - empty:            --> SET     | generator
  - insert: SET x INT --> SET     | generator
  x delete: SET x INT --> SET
  x member: SET x INT --> BOOLEAN
axioms:                           | popisuji semantiku specifikovanych operaci
  member(empty(),j) = false       | axiomy pro vsechny co nejsou generator se vsema moznyma ..
  member(insert(S,j),k) =         | generatorama na vstupu
    if (j=k) then true else member(S,k)
  delete(empty(),j) = empty()
  delete(insert(S,j),k) =
    if (j=k) then delete(S,j)
             else insert(delete(S,k),j)
```

22.2 Modelově orientované metody

Z

- Zdroj: The Z Notation: a reference manual

Formální specifikace je matematický popis (informačního) systému, který přesně popisuje, co má systém dělat, ale neříká jak.

Jazyk Z používá "matematické" datové typy popsané pomocí predikátové logiky (nezávislé na počítačové reprezentaci).

Dekomponuje specifikaci do malých částí zvaných schémata, ty popisují statické i dynamické aspekty systému.

- statické aspekty:
  - stavy
  - invarianty
- dynamické aspekty:
  - možné operace
  - vztahy mezi vstupy a výstupy
  - změny stavů

Schéma může popisovat i transformaci jednoho pohledu na systém na jiný, který přidává více detailů (při korektní implementaci původní abstrakce). Postupným zjemňováním (refinement) specifikace lze dospět až ke konkrétnímu programu.

Schéma

Každé schéma má svůj název a obsahuje:

- deklarace proměnných
- vztahy mězi proměnnými (odděleno vodorovnou čárou)

Příklad:

- (převzat ze zdroje)

Základní typy pro účely příkladu: DATE
Schéma popisující "prostor stavů" (state space): <table border style="border-collapse: collapse;"> <tr><th>BirthdayBook</th></tr> <tr><td> known: P(NAME) // množina
birthday: NAME → DATE // funkce </td></tr> <tr><td> known = dom birthday </td></tr> </table>
Schéma popisující operaci: <table border style="border-collapse: collapse;"> <tr><th>AddBirthday</th></tr> <tr><td> ΔBirthdayBook // Δ značí, že operace mění BirthdayBook (konvence)
name? : NAME // ? označuje vstup (konvence)
date? : DATE </td></tr> <tr><td> name? ∉ known
birthday' = birthday ∪ {name? → date?} // apostrof označuje novou hodnotu </td></tr> </table>

- operace zachovává invarianty předchozího schématu
- podobně jako ? označuje vstupní proměnné, ! označuje výstupní
- podobně jako Δ značí operaci modifikující stav, Ξ (Xí) značí operaci zachovávající stav

Skládání schémat

Schémata lze skládat pomocí operátoru =^ (rovnítko se stříškou) a logických operátorů (∧ a ∨).
Enum (pro účely příkladu):
REPORT ::= ok | already_known | not_known
Složené schéma:
RAddBirthday =^ (AddBirthday ∧ Success) ∨ AlreadyKnown;

- schéma Success vrací ok (typu REPORT), AlreadyKnown vrací already_known (pokud name? ∈ known)

<table border style="border-collapse: collapse;"> <tr><th>Success</th></tr> <tr><td>result! : REPORT</td></tr> <tr><td>result! = ok</td></tr> </table>

- složené schéma RAddBrithday vrací ok v případě úspěchu AddBirthday a already_known jinak
- RAddBrithday by šlo nadefinovat přímo, ale jeho definice vztahů by byly nepřehledně složité

Další syntax

- ∀ z : N • x ≤ z
  - znamená x je menší než všechna přirozená čísla (neboli x je nula)
- birthday' = birthday ⊕ {name? → date?}
  - předefinovává funkci birthday v bodě name? na novou hodnotu (za opertárotrem ⊕ může být i více-prvková množina)

(Více viz zroj...)

VDM

- zdroj: wen:Vienna Development Method

Skupina technik založená na specifikačním jazyce (VDM-SL).
Má variatnu VDM++, která podporuje objektově orientované a "concurrent" systémy.
Umožňuje modelování na vysoké úrovni abstrakce a převod na implementaci pomocí zjemění specifikace (refinement). Má spustitelnou podmnožinu, kterou je možné testovat.
Vydáno jako ISO standard 1996 (ten definuje ASCII syntax pro věci, co jsou jinak popsány "matematickou" syntaxí).
Jazyk je hezky popsán ve wikipedii (wen:Vienna Development Method).

22.3 Analýza algoritmů

- **validace** = "Are we building the right product?"
  - nebo jinými slovy — jde o kontrolu, zda-li daný produkt odpovídá reálným požadavkům
- **verifikace** = "Are we building the product right?"
  - neboli — jde o kontrolu, zda-li daný produkt odpovídá výchozí specifikaci

Hoareova logika

- wen: Hoare logic
- ...v příkladech by David Stotts (UNC)
- cílem je, aby se dala formálně dokazovat korektnost programů pomocí rigorózních prostředků matematické logiky
- základem je Hoare triple, popisující jak kousek kódu změnil stav výpočtu — $\{P\}C\{Q\}$
  - kde P a Q jsou assertions a C je příkaz. P nazýváme precondition, Q postcondition. Obojí jsou formule predikátové logiky.
- Hoaerova logika (dále jen HL) obsahuje axiomy a odvozovací pravidla pro všechny konstrukty jednoduchého imperativního programovacího jazyka
- Standardní HL poskytuje pouze **partial correctness**, neboli říká, že pokud před provedením C platí P, pak po jeho provedení platí Q, **nebo** C neskončí. Existuje ale rozšíření poskytující **total correctness**.

Pravidla jsou:

- Empty statement axiom schema:

$$\frac{}{\{P\}\ \textbf{pass}\ \{P\}}$$

- Assignment axiom schema:

$$\frac{}{\{P[x/E]\}\ x := E\ \{P\}}$$

- Rule of composition:

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

- Conditional rule:

$$\frac{\{B \wedge P\}\ S\ \{Q\}\ ,\ \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

- While rule:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

- Consequence rule:

$$\frac{P' \rightarrow\ P\ ,\ \{P\}\ S\ \{Q\}\ ,\ Q \rightarrow\ Q'}{\{P'\ \}\ S\ \{Q'\}}$$

- While rule for total correctness:

$$\frac{\{P \wedge B \wedge t{=}z\}\ S\ \{P \wedge t{<}z\}\ ,\ P \rightarrow t{\geq}0}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

Jak se to celé používá je hezky vysvětleno tady, nebudu se to snažit zreplikovat.

Dynamická logika

- viz wen: Dynamic logic (modal logic)
- Navrhl Vaughan Pratt v roce 1974
- Je rozšířením modální logiky zaměřené na dedukci o počítačových programech (ovšem později se začla využívat i jinde)
- Modální logika je charakteristická svými dvěma modálními operátory:
  - $\Box p$ — říká že p platí vždy
  - $\Diamond p$ — říká že p může platit
    - platí pro ně $\Box p \leftrightarrow \neg\Diamond\neg p$ a $\Diamond p \leftrightarrow \neg\Box\neg p$
- DL přidává ke dvou termům logik prvního řádu (tvrzení a data) ještě třetí typ termu — akci
- Dynamická logika (dále jen DL) toto rozšiřuje přidáním dvou modálních operátorů pro akce:
  - $[a]$, kde $[a]p$ znamená, že po provedení akce a musí p platit
  - $\langle a\rangle$, kde $\langle a\rangle p$ znamená, že po provedení akce a může p platit
    - opět samozřejmě platí $\langle a\rangle p \leftrightarrow \neg[a]\neg p$ a $[a]p \leftrightarrow \neg\langle a\rangle\neg p$
- DL umožňuje skládání akcí, pro jejich zápis se dá využít notace regulárních výrazů:
  - $a|b$.. provede se a nebo b
  - $a;b$.. provede se nejprve a, pak b
  - $a*$.. a se provede 0 nebo vícekrát
- Dále jsou k dispozici konstantní akce:
  - 0.. nic neudělá a neskončí (aka BLOCK)
  - 1.. nic neudělá a skončí (aka NOP)
- Krom defaultních jsou definovány tyto axiomy:
  - A1. $[0]p$
  - A2. $[1]p \leftrightarrow p$
  - A3. $[a|b]p \leftrightarrow [a]p \wedge [b]p$
  - A4. $[a;b]p \leftrightarrow [a][b]p$
  - A5. $[a*]p \leftrightarrow p \wedge [a][a*]p$
  - A6. $p \wedge [a*](p \rightarrow [a]p) \rightarrow [a*]p$ (s akcí n:=n+1 odpovídá matematické indukci)
  - speciální axiomy
  - A7. axiom schema $[x := e]\Phi(x) \leftrightarrow \Phi(e)$, odpovídá přiřazení z Hoareovy logiky. Tedy $\Phi(e)$ odpovídá fli ve které jsou všechny výskyty x nahrazeny výrazem e.
  - A8. $[p?]q \leftrightarrow p \rightarrow q$, kde p? je speciální akce definována ke každému tvrzení taková, že p? odpovídá NOP pokud je p true, jinak je to BLOCK. If p then a else b se dá tedy zapsat jako $(p?;a)|(\neg p?;b)$
- $x :=?$ znamená přiřazení libovolné hodnoty do x, a tedy $[x :=?]$ odpovídá obecnému kvantifikátoru, zatímco $\langle x :=?\rangle$ odpovídá existenčnímu

$\{p\}a\{q\}$ z HL se dá v DL zapsat jako $p \rightarrow [a]q$

Stejně jako HL se v ní nedají elegantně vyjádřit konkurentní chování (zatímco sekvenční se zapisuje hezky). To ovšem jde v TL (temporální).

Temporální logika

- viz wen:Temporal logic
- jestli někdo vůbec nemá tucha o TL, tak tady je to celkem hezky popsané, i když trochu delší http://www.cs.utexas.edu/user
- přináší prvek času, pravdivost tvrzení se tedy může v závislosti na čase měnit
  - například "mám hlaď" je pravdivé jenom někdy (třeba teď ;), díky temporální logice můžeme být proto přesnější a říct "budu mít hlad dokud se nenajím"
  - přínos pro formální verifikaci je zřejmý — snadno můžeme zapsat tvrzení typu "kdykoli příjde požadavek, zařízení může být zpřístupněno, ale v žádném případě nebude současně zpřístupněno dvěma žadatelům"
- je několik variací — **linear temporal logic** (AKA LTL, existuje jen jedna časová linie) a **branching logic** (více možných časových linií = nedeterminismus, zřejmě má stejnou sílu jako lineární)
- dva druhy operátorů
  - logical operators (klasika: $\neg, \vee, \wedge, \rightarrow$)
  - modal operators (operátory, které říkají například že fle musí někdy v budoucnu platit, nebo že musí platit odteď nafurt)

Modální operátory v LTL jsou následující (přeloženo z wikipedie):

Textově
N ϕ
G ϕ
F ϕ
ψ **U** ϕ
ψ **R** ϕ

Dalo by se zredukovat na dva z těchto operátorů, protože vždy platí:

- **F** ϕ = **true U** ϕ
- **G** ϕ = **false R** ϕ = $\neg$ **F** $\neg\phi$
- ψ **R** ϕ = $\neg(\neg\psi$ **U** $\neg\phi)$

22.4 Petriho sítě

- viz wen: Petri net
- Interactive Tutorials on Petri Nets
- matematická reprezentace diskrétních distribuovaných systémů
- vynalezl je Carl Adam Petri ve své Ph.D práci v roce 1962
- orientované bipartitní grafy ze dvěma druhy uzlů — místa a přechody
- nedeterministické! (není přesně definováno ktery z možných přechodů se nastartuje)

Formálně: Petriho síť je pětice (P, T, F, M_0, W), kde:

- P je seznam míst
- T je seznam přechodů
- F je seznam hran (žádná hrana nesmí spojovat dvě místa nebo dva přechody, tedy $F \subseteq (P \times T) \cup (T \times P)$)
- $M_0 : P \rightarrow \mathbb{N}$ je iniciální označkování, v každém místě $p \in P$, existuje $n_p \in \mathbb{N}$ tokenů
- $W : F \rightarrow \mathbb{N}^+$ je množina váh hran, které každé hraně $f \in F$ přiřazují $n \in \mathbb{N}^+$ které označuje kolik tokenů je při přechodu danou hranou navazujícím přechodem zkonzumováno, případně kolik tokenů výchozí přechod generuje
- stav sítě je vektor, kde v každé položce je počet tokenů aktuálně obsažený v místě s příslušným indexem

- další vlastnosti
  - state-transition list
    - seznam postupně navazujících přechodů mezi možnými stavy sítě
  - state-transition matrix
  - reachability
    - otázka, zda-li je daný stav dosažitelný z výchozího. Řeší se pomocí kreslení grafu dosažitelnosti (kde každý vrchol je možný stav a hrana je přechod mezi stavy). Graf se musi budovat do šířky, protože může být nekonečný a tak bychom se při cestě do hloubky snadno mohli stratit ..
  - liveness
    - vlastnost přechodů. Pokud je každý přechod L_k *live*, pak totéž můžeme říct o síti jako celku.
    - přechod je:
      - L_0 *live* (AKA dead), právě když nemůže být fired (neexistuje firing sequence která by ho obsahovala)
      - L_1 *live*, právě když může být fired
      - L_2 *live*, právě když pro každé celé kladné k může být fired k-krát
      - L_3 *live*, právě když existuje firing sequence kde je přechod fired nekonečně
      - L_4 *live* (AKA live), právě když je přechod v každém dosažitelném stavu L_1 *live*
  - boundedness
    - síť je k-omezená, pakliže v každém dosažitelném stavu má v každém místě vždy nejvýše k tokenů. Petriho síť je ohraničená, pakliže má její graf dosažitelnosti konečný počet vrcholů.
- možná rozšíření
  - barevné petriho sítě (tokeny je možno rozlišovat)
  - prioritizované petriho sítě (přechody mají priority)
- hierarchie petriho sítí
  - **state machine** (každý přechod má 1 in a 1 out hranu => může vzniknout konflikt)
  - **marked graph** (každé místo má 1 in a 1 out hranu => může vzniknout konkurence)
  - **free choice** (hrana je buď jediná která vystupuje z místa, nebo jediná která vstupuje do přechodu => může být konflikt i konkurence, ale ne najednou)
  - **extended free choice** (taková, která se dá transformovat na FC)
  - **asymetric choice** (konkurence i konflikt povoleny, ale ne asymetricky)
  - **petri net** (všecko povoleno)

22.5 Vyjadřovací prostředky a metody návrhu IS

Celý název tématu: Vyjadřovací prostředky a metody (datové modelování, procesní modelování — funkční a dynamické) strukturované analýzy a návrhu informačních systémů

Pro velkou provázanost zde popsáno spolu s konceptuálním modelováním, E-R schámaty a 3NF.

Společné zdroje:

- Konceptuální modelování a návrh databáze (slajdy VUT Brno; pdf)

Datové modelování

- wen: Data modeling

Zabývá se vytvářením datového modelu `informančního` sytému (na základě požadavků zadavatele či klienta). Entity datového modelu reprezentují objekty reálného světa. Datový model může být různého typu, např. objektový nebo relační. Model se tá popisovat různými formalismy (např. pomocí teorie množin a lambda kalkulu), dnes se typicky používá UML (model tříd).

Konkrétnímu modelu nějakého typu se říká instance modelu. Instance datového modelu může být tří typů (úrovní). (Jde o mírně historický pohled – ANSI, 1975):

- konceptuální model – popisuje doménu problému, vyjmenovává třídy entit a relace mezi nimi
- logický model – přidává sémantiku (dle použité technologie, např. definuje sloupce tabulek nebo třídy OO modelu)

- fyzický model – přidává detaily fyzického uložení (diskové oddíly, tablespaces, ...)

Idea rozdělení je, že úrovně jsou relativně nezávislé (např. konceptuální model může mít podobu ER-diagramu a logický může být OO modelem), v ranné fázi modelování lze pracovat jen s konceptuálním modelem, pak zpřesňovat.

Viz také:

- #E-R schémata a jejich transformace do relačního modelu
- wen:Relational model
- UML (diagram tříd)

Procesní modelování (funkční a dynamické)

Procesní modelování má využití v různých případech. Modelujeme stávající procesy organizace, která poptává IS. V tomto případě jde o deskriptivní procesní model, slouží pro analýzu požadavků. Užitečný je i model popisující procesy po nasazení IS (preskriptivní model). Další využití procesního modelování je v optimalizaci obchodních procesů (příp. pouze k jejich dokumentaci).

Procesní model zachycuje procesy, je dobré aby alespoň hlavní proces měl definován cíl. Jednotlivé procesy zahrují:

- začátek (iniciální aktivita; v aktivity diagramu UML plný puntík)
- aktivity (v UML zobrazeny jako obdelníčky se se zakulacenými rohy)
- vstupy/výstupy (signály; obdélníček s levou/pravou stranou ve tvaru šipky doprava)
- rozhodování (kosočtvereček)
- (aktéři) (panáček)
- konec (konečná aktivita, může jich být víc; zakroužkovaný puntík)

(V předmětu Vedení DB aplikací a jazyk UML jsme kreslili aktivity (i samotné procesy) jako obdélníčky s oběma bočními stranami ve tvaru šipek.)

Používané notace:

- wen:Business Process Modeling Notation (BPMN; by OMG)
- UML (activity diagram)

TODO: funkční vs. dynamické ??

Konceptuální modelování, databázové modelování, implementace

- Viz #Datové modelování.

E-R schémata a jejich transformace do relačního modelu

- wen:Entity-relationship model

Transformace

- odstranění (rozepsání) složených a vícehodnotových atributů (1NF)
- převod silných entit na tabulky (relace)
- převod relací:
  - entity v relaci 1:1 lze reprezentovat jednou tabulkou
  - relaci 1:N reprezentujeme pomocí cizího klíče
  - relaci M:N pomocí relační tabulky
- reprazentace slabých entit pomocí tabulky a cizího klíče (jako 1:N)
- možnosti řešení dedičnosti:
  - tabulky pro nadtyp i podtypy
  - tabulky jen pro podtypy s opakováním sloupců
  - vše v jedné tabulce (+ rozlišovací flag)

Návrh relačních schémat v 3NF

- hezky a stručně popsané tady: Third Normal Form (3NF) (na James Cook University)
- a nebo jako obvykle — wen: Third normal form
- a nebo taky tady: Schema Refinement and Normalization (na Římské Univerzitě)
- Armstrongova pravidla
  - X je částí Y pak X -> Y
  - X -> Y pak XZ -> YZ pro každé z
  - X -> Y a Y -> Z pak X->Z

1NF

- Všechny atributy jsou jednoduchého typu
- (Takže třeba atribut datum, který se skládá ze dne měšíce a roku to nesplňuje).

2NF

- Pokud je v 1NF a pro kterýkoli wcs:kandidátní klíč je každý neklíčový atribut závislý na celém tomto klíči (nikoli na jeho části)
- (neboli žádný neklíčový atribut není závislý na části klíče; implikuje, že pokud v tabulce složené klíče nejsou, je tabulka ve 2NF)
- To nám zaručuje, že v tabulce nebudou pohromadě nesouvisející věci.
- příklad:
  - (student_id, student_name, class_id, class_name) se závislostmi student_id->student_name a class_id->class_name není v 2NF (protože klíč je [student_id, class_id] a například student_name není na tomto klíči plně závislé).

3NF

- Pokud je 2NF a každý neklíčový atribut je netranzitivně závislý na každém klíči (neboli mezi neklíčovými atributy nesmí existovat jiná než triviální závislost)
- Relace která není v 3NF se bude patrně týkat více věcí a hrozí tedy aktualizační anomálie.
- příklad:
  - (class_id, instructor, office) se závislostmi class_id->instructor, instructor->office není v 3NF (protože office je na klíči class_id závislá tranzitivně .. když bude tedy instruktor přiřazen více předmětům, přestěhuje-li se, musíme v této tabulce změnit všchny záznamy office s ním spojené)
- často v praxi stačí
- na rozdíl od BCNF se už nezabývá závislostmi mezi klíči

BCNF

- wen: BCNF
- Pokud platí X->A a A není v X, pak X je klíčem.
- příklad:
  - (s_id, s_name, c_id, c_name, date) s klíčí [s_id, c_id], [s_id, c_name], [s_name, c_id], [s_name, c_name] a existují závislosti s_id->s_name, c_id->c_name není BCNF (protože např. klíčový atribut s_name je funkčně závislý na klíčovém atributu s_id z jiného klíče).
- indikuje, že některé atributy se netýkají celku který je identifikován klíči (a tyto by měly být jinde)

TODO (?):

- **Dekompozice**
- **Syntéza**

22.6 Modely životního cyklu softwarových systémů

- hlavní zdroj — Ian Sommerville — Software Engineering (konkrétně 6tá edice, link vede na 8mou)
- **Waterfall model**
  - postupně se prochází přesně definoványmi fázemi:

  1. analýza a definice požadavků
  2. design systému a softwaru
  3. implementace a unit testing
  4. integrace a testování systému
  5. provoz a údržba

  - nevýhody
    - neflexibilní (teoreticky by se nemělo zpětně zasahovat do ukončených fází)
  - výhody
    - v každé fázi je dobře definováno co se bude dělat a z čeho vycházet
    - snazší pro management (uz na začátku se dá rozumně plánovat, ví se co se bude dít)
    - dá se očekávat robustní návrh systému
- **Evolutionary development**
  - idea je udělat jednoduchou verzi systému, která bude následně zpřesňována spolu s požadavky uživatele
  - 2 typy
    - exploratory development — snahou je spolu s uživatelem postupně prozkoumávat požadavky a postupovat směrem k cílovému systému (vždy se udělá to čemu všichni dobře rozumí)
    - throw-away prototyping — narozdíl od předchozího se tady zaměřujeme spíš na špatně specifikované části systému a ty se snažíme pomocí prototypů lépe definovat
  - nevýhody
    - špatně se kontroluje průběh procesu
    - často vznikne špatně strukturovaný systém (časté opravy a zásahy do předchozích částí, aby mohly být ty nově definované dodělány)
    - často jsou potřeba speciální nástroje
  - výhody
    - vyplatí se zejména u menších projektů (míň než 100kloc)
- **Formal systems development**

  1. požadavky a specifikace jsou detailně popsány v některém formálním jazyku
  2. další kroky jsou nahrazeny postupnými transformacemi až do podoby spustitelného kódu (postupně jsou tedy přidávány detaily které zpřesňují specifikaci a zároveň neporušují její predikáty)

  - výhody
    - jednotlivé malé kroky jsou snadno vystopovatelné
    - "snadno" se dá dokázat správnost programu
  - nevýhody
    - potřeba speciálních skillů (rozhodnot o následující transformaci je složité)
- **Re-use oriented development**

  1. analýza komponent
  2. úprava požadavků podle definovaných komponent
  3. design systému s využitím komponent
  4. vývoj ostatních potřebných částí a jejich integrace

  - výhody
    - redukce kódu který se bude psát (=snižování rizika chyby)
  - nevýhody

    * výsledek nemusí 100% odpovídat výchozím požadavkům
    * v případě využití COTS (Commercial Off-The-Shelf) komponent vzniká závislost na dalším vendorovi

- **Incremental development**
  - myšlenka je umožnit uživateli učinit některá potřebná rozhodnutí až v době kdy bude mít jasno

  1. definice požadovaných služeb systému, přidělení priorit
  2. definice počtu jednotlivých inkrementů, rozdělení jednotlivých služeb k inkrementům které se budou dělat
  3. v každém inkrementu se pak detailně dospecifikují služby které se budou dělat a proběhne jejich vývoj, výsledek je nasazen k uživateli

  - * v každém inkrementu se může na danou část použít jiná metodika vývoje

- 1. v dalších fázích jsou pak nově vzniklé inkrementy integrovány do stávajícího systému

  - z této metody vychází extreme programming (kde jsou dané myšlenky dotažené do extrému ;)
  - výhody
    * zákazník nemusí čekat až do konce, než bude moct aspoň něco používat
    * zákazník může uplatnit u pozdějších inkrementů experience, kterou nabyde používáním těch předchozích
    * menší riziko celkového selhání
    * nejdůležitější části systému jsou nejlépe otestované (byly vytvořené v ranných inkrementecha a s každým přírůstkem testovány znovu a znovu)
  - nevýhody
    * inkrementy mají být malé a může být složité na ně namapovat požadavky customera
    * některé komponenty budou využívány mnoha částmi systému a zpočátku nemusí být přesně vidět jejich všechny požadované vlastnosti

- **Spiral development**
  - místo popisu procesu jako sekvence ativit s backtrackingem zpět je využito spirály
  - každá jedna otočka spirály se skládá ze čtyř sektorů:

  1. stanovení cílů aktuální smyčky — identifikace omezení a rizik, stanovení plánů
  2. ohodnocení a redukce rizik (definovaných dříve)
  3. vývoj a validace — výběr metody v závislostí na rizicích (pokud je riziko třeba špatné UI, budeme prototypovat)
  4. plánování — revize otočky spirály a plány co dál

  - na rozdíl od ostatních se v tomto modelu explicitně pracuje s riziky

22.7 Plánování a řízení projektů

- hlavní zdroj — Ian Sommerville — Software Engineering (konkrétně 6tá edice, link vede na 8mou)
  - další odkazy:
    * wen: Program Evaluation and Review Technique
- project management je iterativní proces, který končí až končí vlastní projekt (!)

vše by se dalo popsat následujícím pseudokódem:

```
Establish the project constraints
Make initial assesments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
  Draw up project schedule
  Initiate activities according to schedule
  Wait ( for a while )
  Review project progress
  Revise estimates of project parameters
  Update the project schedule
```

```
  Renegotiate project constraints and deliverables
  if ( problems arise ) then
    Initiate technical review and possible revisions
  end if
end loop
```

- **Plán projektu**
  - úvod — cíle projektu a omezení
  - organizace projektu — popis organizace týmu
  - analýza rizik
  - definice hw a sw požadavků
  - dekompozice práce
  - harmonogram projektu
  - popis monitorování a reportingu
- milestone — ukončení nějaké podfáze projektu, vznikají při něm výstupy pro management
- deliverable — výstup který se předává zákazníkovi (specifikace, design, ..)
- deliverable je milestone, ale milestone nemusí být deliverable (u milestone totiž může jít o nějaké interní dokumenty)
- **Rozvrhování projektu**
- první rozvrh je vždycky moc optimistický, je proto třeba ho průběžně aktualizovat
- jde hlavně o rozpad jednotlivých prací na aktivity a jejich rozvrhnutí tak, aby ty co mohou běželi paralelně a naopak navazovaly ty které jsou závislé
- ale důležitý není jenom čas, je potřeba zajistit taky dostatek prostředků (ať už jde o lidi, nebo třeba místo na disku)
- nástroje
  - activity graph — graf návaznosti aktivit (task = čtvereček, milestone = kolečko, závislost = hrana)
    - kritická cesta — nejdelší cesta ze startu do cíle, její natáhnutí znamená natáhnutí projektu
  - gannt chart (aka activity bar chart) — horizontální osa ukazuje čas, sloupečky pak jednotlivé tasky
    - popisují stejnou informaci jako acivity graph
    - dají se v nich líp znázornit rezervy tasků
- **Řízení rizik**
- riziko = pravděpodobnost že nastane některá nežádoucí událost
- základní rozdělení rizik
  - project risks — rizika ohrožující plán či zdroje projektu
  - product risks — rizika ohrožující výkon nebo kvality vyvíjeného produktu
  - business risks — rizika ohrožující organizaci nebo vlastní vznik softwaru
- risk management je opět iterativní proces, který se musí dít v průběhu projektu, nejen na začátku (!)
- základní kroky risk managementu
  - risk identification — identifikace možných rizik (jako: technology risks, people risks, organisational risks, tools risks, req. risks, estimation risks)
  - risk analysis — zhodnocena pravděpodobnost a dopad jednotlivých rizik
    - nejde o přesná čísla, ale spíš rozsah psti (jako: very low .. $< 10\%$, low .. 10 — 25%, moderate .. 25 — 50%, high .. 50 — 75%, very high .. $> 75\%$)
    - u dopadu podobně (jako: catastrophic, serious, tolerable nebo insignificant)
    - výsledkem je prioritizovaná (a průběžně aktualizovaná) tabulka rizik
  - risk planning — výběr strategie řízení každého z rizik
    - minimalizace psti.
    - minimalizace dopadu
    - přijmutí rizika (když nastane, budu se s tím umět vyrovnat)
    - přesun rizika na někoho jiného (pojištění)
  - risk monitoring

Alokace zdrojů

- DOKONČIT
- když máme připraven rozpad práce na tasky, je potřeba přiřadit zdroje nutné k vykonání daného tasku
  - tzn. třeba i lidi (i když označovat je za zdroje není moc hezké ;)
- ve velkých společnostech může být problém s pracovníky kteří jsou specialisti — prodloužení jeho práce na jiném projektu může zasáhnout i do našich plánů
- ke znázornění se hodí výše popsané activity graphs a gant charts

Použití metrik

- cílem je získat pro nějaký atribut sw produktu numerickou hodnotu
  - porovnáváním takových hodnot se dá sledovat např. vývoj kvality
- problém je že neexistují žádné obecně uznávané standardy a tedy ani rozšířené tooly
- příklady:
  - počet řádek kódu
  - počet reportovaných chyb ve výsledném produktu
  - počet člověko-dní potřebných k vývoji komponenty
- wen: COCOMO = COnstructive COst MOdel — slouží k odhadu počtu člověko-měsíců, které bude vývoj sw. produktu trvat
- **control metrics** — metriky asociované s procesem
  - např. jak dlouho trvá odstranění nalezeného defektu
- **predictor metrics** — metriky asociované s produktem
  - např. počet atributů a operací třídy, nebo průměrná délka identifikátorů
- extrení vlastnosti software se nedají přímo měřit
  - jde třeba o komplexnost či pochopitlenost
- proto je snaha najít vztah mezi vnitřními (velikost software) a vnějšími metrikami
- k tomu je třeba:
  - přesné měření interního atributu
  - existence vztahu mezi tím co umíme změřit a ext. atributem
  - pochopení daného vztahu a jeho vyjádření v podobě vzorce nebo modelu
- **proces měření**
- fáze:
  - výběr měření která se budou provádět — je třeba vědět co chci změřit
  - výběr komponent k posouzení
  - měření charakteristik komponent
  - identifikace měření vybočujících z řady
  - analýza vybočujících komponent
- klíčové ovšem je naměřená data zaznamenávat a mít k dispozici při dalších měřeních
- **metriky produktů**
- dají se rozdělit na:
  - dynamic metrics — měřené na běžícím programu
  - statické metriky — měření nějaké reprezentace systému (program, dokumentace)
- příklady:

  - fan-in — počet volání fce X (hodně znamená že je fce úzce svázaná se zbytkem)
  - fan-out — počet fcí které jsou z fce X volané
  - lenght of code — větší kód = komplexnější = náchylnější k chybám
  - cyclomatic complexity — měří komplexnost programu
  - lenght of identifiers — delší = víc samovysvětlující = vyšší čitelnost
  - depth of conditional nesting — větší hloubka = horší srozumitelnost
  - fog index — čím delší slova a věty v dokumentu, tím je méně srozumitelný (třeba pro dokumentace)

Řízení kvality

- **quality assurance** — zřízení frameworku organizačních procedur, které vedou k vysoké kvalitě
- **quality planning** — výběr patřičných procedur z daného frameworku a jejich adaptace pro specifický projekt
- **quality control** — definice a ustanovení procesů, které zaručují, že jsou procedury a standardy dodržovány vývojovými teamy
- o quality management by se měl starat nezávislý team
- quality management by měl být oddělen od project managementu
  - neměl by být tedy závislý na nějakých schedulech a budgetech
- ke kvalitě existuje sada standardů ISO 9000 (ISO 9000-3 se vztahuje přímo k sw. vývoji)
  - ISO 9000 -> jeho instancí je Organisation quality process -> jeho instancí je Prject quality management
- dva typy QA standardů
  - product standards
  - process standards
- časte jsou standardy považovány za zbytečnou byrokracii (a každý se snaží najít důvod proč nejsou v tom kterém projektu potřeba)
- jak lidi přesvědčit?
  - zapojit je do vývoje standardů — musí rozumět motivaci
  - průběžně standardy aktualizovat podle aktuálních potřeb
  - poskytnout tooly pro podporu
- standardy dokumentace
- jsou důležité, protože dokumentace je jediná hmotná součást vyvíjeného software
- jde o:
  - document process standards — proces v němž dokument vzniká
  - document standards — definují strukturu dokumentů
  - document interchange standards — zaručují kompatibilitu dokumentů
- quality planning
- musí definovat co přesně vysoká kvalita znamená (což je u sw někdy složité — viz požadavek na dobrou udržovatelnost)
  - problém je, že výroba sw. je spíš kreativní proces, takže hodně záleží na tom kdo to dělá
- quality plan by měl být as short as possible (jinak ho nikdo nebude číst)
- je hodně parametrů které je možné optimalizovat (jako: safety, robustness, modularity, portability, usability, efficiency, lernability)
- quality control
- je potřeba kontrolovat dodržování stanovených standardů
  - quality reviews — QA team kontroluje vybraný proces či produkt
  - automated software assessment — automatická kontrola metrik

Stupně zralosti sw. týmů (CMM)

- process improvement = porozumění stávajicímu procesu a jeho změna k lepšímu
- W. E. Demming — po 2. světové válce pracoval na zlepšování japonského průmyslu — aplikoval statistickou kontrolu procesů v průmyslu .. a byl hodně úspěšný.
- proces má podobně jako produkt charakteristiky, např:
  - understandability
  - visibility
  - reliability
  - maintainability
- process improvement má následující klíčové fáze:
  - process analysis
  - improvement identification
  - proces change introduction
  - process change training
  - change tuning
- při snaze o zlepšování procesu je klíčové zjistit, co a jak měřit — GQM paradigm:
  - Goals
  - Questions
  - Metrics

Capability Maturity Model

- vyvinut by Software Engineering Institute (na Carnegie-Mellon University, financuje to americké ministerstvo obrany)
- původní záměr bylo mít prostředky k zjištění schopností potenciálního contractora, který bude něco dělat pro US DoD
- klasifikace sw. procesů na 5 úrovní:
  - **Initial level** — neexistují efektivní manažerské postupy a plány. Možná jsou zformalizovány postupy, ale není nijak kontrolováno jejich dodržování.
  - **Repeatable level** — existují formální manažerské, QA i configuration control procesy. Organizace je schopna úspěšně zopakovat projekty stejného typu jaké byly vykonány dříve. Výsledek projektu ale závisí spíš na konkrétních lidech, manažerech.
    - key process areas: sw. configuration management, sw. QA, sw. subcontract management, sw. project tracking and oversight, sw. project planning, requiremnts anagement
  - **Defined level** — organizace má proces dobře definovaný, takže má základ pro kvalitativní zlepšování. Existují formální procedury ke kontrole dodržování procesu.
    - key process areas: peer reviews, intergroup coordination, sw. product engineering, integrated sw. management, training programme, organization process definition, organizatio process focus
  - **Managed level** — je formálně definován nejen proces, ale i program sběru informací o probíhajících procesech. Data o procesech a produktech jsou sbíráná a na jejich základě probíhá zlepšování.
    - key process areas: sw. qualty management, quantitative process management
  - **Optimising level** — organizace provádí průběžný process improvement. Zlepšování procesů ma plán i budget a je implicitní součástí vnitřních procesů.
    - key process areas: process change management, technology change mgmt, defect prevention
- dá se dobře aplikovat na velkých organizacích (pro malé organizace může být už příliš byrokratický)
- tento model má ale i své nevýhody:
  - zaměřuje se výhradně na project management, ne na product development (nezohledňuje možnosti jako prototyping, formální metody ..)
  - vůbec se nezabývá analýzou rizik

  - není dobře popsáno v kterých typech organizací se dá CMM využít a kde už moc ne
  - organizace která splňuje 80% levelu 2 a 70% levelu 3 dostane pořád level 1 rating (= k postupu je potřeba splnit 100%)

- při srovnání s ISO 9000 se dá říct, že organizace na levelu 2 až 3 jej splňují, ale dá se najít i organizace na levelu 1 která ISu odpovídá. Přesto mnoho key areas s ISem koresponduje.

22.8 CASE systémy

- What is a CASE Environment? (na CMU)

22.9 Třívrstvá struktura informačních systémů, klinet/server

- wen: Client-server architecture

22.10 XML a značkovací jazyky

- zdroje:
  - wen: Markup language
  - Slajdy z přednášky o XML (podle mne by měl stačit obsah 1. přednášky)

22.11 Objektová analýza a návrh (UML)

22.12 Informační bezpečnost

Kapitola 23

Operační systémy (státnice)

Zdroje:

- The Hebrew University in Jerusalem Lectures Notes: The Hebrew University in Jerusalem

see also Operační systémy (předmět)

23.1 Struktura operačního systému

- monolit (vše v kernel mode)
- mikrokernel (client-server model)
- hybridní kernel (wen:Hybrid kernel ... některé služby kompatibility u NT jádra běží v user mode)
- layered system
  - používáno v Multicsu, dnes už ne
  - 6 vrstev: alokace CPU, paměť, přístup ke konzoli, IO, user démony, operátor
  - každá vrstva požívala výhod abstrakce toho co bylo pod ní (paměťová správa se nemusela starat o CPU a tak...)
- virtuální stroje: uvnitř "virtual machine monitor", který dalším vrstvám poskytuje několik čistých kopií hardware. na tom pak může běžet víc různých OS

architektura mikrojádra, abstrakce poskytované mikrojádry

- v kernel mode jen address space management, přepínání vláken a ipc
- zbytek (filesystémy, síťové protokoly, ovladače zařízení) v userspace serverech, jen mají přístup k paměťovým rozsahům svých specifických zařízení
- když nějaký server zhučí dá se restartovat

`http://www.tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/#AEN22` syscalls `http://www.cs.purdue.edu/homes/cs354/LectureNotes/CS354-part3.pdf`

23.2 Virtuální stroje

Původně pro IBM/360 – řešili multitasking pomocí virtuálního stroje. Dnes simulují buď celý systém (VMWare), nebo jen prostředí pro jednu aplikace (Java Virtual Machine, .NET Common Language Runtime). Použití pro abstrakci hardware, izolaci (např. při hostingu), provozování více OS na jednom počítači.

wen:Popek and Goldberg virtualization requirements: virtualizace je možná, jen pokud jsou všechny instrukce které závisí na stavu procesoru (např registrech ochrany paměti) nebo ho mění privilegované – tj. všechny instrukce které by se mohly tlouci předají nejdřív kontrolu hypervisoru, který je může simulovat

Podmínky splňuje například IBM/360. x86 tyto podmínky nesplňuje, hodně neprivilegovaných instrukcí je citlivých – odhaluje stavy procesoru, nejde například simulovat privilegovaný mód v user mode. Tohle poprvé vyřešil VMWare dynamickým překladem, Intel i AMD představily rozšíření pro IA-32/64 i AMD64, které tohle řeší.

Softwaru, který se stará o to, aby aplikace, které běží v guest OS, žily v iluzi, že běží na skutečném stroji a ne na tom virtuálním, se říká hypervisor nebo také VMM (Virtual Machine Monitor).

Hypervisor může být typu I nebo typu II.

- Typ I běží přímo na skutečném HW. Při jeho implementaci je tedy třeba starat se o hodně low-level věci a sahat přímo na železo (není k dispozici žádná abstrahující vrstva). Na druhou stranu je to nejrychlejší (ve smyslu efektivity) způsob, jak dělat virtualizaci.
- Typ II je od skutečného HW oddělen operačním systémem a běží v něm jako (víceméně obyčejná) aplikace. Takže je o něco snazší ho napsat (máme k dispozici vrstvu, která abstrahuje úplně low-level věci), ale každá citlivá instrukce, která vyvolá trap, se musí propagovat přes víc vrstev, a je to tedy pomalejší než typ I.

Docela pěkně je to popsané tady: OK Labs virtualization
Asi by bolo dobré vedieť povedať aj niečo napr. o JVM — ako tam vyzerajú inštrukcie (srandy ako výroba objetkov? :-)), JIT atď...

23.3 Správa procesů a vláken, plánování

- vlákno má kontext procesoru: stavy registrů, stack
- proces má kontext paměti (adresový prostor, data v něm), a prostředí: terminál, otevřené soubory, env, ...

proces může mít víc vláken, plánují se vlákna.
plánovač určuje, které vlákno pustit dál. typické metriky:

- doba odezvy (u interaktivních procesů)
- propustnost (co nejvíc dokončených úloh za jednotku času)
- využití procesoru (neflákat se)
- spravedlnost

off-line plánování: mám pevný počet procesů, vím jak dlouho poběží, nepřerušuji je

- first come first served
- shortest job first: minimalizuje průměrnou dobu odezvy

on-line plánování

- procesy se objevují neočekávaně, doba běhu neznámá
- plánuje se podle priority, vázanosti na IO, interaktivity, chování v minulosti (výpadky stránek)
- preemptivní: potřebuje podporu HW (časovač), možnost měnit plán na základě nových informací
- context switch – přepnutí na jiný proces/vlákno

metody:

- first come first served: nepreemptivní
- round robin: střídám vlákna v časových kvantech, pak přepínám – spravedlivé
- prioritní s více frontami a zpětnou vazbou: některé úrovně FIFO, jiné RR, přesouvám podle toho jak dlouho už běžely

pro více CPU:

- každý procesor má vlastní frontu
- mezi nimi vyvažování zátěže, zohlednění vztahu procesů

real-time systémy:

- běh omezen reálným časem dokončení – deadline

metriky (doba odozvy / spravodlivost / vyuzite cpu / priepustnost) inverzia priorit
multiCPU

- (ne-)distribuovane planovanie
- zviazanost procesov s cpu

realtime planovanie (soft/hard)

23.4 Komunikace a synchronizace procesů, kritické sekce, synchronizační problémy a primitiva

Komunikace – posílání zpráv, přístup ke sdíleným datům
kritické sekce

- část programu která potřebuje exkluzivní přístup ke sdílenému prostředku (typicky paměti)
- (v kernelu část která nemůže být přeplánovaná nebo přerušena)

Synchronizační primitiva:

- semafor: celočíselná proměnná + fronta čekajících procesů... může reprezentovat počet volných/přidělených prostředků
- fronty zpráv: operace send + receive, receive blokuje když není zpráva
- monitor: konstrukce programovacího jazyka, ošetřující kritické operace; v monitoru může být jen jeden proces najednou
- ^ vzájemně ekvivalentní

synchronizační problémy:

- obědvající filozofové
- producent-konzument
- holič

memory model (kompilatory + optimalizacie multithread apliacii)
pipe / signaly (len jeden thread procesu — pthread_sigmask())
posielanie sprav, zdielanie pamati
kriticke sekcie (blokovanie ineho mimo CS, nekonecne cakanie, 1 proces v CS, bez predpokladov o cpu)
metody bez primitiv (dosledne striedanie, petersnovo riesenie)
aktivne cakanie (TSL, zakazanie prerusenia) | pasivne

- mutex, semafor, RWL, monitor, condition variable (wait(+mutex)|signal|bcast)
- fronty sprav (receive zablokuje bez spravy)

convoys, priority inversion, starvation & deadlock
nonblocking synchronizacia, okrem ferovosti aj

- wait free (vsetky skoncia v konecnom case)
- lock free (niektore)
- obstruction free (kazdy po konecnom pocte krokov — ak ostatne nic nerobia)

http://en.wikipedia.org/wiki/Lock_convoy

uváznutí a jeho řešení

Coffmanovy podmínky

1. Výlučný přístup: prostředek je přidělen výhradně jednomu procesu
2. Neodnímatelnost: prostředek nejde odejmout
3. Drž a čekej: proces může zároveň držet prostředek a čekat na další
4. Kruhová závislost: procesy čekají na prostředky v kruhu

Prevence deadlocků: napadení jedné z Coffmanových podmínek

- výlučný přístup: spooling (např. u tiskáren)
- neodnímatelnost
  - jen tam kde jde bez následků (procesor, paměť)
  - většinou nejde bez selhání procesu

- drž a čekej: nutno žádat o všechny najednou, nebo před další žádostí dosavadní prostředky uvolnit
- kruhová závislost
  - očíslování prostředků, pak jde žádat jen o prostředky s vyšším číslem

Předcházení: zabránit realizaci všech podmínek

- bankéřův algoritmus: vím, co bude proces max. chtít k dokončení, bezpečný stav: jsem schopen plně uspokojit alespoň 1 proces, on mi pak svoje prostředky vrátí
- složité rozhodování
- informace typicky nejsou dostupné

Detection & recovery: řešení deadlocků, až nastanou

- graf závislostí + hledání cyklu
- pak odebrání prostředku pod dohledem operátora
- nebo přerušení cyklu zabitím procesu
- recovery:
  - os ukládá stav procesů, při deadlocku se vrátí k savepointu a pustí je s jiným naplánováním
  - transakční zpracování: abort + pustit znovu (typické pro databáze)

pštrosí algoritmus:

- předstíráme, že problém neexistuje
- když už nastane, vyřeší ho za nás uživatel

23.5 Podpora multiprocesorových systémů

- SMP: víc procesorů, stejná paměť (multi-core procesory jsou taktéž SMP, ne vždy však ekvivalentní více procesorům — například u intel core sdílí jádra L2 cache, u některých architektur dokonce některé koprocesory)
- NUMA: víc procesorů, paměť je ale pro každý procesor jinak dostupná (někde pomalejší)
- hyperthreading: jedno jádro, zdvojené části co berou instrukce, takže se jeví jako dva

cache coherency: zajištění, že procesor namá ve své cache zastaralý obsah (zneplatněný jiným procesorem)

1. co procesor zapsal, může i přečíst, pokud tam někdo jiný mezitím nezapsal (chceme vždycky)
2. pokud mezitím do paměti zapsal procesor A, můžou to hned přečíst i ostatní CPU
3. zápisy musí být v daném pořadí: pokud do jednoho místa zapíšu hodnotu A a potom hodnotu B, čtení z kteréhokoli CPU mezi tím musí nejdřív ukázat A a pak B, nikdy naopak

postupy:

- snooping (bus sniffing): řadič cache kouká na sběrnici, co kdo píše do paměti, podle toho si invaliduje svoje záznamy (cache buď musí být write-through, nebo se použije nějaký model níže)
- snarfing: jako předchozí, ale rovnou si podle zápisů upravuje svoje cachovaná data
- directory-based: centrální seznam cachovaných bloků (pro velké systémy nad 64 CPU, kde není centrální sběrnice)

`http://www.cmpe.boun.edu.tr/courses/cmpe511/fall2004/Mehmet%20Senvar%20-%20Cache%20Coherence%20Protocol`
`ppt`

modely:

- MSI – stavy Modified/Shared/Invalid
  - čtení: M/S – dodá data, pokud je v I, tak se ověří jestli jinde není M (ta pak musí zapsat a přejít do S nebo I)
  - zápis: M – modifikuje; S – musí říct ostatním S, ať si to vyndají; I – ostatní S musí vyndat, M zapsat
- MOSI

  - přidává stav Owned, kdy cache vlastní daný blok a dodává data ostatním cache

- MESI – Modified (jen v této cache), Exclusive (jen v této cache, asi odpovídá hlavní paměti), Shared (může být i jinde), Invalid
  - jde psát jen ve stavech M/E
  - cache v M/E stavu musí poslouchat na všechna čtení a dodávat svůj obsah (M), nebo přejít do S (E)
  - Read For Ownership (RFO) – broadcast signál indikující přechod cache ze stavu S do E – ostatní si musí blok přehodit do I

- MOESI
  - Modified: má aktuální data, ty v hlavní paměti jsou stará, jiné cache nemají nic
  - Owned: má aktuální data, ty v hlavní paměti jsou stará, ostatní cache mohou mít data v Shared stavu
  - Exclusive: má aktuální data, totožná s hlavní pamětí, jiné cache nemají nic
  - Shared: má aktuální data, můžou je mít i ostatní cache; aktuální data jsou i v hlavní paměti, leda že by je měla nějaká cache v Owned stavu
  - Invalid: nemá nic, aktuální jsou buď v hlavní paměti nebo v jiné cache

NUMA

Bez cache coherence problematické použití, proto se používá cache-coherent NUMA (ccNUMA), kdy radiče cache mezi sebou zajišťují koherenci. To zpomaluje zejména v situacích kdy na stejné místo sahá víc procesorů rychle za sebou, takže operační systémy co to podporují alokují paměť a procesory tak se takový provoz minimalizoval.

23.6 Mechanismus přerušení v OS

see also Architektura počítačů a sítí#Přerušení

Přerušení předchází nutnosti pollovat zařízení. Místo toho dá zařízení signál procesoru, ten předá řízení OS, který uloží stav aktuálního procesu, spustí obsluhu přerušení (zjištění podle přiřazeného IRQ kanálu, pak se pouštějí rutiny pro ovladače co tak jsou), a nakonec se vrátí k původní činnosti.

Přerušení se dají maskovat, aby nerušily (např. při obsluze jiných přerušení).

Přerušení mohou být:

- asynchronní: vyvolané vnějším zařízením
- synchronní: vyvolané přímo procesorem
  - výpadek stránky, systémové volání (trap)
  - chybná instrukce, dělení nulou (exception)

Aby ošetření přerušení nezdržovalo a zbytečně se neblokovaly další události, je obsluha rozdělena na samotnou obslužnou funkci, která udělá to nejnutnější, a další zpracování (bottom half, softirqs, tasklets v Linuxu, deferred procedure calls ve Windows), která se jen naplánuje na později a třeba dál zpracovává příchozí data pro aplikační programy.

DMA

see also Architektura počítačů a sítí#DMA (Direct Memory Access), wen:Direct memory access

Direct memory access: Přenos dat mezi zařízením a pamětí se odehrává bez účasti procesoru, ten k němu akorát dá pokyn a na konci dostane přerušením info že už je hotovo. Procesor se tak nejen nemusí otravovat s kopírováním, ale ani nemusí na zařízení čekat. Kopírování pak provádí buď DMA řadič (ISA sběrnice), nebo zařízení samo (bus mastering u PCI). (Při DMA je třeba dát pozor na cache procesoru.)

Dříve DMA řadič čekal na signál procesoru že je volná sběrnice, eventuálně číhal kdy ji nevyužije nebo procesor úplně uspával. Dnes už jsou sběrnice pro procesor/hlavní paměť (northbridge) a periferie (southbridge) oddělené`https://dsrg.mff.cuni.cz/pipermail/osy/2005-February/000148.html`.

23.7 Správa periferií, ovladače zařízení

OS kontroluje periferie a zařízení, kód který je řídí se označuje jako ovladače zařízení. Umožňuje přistupovat k zařízení více aplikacím, a zároveň poskytuje abstrakci přístupu k nim.

Driver se skládá ze synchronní části, volané když aplikace po zařízení něco chce, a části asynchronní, volané když přijde přerušení od zařízení. Tyto část mezi sebou komunikují pomocí front a bufferů – aby se v přístupu netloukly, dovoluje operační systém obsluhu přerušení odložit na později (bottom halves etc `http://dsrg.mff.cuni.cz/~ceres/sch/osy/text/ch04s01s01.php`)

API: blokující funkce, asynchronní signalizace, signalizace chyb.

V Unixu zařízení rozdělená na bloková zařízení (náhodný přístup k adresám, cache, ...) a znaková (jen čtení/psaní, bez cache). V Linuxu výpis zařízení podle sběrnic, namatchování k driverům.

Ovladače se dají připojovat z běhu systému (.ko u Linuxu, .sys u Windows).

Scatter/Gather (vectored) I/O `http://en.wikipedia.org/wiki/Vectored_I/O`

23.8 Správa paměti, hierarchie pamětí, segmentace, stránkování, strategie alokace, odkládání

see Architektura počítačů a sítí#Paměťová hierarchie, vyrovnávací paměti

see Architektura počítačů a sítí#Stránkování

see Architektura počítačů a sítí#Segmentace

TLB – asociativní paměť, cachuje položky stránkovacích tabulek; při přepínání adresových prostorů nutno splachovat `http://www.informit.com/articles/article.aspx?p=29961&seqNum=4`

PAE — `http://www.tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/#AEN221` (aj s ukazkou strankovacich tabuliek pre rozne kombinacie noPAE/PAE/4kB/4MB pages)

23.9 Sdílení paměti mezi adresovými prostory, paměťově mapované soubory

V Linuxu obsahuje blok sdílené paměti (shmem) seznam všech procesů, které jej mají připojený. Když se proces pokusí paměť použít, dojde k výpadku stránky, při jehož řešení buď alokuje novou stránku, nebo použije už alokovanou, existuje-li `http://www.linux-tutorial.info/modules.php?name=MContent&pageid=293`. Když se stránky vyswapuje, tak je potřeba opravit tabulky stránek pro každý proces který ji používá `http://www.tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/#AEN220`

Při forku je paměť procesů jen označená jako sdílená, pak se používá copy-on-write.

mmap: soubor se namapuje do paměťového prostoru aplikace, a označí se jako vyswapovaný. Pak při přístupu do paměťi dojde k výpadku stránky, kus souboru se nakopíruje do rámce a v paměti a používá se.

pomenovanie zdielanych oblasti

mmap / swap

- problemy (velkost suboru (nezaokruhlena na napr. 4kB) vs. velkost stranok)

23.10 Souborové systémy, souborové a adresářové služby, síťové souborové systémy

Souborové systémy dovolují přistupovat k v podstatě lineárnímu prostoru na zařízení jako ke stromu adresářů a souborů, řeší přístupová práva, rozmístění souborů na disku a tak. Soubory mají různé atributy, moderní filesystémy mají žurnálování, aby se předešlo nutnosti všechno kontrolovat při výpadku. Příklady: FAT, ext2, NTFS.

Souborové a adresářové služby: ?

Síťové souborové systémy: problémy se zamykáním a stavovostí

see also Distribuované systémy#NFS, AFS, CODA, ...

- NFS – funguje přes RPC, je bezestavový, problémy většinou ignoruje; NLM – zamykání; od verze 4 NFS stavový
- SMB
- AFS – hodně serverů v jedné struktuře, lokální cache (kde se provádí všechny změny, po zavření se to nakopíruje zpátky); server o cache ví, a když se v souboru něco změní tak klienta upozorní
- Coda – dá se operovat i off-line, replikace, odolnost proti výpadkům, občas nutno řešit konflikty

LDAP

- DN (CN + RN), atributy (+schema), X.500, TLS...

open => syscall

- copyfn z userspace
- najst volny fd
- filp_open
  - open_namei => dentry
    - zacne v root/pwd (fsystem+dentry)
    - prehladava dentry_cache, potom realdentry (disk?), pokial moze, po komponentoch
    - (prechod medzi fs?)
  - dentry_open(dentry, mnt)
- ulozit fd do files

23.11 Informační bezpečnost a základy šifrování

ruzne crypto-loopy, ssl, pam, acl, capabilities a tak
hash
sifrovanie

- asymetricke (RSA, diffie-hellman)
- symetricke

podpisovanie
autentikacia / autorizacia

23.12 Síťové služby OS

sockety, RPC.
Při zpracování paketů je žádoucí zabránit zbytečnému kopírování – nechávat místa okolo pro hlavičky.

- packet filtering – iptables a kamarádi
- packet scheduling – různé fronty na výstupu (SFQ, priority, HTB)

Aplikace: síťové filesystémy, load balancing, clustery
sockety

- socket+bind / connect
  - PF_UNIX, PF_INET, PF_NETLINK; SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW
- listen(backlog) + accept
- send(to,msg)/recv(from,msg)
- select(rd,wr,xcept,timeout), poll(fds,timeout)

RPC

- XDR => skeletony+stuby (rpcgen)
- portmapper (cislo sluzby+verzia => port)

kopirovanie paketov (=znizeny vykon)

- data dispatching / smart hardware

iptables (dorucovanie, forwardovanie, zahadzovanie paketov, maskarada)
scheduling

- Stochastic Fair Queue (per process queue+round robin; hashovanie do queues+obcas zmena h)
- token bucket

  - ked treba ohlidat bandwidth; flowu priradeny bucket
  - odoberanie tokenov ked odosiela data, pravidelne pridavanie
  - velkost bucketu = fluctuation limit
- class based ???
- random early detection
  - pre TCP a spol. — ked sa zaplna, pakety su zahadzovane s vacsou pst. => lepsi flow control (early warning)

sietove filesystemy / load balancing / clustery
NFS

- architektura (protokoly, hlavne operacie)
- obsah file handle v NFS
  - fs identifier, node #, generation # (nedostupne z userspace) => export iba celeho FS — nie subtree
- close-to-open cache consistency
  - uplne sync narocna => nerobi sa
  - pri zatvarani suboru sync, close() moze vratit chyby servera
    * potom sa vezme getattr() — ak rovnake ako pri dalsom open, cache sa vyuzije (inak zahodi)
  - cache atributov a/alebo dat sa da zakazat
- problemy bezstavovosti (testovanie pristupovych prav, mazanie otvorenych suborov, zamykanie suborov) + riesenia
  - read pomocou gen#
  - append — malo casty, nedolezity

Kapitola 24

Distribuované systémy

zdroje

- Middleware a Middleware notes
- Principy distribuovaných systémů a jejich prezentace
- `http://www.kiv.zcu.cz/~ledvina/Prednasky-DS-2007/`

24.1 Pozadavky

Komunikace, zasílání zpráv, RPC. Skupinová komunikace, virtuální synchronie, doručovací protokoly. Middleware (klasifikace, protokoly, RMI, EJB, CORBA, DCOM, SOAP, ...). Logické hodiny a jejich synchronizace. Distribuované synchronizační algoritmy. Distribuovaný konsensus. Distribuované sdílení paměti, konzistenční modely. Souborové a adresářové služby, distribuované souborové systémy (NFS, AFS, CODA, ...), replikace. Distribuovaná správa prostorů jmen, identifikace objektů a přístup k nim, služby (LDAP, JNDI, CORBA Namig/Trading). Procesy v distribuovaném prostředí, migrace procesů, vyvažování zátěže, zablokování.

24.2 Komunikace

Zasílání zpráv

V distribuovaném prostředí je hromada problémů se sdíleným adresním prostorem -> komunikace pomocí zpráv
nespolehlivý unicast – to co nám dává hardware, počítáme s best effort

- v IPv4 se pakety mohou na routerech fragmentovat po cestě, v ipv6 vyrazí s nějakou délkou a s tou i dorazí (pokud po cestě nějaká část sítě neumí danou délku přenést, tak je zahodí).

Spolehlivý unicast

chceme aby nám přišel každý paket a přišel nám jen jednou (exactly once sémantika)

- ochrana proti poškození (duplikace, checksums, parita, křížová parita, CRC)
  - forward error correction – když zjistím poškození, doplňuji další opravné informace
- ochrana proti ztrátě – potvrzování
- ochrana proti duplikaci – unikátní ID paketů (při TCP handshake se dohodne náhodné počáteční a pak roste)
- jiné problémy – když jedna strana spadne, zapomene jaká čísla paketů poslala atd

TCP – emuluje stream, pakety jsou číslované, mají dvoubajtový kontrolní součet

flow/congestion/etc

flow control ochrana proti ucpání příjemce; řeší se posíláním "flow control window" počet/velikost zpráv, které příjemce může dál sežvýkat, posílaný v ACK zprávách

congestion control ochrana proti ucpání sítě; "congestion control windows" si upravuje příjemce podle toho jak se mu ztrácí zprávy

Požadavky na real time/propustnost (soft – většinou splněny, hard – splněny vždy) musí být garantovány hardwarem, nad tím se pak dá zajistit rezervace

- **throughput** – propustnost (množství dat za jednotku času)
- **latency** – jak dlouho to trvá (doba na one way trip nebo roudtrip)
- **jitter** – rozptyl latence (výkyvy latence, dobré pro stanovení rozumné stanovení velikosti bufferů streamových aplikací)

Protokol **RSVP** (reservation protokol)

- odesílatel posílá Path zprávy, aby dal najevo uzlům po cestě že je tam nějaká session co něco chce
- příjemce posíla opačným směrem Resv zprávy, aby vyznačil cestu dalším směrem a dal najevo co se má rezervovat

RTP (Real Time Protocol) přenáší real-time data, k němu je RTCP (Real Time Control Protocol) se statistickými zprávami, ze kterých se vyhodnocují jitter, latency a asi i troughput

RTSP (Real Time Streaming Protocol) používá se k dohadování streamingu, nepřenáší data, ovládá třeba RTP, který ty data nese.

Multicast

Zpráva jde k více příjemcům, ale posílá se jen jednou. (Broadcast – ke všem uzlům v síti).

V TCP/IP se k ohlašování příslušnosti k multicast skupinám používá IGMP (Internet Group Management Protocol). Router s numericky nejnižší IP v každém segmentu periodicky posílá Membership Query, uzly odpovídají Membership report. Uzly, které opouští nebo přichází do skupiny posílají State Change Report Další protokoly řídí routování multicast zpráv.

spolehlivost:

- sender initiated protocols – odesílatel ví o všech příjemcích, ti mu posílají co přišlo; když je příjemnců moc máme ACK implosion problem
- receiver initiated protocols – příjemce ví, co má přijít, jinak posílá NAK; když má hodně příjemců problémy, máme NAK implosion problem
  - dá se řešit pomocí posílání NAK multicastem a čekáním náhodný interval, jestli už někdo neposlal NAK dřív
- stromově, pak node posílá lokální ACK (pro flow control) a agregované ACK (když už přide ACK od všech pod ním, kvůli zapomínání poslaných paketů)
- kruh s tokenem – uzel s tokenem posílá ACK odesílateli, uzly bez tokenu posílají NAK uzlu s tokenem
  - Např.**Reliable Multicast Protocol**. Příjmci v logikém kruhu, jeden uzel má token. Odesílatel pošle multicastem zprávu, uzel s tokenem pošle multicastem ACK spolu s globálním pořadím zprávy. Ostatní uzly mohou poslat NACK (s označením preferovaného příjmece s tokenem??). Každá zpráva obsahuje lokální pořadové číslo. Token se předává po poslání ACK, ten, kdo přebírá token ho nemůže převzít, dokud nedoručil všechny zprávy s nižším pořadovým číslem.

interfacy:

- **blokující** × **neblokující** (s callbackem nebo pollováním)
- **synchronní** (dokončení operace znamená že příjemce zprávu přijal) × **asynchronní** (operace se vrátí hned po odeslání)
- **Pozor**, může být např asynchronní a blokující akce (např, pokud jsou plné odesílací buffery), není to nesmysl... Stejně tak může být podle nějakého výkladu synchronní neblokující akce — např s callbackem, za dokončení se počítá až provedení callbacku.

RPC

Myšlenka je v tom, že na klientu se zavolá funkce, která se provede na serveru. Realizace je nakreslená ve skriptech na PDS,

- Nejprve se vygeneruje UUID rozhraní
- Programátor dodá definici rozhraní (interface definition file)
- IDL kompilátor udělá header, co se naincluduje do klienta i serveru
- IDL kompilátor taky udělá klientskou a serverovou část, která se stará o komunikaci (client stub, server skeleton)

- Programátor dodá implementaci serverové části
- Celé se to zkompiluje a slinkuje.
- Server si zaregistruje u nějakého directory serveru to, že dělá tenhle servis
- Klinet se rozběhne, když si chce zavolat RPC, tak
  - Normálně zavolá fci.
  - Vygenerovaný Stub si vytvoří zprávu (marshaling), předá jádru jádro, look-upne službu na directory serveru, předá jí jádru serveru, to ji dá skeletonu, ten si ji rozbalí (unmarshaling), a putuje to podobně zpět.
- marshalling / unmarshalling – balení a rozbalování parametrů funkcí do zpráv
- rozhraní může být i ze signatury v "normálním" jazyce, jen s nějakými doplňujícími informacemi
  - IDL – popisuje rozhraní funkcí, z něj se pak generují skeletony a stuby; pak mapování do jazyků
- varianta s objekty – máme proxy objekty u klienta a servanty na serveru. Implementace tohohle objektovýho cirkusu je třeba **RMI**.

Paralelizace na serveru

- single threaded
- thread per connection
- per request
- thread pool (šetří se overhead na vytváření vláken)

24.3 Skupinová komunikace

Posílání zpráv od jednoho odesílatele více příjemcům. Problémy

- Atomicita (všem nebo nikomu)
- Synchronizace (nějaké pořadí)
- Adresování (adresování skupin)
- Technické řešení (multicast, posloupnost unicastů, broadcast?)

Otevřenost skupin

- Uzavřené (posílat mohou jen členové)
- Otevřené, posílat může kdokoliv

Uspořádání skupiny

- Rovnocené
- Hierarchické
- S koordinátorem

Virtuální synchronie

Group view – množina uzlů ve skupině (též group membership, delivery list, etc). Značí se L (globální), L_i (lokální verze procesu i), L^x (verze pohledu x), ${L_i}^x$

Algoritmus spolehlivého doručování je virtuálně synchronní, když:

1. všechny uzly ve skupině udržují stejný L
2. pokud je zpráva m odeslána skupině s L^x před změnou na L^{x+1}
   - buď m doručí všechny uzly z L^x před provedením změny na L^{x+1}
   - nebo žádný uzel z L^x, který provede změnu na L^{x+1}, zprávu m nedoručí

Virtuální synchronie vlastně říká, že pokud někdo přibude nebo odejde ze skupiny, tak se všechny uzly shodnou na tom, které zprávy byly odeslány před touto změnou, a které po ní.

Přitom nemusí platit, že přijetí zprávy členem L implikuje doručení všem členům L (nespolehlivost, havárie odesílatele).

Když se uzly B a C dozvědí, že uzel A přestal být členem jejich skupiny, už od něj pak nemůžou přijímat skupinové zprávy.

doručovací protokoly

Rozlišením mezi přijetím a doručením zaručují nějaký typ uspořádání, nejčastěji kauzální uspořádání

- Source ordering — zprávy vyslané jedním uzlem dojdou v pořadí, vjakém byly odeslány
- Causal ordering — zprávy dojdou podle seřazené podle kauzální závislosti
- Total ordering — všichni účastníci komunikace vidí zprávy v nějakém, ale stejném pořadí pro všechny

Doručování zajišťující uspořádání zpráv

- Myšlenka je v rozdělení přijetí a doručení zprávy. Ikdyž zprávu fyzicky mám, logicky jí nedoručím, dokud není správný čas...

Kauzální doručování pro jednu skupinu

- Vektorové hodiny pro každou zprávu a každý proces (dékla je počet procesů ve skupině). U procesů nastaveny na začátku na samé nuly.
- Při odesíláni zprávy si svoji složku vektoru zvednu o jedna, a svůj stav vektorových hodin přibalím ke zprávě
- Po přijetí zprávu pozdržet, dokud značka ve zprávě nebude
  - V hodnotě náležeící odesílateli o jedna menší, než má přijímající proces (mám předešlé zprávy od něj)
  - V ostatních hodnotách menší nebo rovna než hodnoty co má k dispozici přijímající vektor (kauzální závislost na ostatních)

Kauzální doručování pro překrývající se skupiny

- Totéž, ale posílaji se vektory vektorových hodin (s každou zprávou včechny vektory skupin, ve kterých je odesilatel)
- Musí být zaručena kauzalita vzhedem ke skupině, **kam to je poslané**
- Pro všechny ostatní skupiny, jihž je příjemce členem, musí být také zaručena kauzalita.

Total Order protokol

- Odesílatel rozešle zprávu, všichni mu na ni odpovědí zprávou s timestamp, kdy ji dostali a odesilatel odešle finalizační zprávu s nejpozdější časovou známkou, (se známkou toho, kdo ji dostal nejpozději). Tento nejvyšší čas, který všechny procesy dostanou ve finalizační zprávě je čas, kdy mohou zprávu doručit.

Spolehlivé doručovací protokoly

záplavový algoritmus při každém přijetí zprávy, kterou uzel ještě neviděl, ji pošle všem ostatním – spolehlivé a neefektivní

algoritmus s potvrzováním

- p_s odesílatel, p_r (z L) je příjemce, p_x uzel co havaroval
- p_s odešle zprávu všem v L, schová ji dokud nedostal ACK od všech, nebo než nazná že zhavarovali
- p_r po příjmu odešle ACK p_s, zprávu si schová než zjistí že všichni ostatní přijali
- jestliže p_r zjistí, že p_s havaroval, odešle zprávu všem z L, o nichž neví, že ji dostali

Jak p_r zjistí které uzly zprávu přijaly?

- Třeba tím že se ACK posílají taky všem – neefektivní, pokud se to stejně nedělá broadcastem nebo multicastem
- Ack se dají nalepovat na jiné zprávy, a využitím kauzality – při korektním kauzálním doručování jsou potvrzení tranzitivní
  - D může z příjmu a, b+ACK(a), c+ACK(b) odvodit, že B přijal a, a C přijal a i b
  - z příjmu b+ACK(a), c+ACK(b) může D odvodit, že B přijal a, C přijal a i b, a taky že A poslal a (a vyžádat si jeho resend)

====== Trans algoritmus ====== Uzel si udržuje kauzální graf zpráv které přijal a ještě je nemají všichni, z došlých potvrzení si vypočítává další, přeposílá zprávy od kterých dostal NAK, detekuje stabilní zprávy. Na protokol se dá hledět tak, že posouvá stable line grafem kauzální závislosti. Každý uzel si udržuje jen graf zpráv, které přijal ale ještě nejsou stabilní. Z toho plyne, že když někdo zhavaruje a zpráva se nestane stabilní, může mít neomezenou paměťovou náročnost. Je tedy potřeba doplnit o členství ve skupinách — transis algoritnus

====== Transis algoritmus ====== spolehlivý kauzální multicast + členství ve skupinách. Při pomalé odpovědi vyhodí uzel ze skupiny, ten se pak musí vrátit explicitně.

Založeno na konzistentních změnách pohledů a doručování v rámci pohledů.

- Při detekci havárie procesu p zpráva FAULT(p)
- kauzální hranice pohledu – vynutí doručení kauzální předcházejících zpráv, pozdrží zprávy následující (doručí se až v L^{x+1})
- dvě havárie zároveň – společná hranice
- kauzální doručení zpráv havarovaného procesu vzhledem ke změně pohledu
  - zprávy odeslané kauzálně před zjištěním havárie se doručí v L^x
  - zprávy odeslané konkurentně se zjištěním havárie se zahodí
  - zprávy odeslané kauzálně po zjištěním havárie se zahodí

====== ISIS protokol ====== Maticové hodiny – každý proces si udržuje vektor s odeslanými zprávami (sem si uklada, pri prijeti nejake dalsi zpravy od prijemce cas odelsani posledni zpravy od nej, kterou prijemce odstal), a zároveň vektory co mu došly od všech ostatních. Z toho zjistí které zprávy už mají všichni (a jsou stabilní).

Zaručení synchronie: když se proces dozví o novém pohledu, rozešle všechny nestabilní zprávy a potom potvrzení instalace (flush message), když pak dostane flush od každého procesu, může nový pohled nainstalovat. Každý proces si udržuje seznam "havarovaných" procesů dle aktuálního pohledu, ten se posílá se zprávami a sjednocuje při příjmu. Zprávy od havarovaných se zahazují.

řazení zpráv

- source ordering – zprávy jednoho odesílatele dorazí v takovém pořadí v jakém je poslal
  - stačí sekvenční číslování lokálně odesílatelem
- causal ordering – zprávy o události dorazí před zprávami o jejích následcích
  - pomocí vektorových hodin (see #Vektorové hodiny)
- total ordering – všem příjemcům přijdou všechny zprávy ve stejném pořadí
  - sériová čísla zpráv vydává centrální autorita

24.4 Middleware

Midleware je software umožňující nebo usnadňující běh aplikací nějakým způsobem rozložných po více počítačích...

Klasifikace

převzato z VŠE, berte s rezervou :-)

- communication middleware – zajišťuje přenos zpráv, vzdálené vykonávání kódu a tak podobně
  - synchronní – RPC, RMI (Remote Method Invocation) – SunRPC, DCOM, CORBA, Java RMI, SOAP
  - asynchronni – Message-Oriented Middleware
- data management middleware – přístup k datům
  - Remote Database Access – ODBC, JDBC, ADO.NET
  - Remote File Access
- platform middleware – běhové prostředí
  - Transaction (Processing) Monitor (TPM) – zajišťuje transakce – EJB — Java Transaction Service
  - Object Request Broker (ORB) – RPC v objektovém prostředí (+ life cycle services, naming services, ...)
  - Message Broker – zajišťuje doručení zpráv a tak (JMS – Java Message Service)

  - Application Server – kontejnery, které zajišťují standardizované služby pro aplikace (CORBA, .NET, J2EE EJB – perzistence, transakce, kontrola souběžných přístupů)

jiná struktura, podle lokiho

- messaging
- RPC
- data access
- kontejnery

Protokoly

Moc nevím, co tím myslely, možná třeba vědět co a k čemu jsou (už dřív popsané), které midleware může používat

- Reliable Multicast Protocol
- Resource Reservation Protocol
- Realtime Protocol
- Realtime Streaming Protocol
- RPC

Tohle tu bylo, ale podle mě je protokol jen SOAP. MPI je knihovna (rozhraní) a .NET Remoting je buhvico, ale asi ne protokol...

- SOAP – založené na XML, určené pro web services (zpráva obsahuje hlavičky pro routující systémy a tělo pro koncového příjemce)
- MPI (Message Passing Interface) – C/C++/Fortran knihovna (rozhraní) pro psaní paralelních aplikací komunikující pomocí zasílání zpráv, zprávy mají definovaný formát, jsou přenositelné, umí p2p, ale i skupinové metody broadcast, scatter, gather, reduce volitelnou funkcí). Komunikující procesy rozděluje do skupin...

.NET Remoting

Objektové RPC (podobné Java RMI). Hlavní idea je komunikace mezi doménami (různými aplikacemi), spojením se síťovými službami je pak jedno, jestli aplikace běží na jednom PC a povídají si přes paměť, nebo jestli běží na jiných strojích a posílají si zprávy.

- Marshaling používá tzv. data sink. Sink je interface, který zajišťuje zabalení zprávy (dostane objekt vyplivne stream dat). Jsou předimplementované 2 sinky (binární a HTTP). Binární kóduje data do vlastního formátu (klasická serializace objektů) a komunikuje přes TCP. HTTP balí data do SOAP zpráv a používá HTTP protokol, stejně jako třeba web services. Binární je efektivnější, HTTP zase lépe prolézá přes firewally.
- Při vytváření serveru se otevře zvolený port. Na něm poslouchá .NET a může na něm viset víc služeb. Služby se identifikují unikátním řetězcem. Každá služba může být
  - Singleton (jeden remote objekt sdílený přes všechna volání všech klientů)
  - Single Call (pro každé volání se vytvoří nový objekt, na kterém se volání provede)
  - Alternativně je možné používat Client Active, kdy si klient sám řeší vytváření instancí (ale to už je vyšší dívčí).
- Data se kopírují klasicky, objekty se předávají referencí (.NET automaticky generuje proxy objekty).
- Každý vzdálený objekt dostane time lease (dobu, jakou je platný) a sponsors. Když vyprší lease, ptáme se sponzorů jestli to ještě chtějí – stačí kladná odpověď od jednoho, takže je to lepší než pingování.

RMI – Remote Method Invocation

Jde vlastně o objektové RPC...

see also RMI tutorial

Objekty implementují Remote interface (kde navíc mohou házet RemoteException), implementace dědí z RemoteObject.

- Třída UnicastRemoteObject pro dočasné objekty
- Třída Activatable pro perzistentní objekty
- Klient si vyžádá referenci na objekt třeba v RMIregistry (naming udělátko).
- Lokálni stub (proxy), remote implementace
- Využití standardní serializace typů.

Lifecycle se vzdáleně řeší pomocí počítání referencí a keepalive zpráv – klient si při rozbalování příchozí reference na objekt vyžádá tzv. lease, ten pak periodicky obnovuje. Nakonci lease vrátí.

CORBA

wen:CORBA, CORBA EXPLAINED SIMPLY

- IDL, pak mapování do různých jazyků.
  - Problémy např s délkami integerů v C a C++, řeší se mapováním na třídy nebo typedefy.
  - Umí i složitější datové typy, např struktury, uniony, stringy, sekvence, pole, inteface typy (reprezentují objekty předávané referencí) nebo vyjímky — tam je zas problém jak je mapovat do C...
- O samotný přenos dat a komunikaci se stará ORB Core, která je k aplikacím přilinkovaná jako knihovna.
- Protokol GIOP (General InterORB Protocol), součást ORB (Object Request Broker) — definuje Common Data Representation – CDR a formáty zpráv, nadstavba IIOP (Internet InterORB Protocol), implementace GIOP pro internet (mapování GIOP na TCP/IP)
  - GIOP umí i location forward – zprávu, že požadavky mají teď jít na jiný server
- Messaging (hlavní dva typy zpráv, REQUEST a REPLY, celkem sedm druhů)
- stub/skeleton – jako u RPC
- proxy/servant – proxy objekt na klientovi, servant (implemntace objektu na serveru) na serveru – pro RMI
- POA – Portable Object Adapter – asociuje server s objekty — směruje volání buď do servantů (místních, nebo na jiné servery), demultiplexuje příchozí požadavky na server a spolupracuje s IDL
  - default servant – vyřizuje požadavky pro které není servant
  - servant activator – když není servant, vytvoří ho
  - thread pool – připravené thready k obsluze požadavků
  - servant retention policy – dá se úplně vypnout vedení info o servantech, všechny požadavky pak jdou na default nebo activator
- Naming Service – operace resolve a bind (viz Distribuované systémy#CORBA Naming.2FTrading)
- Trading Service – operace export, query Distribuované systémy#CORBA Naming.2FTrading
- operace se dají volat i neblokujícně, s callbackem nebo pollingem

DCOM

Microsoftí middleware pro RPC (s objekty). Místo IDL má MIDL (Microsoft Interface Definition Language), parametry jdou definovat jako in, out, několik typů pointerů (ref, unique, ptr podle tohle, jestli mohou být NULL a mohou být aliasované, tj jestli dva mohou ukazovat na tu samou paměť), taky pipe, který reprezentuje datový stream mezi klientem a serverrem.

Interface může být buď RPC nebo COM. COM verze musí dědit z IUnknown (metody QueryInterface, AddRef, Release) nebo IDispatch (GetTypeInfo, GetIDsOfNames, Invoke), a musí mít uuid atribut (unikátní id). Správa paměti pomocí počítání referencí, vzdáleně se to řeší před další IRemUnknown, které AddRef a Release před posláním serveru agreguje.

Kombinování objektů:

- Containment (nový objekt přeposílá volání vnitřním implementacím)
- Aggregation (export vnitřních interfaců; problém s QueryInterface, aby znal celý nový interface – nutná explicitní podpora agregace agregovaným rozhraním).

Když klient udělá nějaké volání, knihovna OLE32.dll se mrkne do System Reqisty a vyloví jak udělat stub a pomocí LPC (Local Procedure Call) nebo RPC zavolá skeleton a provede, co se po ní chce.

Reference na objekty se získávají buď přes factories, nebo nějakými metodami co vrací reference.

Servery mohou být buď in-process (linkované v DLL, adress space klienta, běží na tom samém stroji), nebo out-of-process (v samostatném procesu(v EXE), mohou běžet i na jiném stroji ve vlastním adresním prostoru).

Interfacy pro perzistenci, s metodami jako Load, Save a IsDirty.

JMS – Java Messaging Service

JAVA interface pro messaging middleware

- zprávy se vyměňují v rámci session se service providerem;
- předpokládá se existence messaging service providera, na něj se napojí přes JNDI (viz Distribuované systémy#JNDI)
- session je vázáno na jeden thread a stará se o pořadí a tak
- různé typy specifických obsahů zpráv (objekt, mapování, stream primitivních typů, text, stream bajtů)
- rozhraní Producer a Consumer, vytvářené volání metod ze Session; odesílání je blokující asynchronní; příjem je synchronní nebo asynchronní, dá se filtrovat co přijmout
- modely Point To Point (odesílatelé ukládají do fronty, příjemci vybírají; nejsou-li zůstavají zprávy ve frontě) a Publish Subscribe (kanál od odesílatelů příjemcům; když nejsou příjemci tak se zprávy zahazují – leda že by si někdo objednal durable subscription)

EJB

Prostředí pro komponentové aplikace. Beany obsahují logiku aplikace a bydlí v kontejnerech, které jim zajišťují přístup klientů, životní cyklus, perzistenci, transakce a tak. Volání metod u všech beanů je serializované.

- stateful session beans
  - Z pohledu klienta se objekt se vytváří když se na něj dodá reference, pak se stav inicializuje business metodou, další speciální metoda stav sundá;
  - Z pohledu kontejneru se bean aktivuje, pasivujce, stav se v kontejneru uchovává jako serializace tranzitivního uzávěru polí objektu.
- stateless session beans – jako stateful, ale odpadá aktivování/deaktivování
- message driven beans – požírají JMS zprávy, implementují JMS listener
- entities – reprezentují entity v databázi; vlastnosti instancí jsou perzistentní, odpovídají primitivním/serializovatelným typům a kolekcím. proměnná Id je primární klíč. Entity manažer poskytuje metody k vyhledávání. O perzistenci se stará kontejner.

Beany můžou mít definovaný stav vůči transakcím (neumíme, požadujeme v transakci, může být, pak taky co se má dělat když transakce je/není). Stav session beanu není v transakci a není ovlivněn commitem/rollbackem.

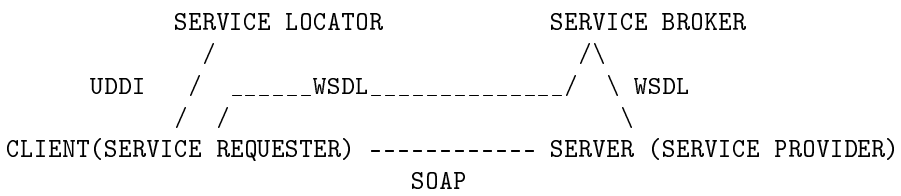
Transakce mohou být bean managed (pak commit nebo rolback startuje metoda beanu), nebo container managed, kde si bean nastavi nejake atributy a kontejner si podle toho nějak řídí commit a rollback, třeba atribut mandatory říká, že se metoda beanu musí vykonat ve volající transakci, a pokud taková neexistuje má se vyvolat vyjimka. Podobně never zas říká, že medora musí být vyvolána mimo transakci.

- business interface – samotné metody na beanu, dělají tu věc co se od beanu chce
- remote interface – proxy k beanu, třeba hází navíc výjimky typu "selhalo spojení"
- home interface – metody nejsou vázané na konkrétní instanci (třeba vytvoření nové instance, nebo vyhledání existující) (jen v EJB 2.1, ne v EJB 3.0... asi)

SOAP

- Simple Object Access Protocol — prototokol založený na XML, určený pro web services (přestože je to "protokol" definuje pouze formát zpráv)
- zpráva obsahuje hlavičky pro routující systémy a tělo pro koncového příjemce
- počítá se s přenosem přes HTTP
- umožňuje definovat datové typy, i složené (z toho plynou probémy s XML schema popisem SOAP souborů a tím s jejich validací)
- spolupracuje s WSDL, které popisuje webové služby v XML a s UDDI, které zas umí registraci, lokaci a klasifikaci webových služeb
- UDDI obsahuje:
  - White Pages — Naming Service
  - Yellow Pages — Tradin Service, používá globální specifikace properties (UNSPC, NAICS)
  - Green Pages — technické informace

Model spolupráce je přibližně



24.5 Logické hodiny a jejich synchronizace

Fyzické hodiny se nedají dostatečně přesně synchronizovat, takže se používají logické.

Lamportovy hodiny

Je důležité pořadí, nikoli přesný čas; nekomunikující procesy nemusí být synchronizovány.
Integer u každého uzlu, a:

1. Kdykoli proces zaznamená důležitou událost (generování zprávy), inkrementuje timestamp
2. Ke každé poslané zprávě přidá timestamp
3. Když proces p přijme zprávu m, aktualizuje si svůj timestamp: TS(p) = max(TS(p), TS(m))+1

Pak platí, že když událost A kauzálně předchází B, tak TS(A) $<$ TS(B). Opačná implikace ale neplatí. To řeší až vektorové hodiny.

Vektorové hodiny

Každý proces má svůj vektor hodin VT.

1. odeslání zprávy m procesem p_s
   - VT(p_s)[s]++; VT(m) = VT(p_s)
2. přijetí zprávy procesem p_r; proces pozdrží doručení, dokud
   - VT(m)[k] = VT(p_r)[k] + 1 pro k=s (tj. ve slotu odesílatele je zpráva o jednu napřed)
   - VT(m)[k] VT(p_r)[k] jinak
3. po doručení zprávy m si proces p_r upraví VT

    - VT(p_r)[k] = max(VT(p_r)[k], VT(m)[k])
- událost A kauzálně předchází B <=> timestamp A je menší než B
- událost A nemá kauzální vztah k B <=> timestamp A není porovnatelný s B

Maticové hodiny

Vektory vektorových hodin, proces si udržuje co ví o ostatních procesech (v jakém timestampu o nich naposledy slyšel) viz Distribuované systémy#ISIS protokol

24.6 Distribuované synchronizační algoritmy

- synchronizace fyzických hodin
- logické hodiny — lamport, vektorové, maticové
- vzájemné vyloučení
- detekce globálního stavu
- volba koordinátora
- členství ve skupinách

Synchronizace fyzických hodin

- V distibuovaném prostředí jsou fyzické hodiny celkem nanic (nedají se synchronizovat dostatečně přesně, aby k něčemu byli)
- Po nějaké době odchylka (x.t), jde určit po jaké době se je nutné synchronizovat k udržené nějaké maximální odchylky od správného času
- **Cristianův algoritmus** s jedním UTC serverem, který čas zná. Proces se na něj zeptá a nastaví si čas UTC s opravou danou odhadem chyby dané zpožděním komunikace a dobou zpracování požadavku serverem
    - Přenos po síti může mít klidně dobu obrátky v řádku 100vek ms, což přestavuje až stovky milionu instrukci na běžném procesoru (třeba geostacionární družice jsou ve výšce cca 36 000 km ... světlo tam letí přes 100ms, takže taková družice nemá dobrý ping :o))
- **Berkley algoritmus** sever se zeptá všech na čas, spočte průměr a pošle zprávy o kolik se má kdo opravit
- **Distribuovaný algoritmus** Bez koordinátora. Broadcast ve fyzicky nestejný čas, počítání průměru se zahozenými extrémy, oprava.

Vzájemné vyloučení

Není společná paměť, nutno synchronizovat přes zprávy.

- centralizované (s koordinátorem)
- princip soutěže (Lamport, Ricard-Agrawalla)
- volby (Maekawa)
- token-passing (Suzuki-Kasami)
- kruh, strom (Le Lann, Raymond)

Centralizovaný algoritmus

Jeden server s frontou, na žádost posílá potvrzení/zamítnutí/uvolnění

- ideově nevhodné, ale nejjednodušší a nejefektivnější
- výpadek serveru – ztráta informace
- výpadek klienta – vyhladovění

U všech následujících algoritmů je problém s výpadkem skoro libovolného procesu – řešit centralizovaný problém distribuovaným algoritmem nemá moc smysl.

Lamportův algoritmus

Proces vyšle žádost, a čeká až dorazí odpovědi od všech ostatních, a všechny žádosti v jeho frontě mají vyšší časovou značku.

- p posílá žádost Mp se svým timestampem
- přijetí žádosti od i: zapamatuje si žádost, pošle ACK s vlastním timestampem
- když dostane od někoho ACK, přidá si ho ke svému požadavku
- do kritické sekce proces vstoupí, když
  1. od všech ostatních dostal ACK
  2. a zároveň neví o žádném starším požadavku
- když skončí s kritickou sekcí, pošle ostatním release
- po přijetí release si proces vymaže k němu patřící žádost (a někdo další pak na základě toho může vlézt do kritické sekce)

Ricart & Agrawala

Proces chce vstoupit do kritické sekce

- zašle žádost ostatním a čeká na došlé odpovědi s potvrzením

Proces přijme žádost:

- jestliže není v kritické sekci a ani nechce, pošle potvrzení
- jestliže je v kritické sekci, neodpovídá a požadavek si zařadí do fronty
- jestliže do kritické sekce chce, porovná čas příchozí žádosti se s časem své vlastní
  - pokud je vlastní dřív, neodpovídá a zařadí do fronty
  - pokud je příchozí dřív, pošle potvrzení
- po opuštení kritické sekce pošle potvrzení všem procesům co má ve frontě

Princip voleb

Proces se snaží zíkat hlasy ostatních, kdo má nejvíc může do kritické sekce. Proces může v jeden okamžik hlasovat jen pro jednoho. Problém: jak počítat výsledky, kdy už proces ví že vyhrál. Při stejném počtu hlasů může nastat deadlock.

Maekawa – optimalizace komunikační složitosti pomocí volebních okrsků, pro vstup do kritické sekce je potřeba získat všechny hlasy z vlastního okrsku (podmínky: každé dva okrsky mají společného člena, velikost okrsků je konstatní, každý proces ve stejném počtu okrsků). Komunikační složitost odpovídá velikosti okrsků.

Prevence deadlocku – logické hodiny (proces ruší původní hlas, pokud ještě pak dostane žádost s nižším TS, realizace pomocí zprávy reject a přidělení hlasu procesu s nižším timestampem)

Optimalni rozděleni pro $M = K * (K-1) + 1$(M počet procesů, K velikost okrsku), reálně se rozdělují na pruniky vždy jednoho sloupce a řádku ve čtverci procesů.

Token based

Kdo má peška může do kritické sekce. Problém se ztrátou peška.

Detekce globálního stavu

- Množina událostí v systému $E = \{e\}$
- Řez c je rozdělení E na Pc a Fc : $P_c \cup F_c = E \; AND \; P_c \cap F_c = \emptyset$
- Konzistentní řez c :$a \to b \; AND \; a \in F_c \to b \in F_c$
- Konzistentní stav je Pc, tak, že c konzistentní řez.

Algoritmus detekce GS:

- Jeden iniciátor pošle značku, která říká že se jde detekovat globální stav.

- Po přijetí první značky si zapamatuju svůj stav (poslední odeslanou a přijatou zprávu/zprávy), a stav kanálů označím za prázdný a přepošlu značku
- Pak si pamatuju zprávy co mi chodí od uzlů, od kterých mi ještě nepřišla značka.
- Když mi od nějakého uzlu přijde značka (ale už jsem predtím od někoho značku dostal a detekuju), tak uzavřu stav kanálu od toho uzlu.
- Po přijetí všech značek konec, zapamatované stavy uzlů a kanálů definují konzistentní stav.

Volba koordinátora

Bully algoritmus

Předpokládá se omezená doba přenosu a zpracování zprávy kvůli detekci havárií.

- Když proces usoudi ze stavajici koordinator zhavaroval, rozhodne volit. Zašle zprávu všem procesům s vyšší identifikací (ProcessID)
  - Když přijde odpověď od nejakeho procesu z vyssim ID, process vycka nejaky zvoleny casovy interval jestli ten proces s vyssim ID se prohlasi za koordinatora. Pokud ne, zacne volby zas.
  - Když nepřijde nic, proces vyhrál, je novým koordinátorem a pošle o tom zprávu všem ostatním.
- Když proces přijme zprávu o volbě, vrátí svou odpověď a pošle žádosti všem vyšším procesům.
- Kdyz proces prijme zpravu od noveho koordinatora a zjisti ze ma vyssi Process ID nez ten koordinator tak se naštve :) a zacne volbu. Proto se tomu algoritmu se rika "bully" (z angl. je to žák týrající spolužáky)

Volba se provede ve dvou kolech (Proc?).
Asi protoze prijde vic odpovedi od procesu z vyssim PID a ty pak maji mezi sebou vyresit kdo je koordinator ve druhem kole.

Invitation algoritmus

Procesor může zhučet, při selhání komunikace se může síť rozdělit na izolované segmenty, zprávy se můžou ztratit – nelze spolehlivě detekovat havárii.
Idea: koordinátor je vázán na skupinu (všichni členové skupiny vidí stejného koordinátora), skupiny lze štěpit.
pravidelná výzva AreYouCoordinator

- příjem koordinátorem – sjednocení skupin pod vyššího koordinátora
- pokud člen skupiny nějakou dobu neobdrží AYC svého koordinátora
  1. prohlásí se za koordinátora nové vlastní skupiny
  2. rozešle AYC ostatním

Konzistence je relativní vzhledem ke skupině. Procesy se shodují na členství ve skupině a na nějaké hodnotě. Separované uzly jsou konzistentní samy se sebou.

Kruhový algoritmus

1. Proces se rozhodne volit, pošle zprávu následníkovi.
   - zpráva obsahuje čísla procesů (odesílatel a nejvyšší živý)
   - po návratu obsahuje zpráva nového koordinátora
2. následuje fáze oznámení

Stačí znát následníky a mít možnost zjistit následníka nedostupného uzlu. Složitost je $O(n^2)$

členství ve skupinách

- Procesy si udržují informaci, kdo je v daný okamžik členem skupiny
- viz TRANSIS a ISIS protokoly

24.7 Distribuovaný konsensus

Byzanstký konsensus (1->1) iniciátor zvolí hodnotu rozešle ji všem; všechny loajální uzly se musí shodnout na stejné hodnotě, je-li iniciátor loajální, musí se shoudnout na jeho

Konsensus (n->1) každý uzel má iniciální hodnotu, všechny loajální uzly se musí shodnout na společné hodnotě, pokud je iniciální hodnota všech loajálních uzlů stejná, musí se shodnout na té

Interaktivní konzistence (n->n) každý uzel má inicální hotnotu, všechny loajální uzly se musí shodnout na společném vektoru, hodnota položek vektoru odpovídající loajálním uzlům se musí shodovat s jejich init hodnotou

Problém dvou armád

- Početnější armáda je rozdělena, uspěje jen při synchronizovaném útoku.
- Obě části musí mít jistotu, že druhá část začne útok také.
- Komunikace pouze nespolehlivým kurýrem.

Řešení neexistuje – pošlu-li "Útok v pět", nevím jestli to dostala druhá strana, když mi ona pošle ACK, tak neví jestli jsem ho dostal.

Problém Byzanstkých generálů

- Někteří generálové jsou zrádci.
- Všichni loajální generálové musí rozhodnout shodně.
- Každý generál se rozhoduje na základě informací od ostatních generálů (mají spolehlivou komunikaci).

BÚNO: 1 generál, ostatní důstojníci. Generál vydá rozkaz, důstojníci předají dál dolů – rozkaz bude vydán na základě většiny. Cíle:

1. všichni loajální důstojníci vydají stejný rozkaz
2. nebo, je-li generál loajální, každý loajální důstojník vydá rozkaz generála.

Pro tři uzly s jedním zrádcem nejde.
Pro 4 uzly:

- zrádce generál: alespoň 2 stejné rozkazy: důstojníci si vzájemně přepošlou většinový rozkaz (C1), pro tři různé se shodnou že je generál zrádce (C2)
- zrádce důstojník: v nejhorším případě pošle všem ostatním falešný rozkaz, loajální ale dostanout většinu správných (C2)

Existuje řešení pro 4 uzly s jedním zrádcem, obecně pro m zrádců existuje řešení pro n3m+1 uzlů

Konsensus s nezfalšovatelnými zprávami

Pokud nelze předávanou zprávu zfalšovat, pak libovolný počet zrádců neznemožní konsensus.
Idea algoritmu:

- každý přepošle vše, co dostal, v nezměněné podobě
- každý uzel nakonec sám uvidí co kdo komu poslal
- loajální uzly se shodnou buď na majoritní nebo default hodnotě

24.8 Distribuované sdílení paměti

- Konzistenční modely
- Distribuované stránkování
- Distribuované sdílené proměnné a objekty

Různé mechanismy, v HW/SW, od SMP s busem/switchem přes NUMA, distribuované stránkování po distribuované sdílené promenné a objekty.

Konzistenční modely

Modely bez synchronizační proměnné

implementace možná na úrovní virtuální paměti, procesy o tom nemusí vůbec vědět

striktní konzistence jakékoli čtení z adresy x vrátí hodnotu uloženou při posledním zápisu do x – absolutní časové uspořádání, všechny zápisy okamžitě všude viditelné; musí existovat přesný globální čas

sekvenční kozistence výsledek výpočtu je stejný, jako kdyby všechny operace všech CPU byly vykonávány v nějakém sekvenčním uspořádání, a operace každého CPU jsou v pořadí specifikovaném programem – snadno implementovatelné, povoleno libovolné prokládání instrukcí na různých CPU, všechny procesy vidí stejné pořadí změn; platí že čas čtení a zápisu dohromady musí trvat alespoň tak jako přenos jednoho paketu -> pomalé

kauzální konzistence kauzálně vázané zápisy musí být viděny všemi procesy ve stejném pořadí; vyžaduje udržování grafu závislostí zápisu na čtení

PRAM (pipelined RAM) konzistence zápisy prováděné jedním procesem jsou viděny ostatními procesy v tom pořadí, ve kterém byly prováděny; neexistuje jednotný pohled na rozvrh, snadná na implementaci

slow memory zápisy jedním procesem do jednoho místa musí být viděny ve stejném pořadí; lokální zápis, pomalá nesynchronizovaná propagace, neposkytuje žádnou synchronizaci

Všechny modely vyžadují propagaci všech zápisů všem procesům, přitom ne všechny aplikace potřebují vidět všechny zápisy a jejich pořadí.

Modely se synchronizační proměnnou

Operace Synchronize, Acquire a Release určují kdy je proces v kritické sekci. (Po jejím skončení se data propagují ostatním). Procesy musí o SP vědět, ale výkonnost je vyšší. Rozlišení Acq() (vstup do kritické sekce) a Rel() (výstup z kritické sekce)

slabá konzistence

1. přístup k SP je sekvenčně konzistentní (všechny procesy ho vidí ve stejném pořadí)
2. před přístupem k SP musí být dokončeny všechny předchozí zápisy
3. před přistupem k obyčejným proměnným musí být dokončeny všechny předchozí přístupy k SP

Sáhnutím na SP před čtením se zajistí aktuální verze dat.

Výstupní konzistence

1. Před přístupem ke datům musí být úspěšně dokončeny všechny předchozí Acq() procesu.
2. Před provedením Rel() musí být dokončeny všechny předchozí zápisy a čtení prováděné procesem.
3. Acq() a Rel() musí být PRAM konzistentní.

- eager release consistency – změny se všem propagují po Rel(); optimalizace přístupové doby
- lazy release consistency – změny se propagují až po Acq() jiného procesu; menší nároky na síť

Vstupní konzistence

1. Před Acq() k SP se aktualizují chráněná sdílená data procesu
2. Exkluzivní přístup k SP je povolen jen když k ní nepřistupuje jiný proces (ani neexkluzivně)
3. Pro exkluzivní přístup si musí proces vyžádat aktuální kopii dat od posledního vlastníka (kdo to měl exkluzivně)

Distribuované stránkování

obdoba virtuální paměti (problémy: replikace, nalezení stránky, správa kopií, uvolňování stránek, falešné sdílení — v jedné sdílené stránce jsou dvě proměnné, co spolu nesouvisí)

- sekvenčně konzistentní – stránky mají vlastníka co na ně může psát, ostatní mají kopie pro čtení
- kauzálně konzistentní – vektorové hodiny u stránek i procesů, velká prostorová režie
  - Udržuje se defakto graf
  - U stránek se udržuje vektor udávající na kterých stránkách závisí obrah
  - U procesů vektor, ze kterých stránek znám data
  - Při čtení si proces aktualizuje svůj vektor, pokud je nižší než stránky, případně se invalidují staré stránky
  - Při zápisu se aktualizuje vektor stránky, jako inc(vektor_procesu).

Distribuované sdílené proměnné

- implementováno v knihovnách (např. Munin – sdílení read-only; migratory s eager release konzistencí; write-shared – do lokální kopie se dá psát, po release se propagují změny, případný merge, při konfliku runtime error; normální sdílená data se sekvenční konzistencí).
- odpadá problém s falešným sdílením.
- nutnost rekompilace pro různé jazyky
- distribuované objekty – flexibilnější díky zapouzdření (CORBA, RMI, etc)

24.9 Souborové a adresářové služby

see #Identifikace objektů a přístup k nim

Distribuované souborové systémy

- distribuovaný FS vs. jednotný přístup k síťovým FS
- monolit vs. oddělené souborové a adresářové služby (které mapují uživatelská jména na systémová)
- stavové vs. bezestavové servery
- replikace, cache

sémantika přístupu k souborům

- centralizovaná (každá změna hned vidět)
- relační (změny jsou vidět až po zavření – AFS)
- imutabilní soubory
- transakce

NFS

wen: Network File System (protocol)

- postaveno nad RPC
- vyvinul v 80tých letech Sun
- XDR — eXternal Data Representation – popis datových struktur které NFS používá
- Klient si pošle mount na server, ten mu pošle file handle na mounted directory.
- verze 2, původní, první vypuštěná ven ze Sunu
- verze 3
  - bezstavová, nemá open a close, jen operace READ, WRITE, LOOKUP, REMOVE, MKDIR, RMDIR. Z toho ale plynou problémy, např s ověřováním práv (normálně se to řeší v open). To jde řešit třeba sdílením UID a GID a ověřováním na klientovi.

  - některé podpůrné protokoly běží na různých portech
  - jedna akce nad NFS se typicky skládá z většího množství RPC callů
  - zavádí NLM protokol na managování zámků.
- verze 4
  - stavová (!)
  - složené operace – umožňují omezit počet nutných RPC callů
    - odhaduje se až pětinásobná úspora potřebných client-server interakcí
  - Podpora replikací, bezpečnosti...

AFS

wen: Andrew File System

- vznikl v rámci projektů Andrew Project na Carnegie Mellon University (jméno podle Andrew Carnegie a Andrew Mellon)
- Soubory jsou organizovány pod jednotný globální namespace, rozdělený na cels (administrativní jednotky uzlů) (dejte si ls na /afs na unixu na MS)
- Jednotlivé servery udržují podstromy ve svazcích, servery se na se navzájem replikují.
- poskytuje lepší možnosti scalability a security
- pro security využíván Kerberos, implementována ACL adresářů
- soubory jsou cacheované na klientovi => možnost omezeně fungovat i po pádu serveru/sítě
  - změny jdou do cache a jsou na server propagovány až při zavření souboru
  - pokud je soubor na serveru změněn, klienti kteří ho mají v cache jsou informování
- svazek – strom souborů, adresářů a mountpointů. Sestavuje jej admin.
  - uživatel s ním může pracovat jakoby byl lokální
  - může mít nastaveny kvóty
  - admin ho může přesunout na úplně jiný server bez toho aby se to uživatel dozvěděl
  - může mít několik read-only kopií, AFS zajistí že budou obsahovat správná data, pokud mám jednu připojenou a server kde je spadne => nic se neděje, začnu seamlessly pracovat s jinou
- hodně se jím inspirovalo NFS verze 4, z AFS 2 vychází CODA

CODA

wen:Coda (file system), základy přímo na CMU

- začal vznikat na Carnegie Mellone University v 1987
- vychází přímo z AFS 2
- client-side caching souborů, adresářů a atributů
- čte se z jednoho serveru, píše se na všechny, případně se řeší kolize
- write-back cache
- kerberos-like autentizace
- ACLka
- reintegrace dat na čas odpojených klientů
  - Jde pracovat v connected a diconected mode, tj stahnout si soubor a pak na něm offline dělat.
  - Ve strongly connected modu jsou změny změny zapisovány synchronně
  - Ve weakly connected modu jsou zapsány dodatečně a ručně se zamergují konflikty
- možnost replikace serverů (read/write)

Replikace

viz http://www.kiv.zcu.cz/~ledvina/Prednasky-DS-2007/DS-07-Replikace.pdf

Udržování kopií na více fileserverech. Důvody: spolehlivost, dostupnost, výkon.

- explicitní (uživatel se stará sám)
- odložená (aktualizuje se primární replika, sekundární pak)
- skupinová komunikace (zápisy se simultánně posílají všem replikám)

Aktualizace kopií:

- primární kopie vítězí
- většinové hlasování
- vážené hlasování (různý důraz na čtoucí a zapisující procesy)
- hlasování s duchy (bezdatový server – ghost, obsahuje pouze verze, účastní se hlasování o zápisu ale neučastní se hlasování o čtení)
- dynamická kvóra

klientocentrické konzistenční modely

see also `http://www.cc.gatech.edu/classes/AY2005/cs4210_spring/Lectures/22-Concurrency-2.ppt`

Replikovaná databáze (www+cache, zápisy málo časté), když se klient přesune jinam, musí vidět stejná data.

Implementace: klient si udržuje read-set a write-set (množiny čtení a zápisů co už viděl), posílá s požadavky, podle toho se vynucuje aktualizace replik, jde i vektorovými hodinami (podle replik?). Protože tohle má neomezenou paměťovou náročnost, lepší implementace se sdružováním do sessions vázaných na aplikaci/výpočet/modul a následné mazání z write a read sets. Viz hnusinka zelená :-)

eventuální konzistence po ukončení všech zápisů budou všechny repliky v konečném čase aktualizovány; problém je, že jeden proces může koukat na data jiných replik a vidět něco jiného

monotonní čtení po přečtení hodnoty x všechna další čtení vrátí stejnou nebo novější hodnotu (při připojení k jiné replice uživatel vidí všechny zprávy co už si přečetl dřív – dá se řešit třeba logem updatů, co už klient viděl – replika si pak si může ověřit, že je dost aktuání případně si sehat aktalizaci a poskytnout správná data.

monotonní zápis zápis do proměnné je proveden před každým následním zápisem do ní; než do repliky zapíšu, musí si aplikace přijmout aktuální změny od ostatních. Implementace: podle klientského write-setu si replika ověří, jestli něco nemá dozapsat, po zápisu si klient aktializuje write set.

read your writes procesy při následném čtení vidí svoje zápisy (po aktualizaci wiki nekoukám na kopie z cache). Implementace: buď forward čtení na aktuální repliku, nebo replika ověřuje podle write-setu svoji aktuálnost.

writes follow reads zápis se provede do kopie proměnné, která je alespoň tak aktuální jako ta, která se předtím přečetla. Implementace: aktualizace podle read setu. Po zápisu se aktualizuje read-set i write set klienta.

epidemické protokoly

Eventuální konzistence, optimalizuje pro hodně velké systémy, neřeší konflikty.

- servery jsou: infected (rozšiřují "epidemii"), removed (data mají ale nerozšiřují), susceptible (data nemají)
- antientropie: každý server jednou za čas zkontaktuje náhodný jiný, vymění si co ještě nemají (nebo jen push nebo pull)
- gossiping: pokud byl P aktualizování, kontaktuje nějaký další server Q, aby šířil update; jestliže Q už update má, P se s pravděpodobností 1/k nastaví jako removed
  - nezaručuje že update budou mít všechny servery

24.10 Distribuovaná správa prostorů jmen

Identifikace objektů a přístup k nim

Problémy k řešení

- Který objekt má být použit
- Kde je umístěn
- Jak se k němu jde dostat
- identifiace a přístup k objektům (který, kde je, jak se k němu dostat)
- struktura jmen, trvanlivost, distribuovaná správa jmen, řešení systémových jmen
- kapability a jejich ochrana
- přístup k objektům, distribuovaná správa prostředků, prostředky mohou být replikované a přístup tím komplikovaný
- objekty: aktivní (kód) / pasivní (přes správce, nebo prostředky jako třeba paměť)

Pojmenování, indetifikace, struktura jmen

- prostory jmen mohou být separátní (file system, registry, URL) nebo může být jednotný (distribuovaný name server).
- nestruktorovaná jména (UUID) versus strukturovaná (ms.mff.cuni.cz)
- jména: uživatelská (human-readable) / systémová (interní čílesné kódy)
  - mapování jmen na replikované objekty
  - jména mohou být plochá, strukturovaná (hierarchická), nebo popisná (více atributů)

Kapabilita je datová struktura umožňující jednoznačnou identifikaci objektu, obsahuje i přístupová práva pro držitele – k jednomu objektu typicky patří víc různých kapabilit. Uživatelským procesům je znemožněno vlastní generování kapabilit i změny práv. Kapability se publikují na nameserveru, zároveň si server registruje u Reg serveru. Klient si najde kapability na NS, otevře si ji u Reg serveru, ten ověří a předá serveru, ke kterému si pak klient otevře kanál.

- snadný test oprávněnosti, každý správce si může nadefinovat vlastní druhy práv
- problematické kontrola propagace, potřeba definovat oprávněné uživatele, nebo revokovat
- kapability mohou být buď podepsané, nebo je jejich část s přístupovými právy zašifrována
- Uživatel, který kapabilitu má ji nemůže měnit nebo replikovat.
- Možné jiné řešení pomocí Access Control Listu — prostředek a k němu seznam oprávněných uživatelů. V distribuovaném prostředí nepříliš vhodné.

Adresáře jsou množina položek (jméno,hodnota), hodnota může být:

- primitivní (čísl, řetězce, binární data)
- perzistentní reference (trvalé odkazy na objekty, kapability)
- tranzientní reference (na živé objekty, porty, kanály)
- odkazy na jiné adresáře
- operace jako SET, LOOKUP(jmeno) — změna kontextu, LOOKUP(složené jméno)
- Vyhledávání typicky přes server, ten publikuje jména a vrací reference na objekty (to může být třeba nějaká kapabilita nebo něco jako handle do FS)

Služby

LDAP

see also wen:Lightweight Directory Access Protocol

"Odlehčená" verze X.500 DAP pro použití v TCP/IP sítích.

- adresář je strom položek, každý má sadu atributů
- atribut má jméno a jednu nebo více hodnot (podle definovaného schématu)
- každý položka má DN (distringuished name) – skládá se z RDN (relative DN, vyrobeného z nějaké položky) a DN rodiče
  - DN se může v průběhu živata měnit, někdy se jim přidává i UUID

LDAP poskytuje autentizaci přístupu, služby čtení a vyhledávání v položkách, ověření jestli má položka nějakou hodnotu atributu, aktualizace dat a tak.

JNDI

see also wen:Java Naming and Directory Interface

Jde o nástroj jak unifikovaně z JAVY přistupovat k adresářově organizovaným datům.
Hledání objektů pro Java RMI a Java EE, poskytuje:

- bind objektu ke jménu
- hledání v adresáři (directory lookup iface pro obecné dotazy)
- event interface, dovolující klientům zjistit když se položky změnily
- Service Provider Interface (SPI) pro napojení libovolných adresářových služeb (LDAP, CORBA naming service,...)
- hledá se v kontextu, root je initial context

CORBA Naming/Trading

Naming service:

- naming context: sada vazeb jméno objekt (cosi jako adresář)
- resolve: nalezení objektu podle jména v kontextu
- bind: vytvoření vazby v kontextu (kontext je něco, jako nadřazený adresář, tj. na začátku se dostanu do root kontextu a pak až něco můžu)
- v kontextu může být pod jménem i jiný kontext – složené cesty (jako ve stromovém fs)

viz http://www.iona.com/support/docs/orbix/gen3/33/html/orbixnames33_pguide/Introduction.html
Trading Object Service:

- podobný jako naming service, připomíná zlaté stránky telefonního seznamu
- nabízí služby spolu s referencí (IOR) a popisem, organizované do kategorií (service offer types)
- kategorie jsou definovány pomocí rozhraní `ServiceTypeRepository`
- aplikace exportují reference pomocí rozhraní `Register`, operace `Export`, objekt dá traderovi kapabilitu (popis služby, a interface kde je)
- rozhraní `Lookup` definuje operaci `query`, která umožňuje vyhledat službu podle nějaké podmínky. Někdo se zeptá tradera na službu s danými vlastnostmi, trader dodá umístění
- podobně jako naming service je možné trading services propojovat (tradery se dají navzájem linkovat)

viz http://www.ciaranmchale.com/corba-explained-simply/trading-service.html

24.11 Procesy v distribuovaném prostředí

- sdílení výpočetní síly systému
- vzájemná synchronizace
- vzdálené spouštění, alokace procesorů, migrace, load balancing

Vzdálené spouštění by mělo být transparentní, a vytvořit prostředí odpovídající domácímu

1. registr volných počítačů, tam se nějaký najde
2. vytvoření prostředí pro proces, ten se pustí, po ukončení zpráva jeho domovskému systému

Pokud hostitel přestane být volný, tak se proces zabije, nechá doběhnout, dostane čas na uložení stavu, nebo přemigruje.

Alokace procesorů:

- up-down alogoritmus (koordinátor má tabulku procesorů, ty mu hlásí co dělají; dostávají trestné body za proces jinde, odebírají se jim za neuspokojené požadavky, jinak jdou směrem k nule; při uvolnění procesoru ho dostane proces z fronty neuspokojených požadavků, jehož vysílající procesor má nejméně trestných bodů)
- deterministický grafový – minimalizuje komunikaci, nutno vědět jak co bude komunikovat. Optimální deterministický algoritmus – tok v sítích
- hierarchický – manažeři skupin, při neúspěchu žádost nahoru
- distribuovaný heuristický – několik náhodných výběrů cíle
- bidding – procesy kupují výpočetní sílu

Migrace procesů

- vyvažování zátěže, shutdown, optimalizace
- korektnost – ostatní procesy nejsou migrací ovlivněny, přenesený proces potom je ve stejném stavu
- transparentnost – proces o migraci neví a nemusí spolupracovat

problémy:

- přenesení stavu a adresového prostoru
- komunikace mezi procesy (neztrácet zprávy a tak)
- reziduálni dependence (nechat něco na původním místě)
- vícenásobná migrace

Postup přenosu

1. zmrazení
2. oznámení příjemci, alokace místa tam
3. přenos stavu (registry, zásobník) a kódu / adresového prostoru
4. přesměrování / doručení zpráv
5. dealokace, vyčištění původního místa
6. vazby na nové jádro, nastartování přeneseného procesu (přesunutí částí stavu spolu s procesem, jiné požadavky forwardovat – konzole, některé se používají z nového místa – alokace paměti)
7. dokončení přenosu vazeb

jak kopírovat paměť

- celou při migraci – eliminuje reziduální dependence, ale je pomalé
- pre-copying – proces zmražen jen krátkou dobu, ale ty věci co změní od kopírování se přenášejí víckrát
- copy on reference – stránka se přenese až když je vyžadována, na zdrojové stanici se smaže

zprávy:

- dočasné nebo trvalé přesměrování
- upozornit kamarády předem (ale které?)
- neposílat ACK, on si je zdroj pošle znovu
- migrace kanálu

Vyvažování zátěže

Rozhodnutí o okamžiku migrace – nutno porovnávat zatížení procesorů (nějak konzistentně), pak vybrat co bude migrovat a kam.

- párový algoritmus – vytvoří se páry které se vzájemně vyvažují, zatíženější procesor vybere proces podle míry vylepšení stavu
- vektorový algoritmus (MOSIX) – první vždy vlastní zátěž, pak pošle první půlku nahodnému uzlu, došlá se proloží s vlastní půlkou
- bidding algoritmus: procesy pravidelně vyhodnocovány, pod určitý prah se migruje:
  1. broadcastem žádost o nabídku do vzdálenosti d
  2. adresát proces ohodnotí, případně vrátí nabídku
  3. odpověďi se zkorigují o cenu přenosu, bere se nejlepší; když nedorazí žádná zvýšíme d
- centralizované/hierarchické vyvažovací algoritmy – koordinátor zná zátěže svých procesorů
- lokální (prahová hodnota, když přelezu, ptám se po volných n počítačů, vyberu nejlepší odpověď)

Zablokování

- oblíbené řešení – pštrosí algoritmus
- detekce horší než lokálně: wait-for-graph
- chceme: Každý existující deadlock je v konečném čase detekován, detekovaný deadlock musí existovat

modely deadlocků

Kdy už je deadlock?

- single
- AND model – všechny požadované prostředky musí být přiděleny, aby se proces odblokoval – na deadlock stačí cyklus
- OR model – výpočet může pokračovat, pokud proces dostane alespoň jeden požadovaný prostředek – cyklus je nutná podmínka deadlocku, na postačující je nutný nějaký horší uzel
- k of m
- AND-OR

metody kontrukce wait-for grafu

- centralizovaně (prenos informací po každé změně, v intervalech, nebo na požádání)
  - kauzální doručování proti falešnému uváznutí kvůli zpoždění zpráv
  - hierarchický (každý řeší deadlocky podřízených)
- path-pushing (uzly spravují lokální kusy WFG, sousedním uzlům zasílání externí žádosti, potřeba rozlišovat různé procesy uvnitř jiných uzlů)
- edge-chasing (pošlu zprávu všem, na které čekám, pokud se mi vrátí, jsem v háji; mezitím se to ale mohlo odblokovat – řešením je aging; overkill – zpráva zároveň hledá kandidáta na zahubení)
- diffusing computation – těm na které čekám se posílají pingy, oni je vrací pokud jsou taky zablokováni. pokud dostanu všechny své pingy zpět, mám deadlock
- detekce globalniho stavu – existuje-li deadlock, pak existuje i v konzistentním řezu; při příjmu značku (okamžik řezu) uzel zaznamená lokální WFG, externí závislosti jsou zaslány iniciátorovi

Kapitola 25

Architektura počítačů a sítí

Zdroje:

- zdroj Siti — Peterkovy slidy 3.1 (`http://www.earchiv.cz/l214/index.php3`)
- Peterkovy TCP/IP slidy 2.3(`http://www.earchiv.cz/l215/`)
- architektura pocitacu — Obdrzalkovy slidy (neumi odpovedet na mail), souborkove texty 3.99k, Jirovskeho slidy
- wikipedia (`http://www.wikipedia.org/`)
- Peterkuv archiv obecne `http://www.earchiv.cz/`
- Základ byl převzat z Majklových státnicových výsucků (v 0.5) — viz `http://mff.modry.cz/statnice/`

25.1 Von Neumannova architektura a její alternativy

see also wen:Von Neumann architecture, nebo taky `http://www.earchiv.cz/a94/a406c500.php3`

- Vychází z koncepce vzniklé v USA v letech 1940/1945, jejíž hlavní tvůrce je John von Neumann.
- Vnitřní strukturu počítače tvoří pamět, vstup, výstup a procesor. Struktura počítače se nemění s typem úlohy. Je jednotná pamět pro text programu i data. Rozdíl mezi programem a daty je dán jen jejich interpretací. Pamět je posloupnost stejně velkých pamětových míst. Počítač je řízen tokem instrukcí. Ty se provádějí, kdyz na ně dojde řada. V každém okamžiku probíhá jen jedna činnost.
- **nevýhody** — zastaralá, navrhována jako sekvenční, von Neumann bottleneck (propustnost dat mezi RAM a CPU je prilis nizka (relativne vzhledem k velikosti pameti a rychlosti cpu)
- Na Von Neumanove architekture jsou zalozeny vsechny dnesni mainstreamove pocitace. I kdyz od te doby doslo ke rade uprav (asynchronni I/O, cache, virtualni pamet)
- **alternativy** — počítač řízený tokem dat — operace se provede, jakmile jsou známy všechny její operandy

`http://www.root.cz/clanky/jak-pracuje-pocitac/`

Harvardska architektura

see also wen:Harvard architecture

- od Von Neumannovy se liší tím, že program ukládá do jiné paměti než data
- např DSP procesory a mikrokontrolery
- nehrozí přepsání programu sebou samým, ale je náročnější na výrobu kvůli většímu počtu paměťových sběrnic
- Jiná velikost slova (word) v programové a paměťové částí, t.j. instrukce s konstantou vejde do jediného načtení z programové paměti.
- Načítání instrukce a argumentu (program a data) se nepere o sběrnici (narozdíl od von Neumannovy arch.)

25.2 Multiprocesory

HW Architektura

- Volně spojené (Loosely coupled, grids) (propojené skupiny počítačů)
- Těsně (Tightly coupled) spojené multiprocesory
  - UMA (Uniform memory access, symetrický přístup k paměti)
    - Propojené sběrnicí
    - Propojené n^2 přepínači (crossbar nebo crosspoint switched)
    - Propojené $log_2(n)$ přepínači (multistage switched, omega networks)
    - Propojené jinak, třeba hyperkrychle, klika...
  - NUMA (Non uniform memory access, každý procesor svoji paměť)
    - S-COMA, skupina procesorů má svoji "bližší" paměť, požadavek na obsah paměti náležející jiným procesorům se propaguje vyšší vrstvě
    - ccNUMA, globální adresace, hodně řeší cache, direct memory — speciální paměť instancí a jejich kopií a platností.

Crossbar switched (propojeni pamětí a procesorů)

- Nákladé, kvadratický počet přepínačů
- Možnost zapojit více počítačů

```
M1  M2
|   |
o---o-- P1
|   |
o---o-- P2
```

Multistage switched (omega networks) (propojeni pamětí a procesorů)

- Postupně přepínaná cesta mezi procesorem a pamětí
- Při větším počtu stupňů pomalé
- n log(n) přepínačů

```
P1--                       --M1
    \                     /
     --SWITCH --- SWITCH -
    /        \  /         \
P2--          \/           --M2
P3--          /\           --M3
    \        /  \         /
     --SWITCH --- SWITCH -
    /                     \
P4--                       --M4
```

Rozložení vykonávaného kódu

- OS na jednom, aplikace na druhém (master CPU je botle neck),
- symetricky rozložené OS i aplikace na všech, spousta synchro problémů

Synchro problémy

- Instrukce TSL (test and set lock) musí mít podporu na HW sběrnice (jinak dva CPU mohou nesynchronizovaně volat TSL)
- Cache ping pong při spin locku
- Spin lock je smysluplný, někdo jiný ho může uvolnit (na uniprocesoru se přeplanuje)

Plánování

- Dva rozměry CO a KDE
- time-sharing, v systému jednotná fronta ready procesů, každý procesor si bere co může
- space-sharing — skupina procesů běží na skupině procesorů
- gang-sharing procesy sdruženy do gangů a všichni členové gangu jedou souběžně na různých procesorech

Out-of-order execution

wen:Memory barrier Kratce: procesor muze vykonovat operace v jinem poradi nez jsou v opcode. Na jednoprocesoru to nevidime, ale na multiprocesoru muze dochazet k tomu ze procesory vidi takove, rekneme, sekvencne nekonzistentni mezivysledky. Barierova instrukce prave zajisti synchronizaci.

```
Processor #1:
 loop:
  load the value in location f, if it is 0 goto loop
 print the value in location x

Processor #2:
 store the value 42 into location x
 store the value 1 into location f
```

Pri out-of-order vykonavani instrukci tento kod muze vypsat 0 (BUNO predchozi hodnota prom. x) nebo 42.

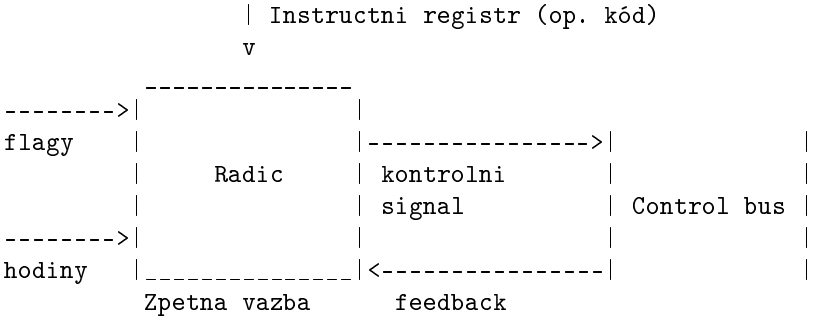
z wikipedie

- – MIMD multiprocessing architecture is suitable for a wide variety of tasks in which completely independent and parallel execution of instructions touching different sets of data can be put to productive use. For this reason, and because it is easy to implement, MIMD predominates in multiprocessing.
  – SIMD multiprocessing is well suited to parallel or vector processing, in which a very large set of data can be divided into parts that are individually subjected to identical but independent operations. A single instruction stream directs the operation of multiple processing units to perform the same manipulations simultaneously on potentially large amounts of data.

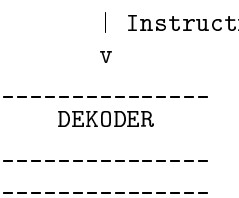
25.3 Mikroprogramové a klasické řadiče, mikroprogramování

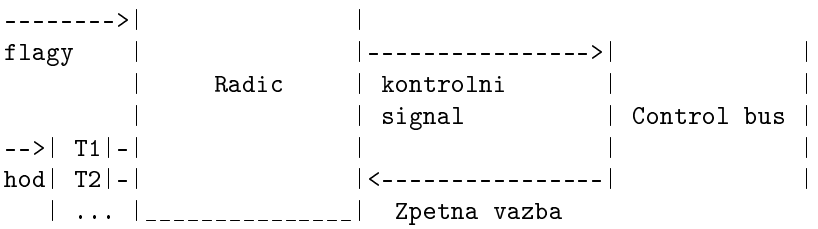
see wen:Microcode

- Klasický řadič má všechny postupy vykonávání instrukcí zadrátované v HW. Pro každou instrukci a stav je naprogramováno, jaké řídící signály mají být dány. Jde vlastně o stavový automat.



- Mikroprogramový řadič má místotoho mikrokód, který popisuje proces vykonávání instrukcí na úplně nejnižší úrovni
- Trochu komplikace s pipeliningem, řídící signály ještě závisí na instrukčním cyklu



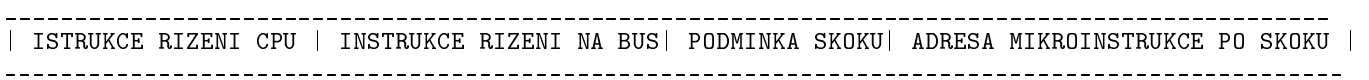


Mikroprogram vypadá nějak takto: Propoj registr 1 se vstupem ALU "A", propoj registr 2 se vstupem ALU "B", nastav ALU na sčítání, vynuluj carry bit v ALU, propoj výstup ALU s registrem 8, aktualizuj registr příznaků podle příznaků v ALU, posuň čítač instrukcí, načti další instrukci, ...

Mikroinstrukce jsou uloženy v oblasti zvané control store, z něhož vybírá instrukce mikroseqeuncer – ten typicky nějak spojuje počítadlo, aktuální hodnotu instrukce kterou CPU zpracovává a pole v mikroinstrukci které říká kam dál.

- horizontální mikrokód – široký, mikroinstrukce obsahuje za sebou pokyny pro všechna propojování, nastavování, skok dál, etc; instrukce jsou široké 56 bitů nebo i víc

Format může vypadat třeba takhle



Provedení instrukce:

- Nastav řídící signály
- Podle podminky se rozhodni
  - Buď proveď další instrukci
  - Nebo skoč
- vertikální mikrokód – mikroinstrukce jsou zakódované, takže jsou kratší, ale vyžadují další zpracování, aby se z nich dostaly ty pokyny
  - někdy to je assembler jednoduššího počítače, který emuluje počítač složitější
  - další možná forma je dvojice (jednotka, pokyn_jednotce) – je potřeba míň paměti, ale CPU je pomalejší
- Nanoprogramování, instrukce mikrokódu se komprimuje tak, že je vlastně jen indexem v adresáři nanoinstrukcí, které něco dělají?

Se klesající cenou tranzistoru začal dominovat horizontální mikrokód.

Wilkesova řídíci jednotka. Založena na řídící matici. Mikroinstukce vybere řádek, podle toho jestli je na dané pozici v řádku jednička se provedou řídící signály odpovídající sloupcům v první části matice a vybere se nasledující instrukce podle jedniček a nul ve sloupcích v příslušném řádku druhé části matice.

25.4 Paměťová hierarchie, vyrovnávací paměti

- obdrzalkovy slidy na principy 07
- registry — cpu, nejrychlejsi — 1ns, nejmensi ~B
- vyrovnavaci pamet — L1, L2 cache, 10ns ~kB
- operacni pamet — RAM 10-100ns ~MB
- sekundarni pamet — HDD 10ms ~GB
- archivni pamet — pasky, DVD, nejpomalejsi nejvetsi 100+ ms, TB
- **vyrovnávací paměti**
  - obvykle pouziti — tam, kde je vyrazny rozdil v rychlostech — procesor vs ram, pristupy na disk.
  - Vyuziva se casove lokality pristupu — data jednou pouzita budou pravdepodobne pouzita podruhe a pametove lokality — kdyz pouziju nejaka data, tak asi budu pouzivat i data ulozena v nejblizsim okoli (pamet pracuje po strankach, takze typicky nactu i okoli) data v cache ulozena spolu se svou adresou. k vyhledavani je mozno pouzit nekolik druhu mapovani (plne asociativni, prime, ...)

- cache ma typicky mnohem mensi kapacitu nez ta pamet, ktera je cacheovana (cpu ma radove 0,5MB cache vs RAM 0,5GB; HDD ma 8MB cache a kapacitu 200GB).
- RAM je v podstate taky cache sekundarni pameti (disku).
- Pro uvolnovani bloku z cache mozno pouzit ruzne algoritmy (LRU, LFU, FIFO, ...)
- typy cache:
  * **write-thru** — používat průpis (co se zapíše do vyrovnávací paměti, zapíše se okamžitě i do hlavní)
  * **write-back** — uklízet modifikované bloky do hlavní paměti (vyšší výkon). Při zápisu do vyrovnávací paměti se pouze nastaví (dirty) příznak a do hlavní paměti se data zapíší, až když jsou vyhozena z cache (a pouze pokud mají nastaveno dirty).
- Vyhledávání v cache:
  * **plně asociativní mapování** — při hledání hledána přímo adresa v asociativním adresáři
  * **přímé mapování** — každý blok v cache své místo
  * **skupinové asociativní mapování** — kombinace předchozích dvou (2-way atd.)
- U multiprocesoru je potřeba zaručit konzistenci dat mezi procesorovými cache.
- Efektivni pristupova doba — "amortizovany" cas potrebny k pristupu do pameti — pomoci cetnosti vypadku stranek a prumerneho casu pristupu do cache a cacheovne pameti se spocita, jaky byl prumerny pristupovy cas ke strance.

25.5 Stránkování a segmentace

Virtualni pamet — zajišťuje větší adresový prostor, než je k dispozici fyzicke pameti. Existují dvě základní metody:

Stránkování

Prime strankovani

Procesy mají k dispozici velký virtuální adresový prostor. Operační paměť tvoří fyzický adresový prostor. Virtuální i fyzická pamět je rozdělena na bloky stejné velikosti — virtuální na stránky, fyzická na rámce. Proces se odvolává pouze na adresy ve svém virtuálním prostoru. Operační systém plní stránkovací tabulky informacemi o tom, které stránky jsou v kterých rámcích — mapování. Při pokusu procesu o přístup k paměti jednotka řízení paměti(MMU — memory management unit) dle stránkovacích tabulek zjistí, zda se stránka obsahující požadované místo nalézá v operační paměti. Pokud ano, provede překlad adresy na fyzickou a instrukci předá procesoru. Pokud ne, vygeneruje přerušení — nastal výpadek stránky(page fault).

OS nalezne volný rámec (případně nějaký uvolní tím, že jeho obsah zapíše do odkládací paměti — viz strategie vymeny stranek) a z odkládací paměti natáhne požadovanou stránku do rámce. Procesor nyní zopakuje instrukci, která přerušení vyvolala. Celý postup je pro proces naprosto transparentní. Pro 32bitovy prostor a 4kB stranky maji strankovaci tabulky uz velikost 4MB, takze se pouziva vice urovnove strankovani (strategie jako indexovani souboru)

32bitova virtualni "pametova adresa" pouzivana procesem je rozdelena na adresu stranky a adresu ve strance. U 4kB stranek se prvnich 20bitu pouzije pro mechanismus hledani fyzicke stranky a az je nalezena, tak poslednich 12 bitu urcuje poradi v ramci stranky, kde je to, co hledame. zvysovani vykonu: mmu primo soucasti procesoru, TLB — transaction lookaside buffer — naposledy pouzita mapovani jsou v malem bufferu, kde se vyhledaji rychleji.

Strategie vymeny stranek:

- OPT — vyhodit stranku, ktera bude potreba za nejdelsi casovy usek — neimplementovatelna teoreticka strategie.
- FIFO — klasicka fronta, trpi Beladyho anomálií (012301401234 pro 3/4 sloty – přidání paměti vede k více výpadkům).
- LRU — Least Recently Used — nejlepsi aproximace OPT
  - casovy "citac" — obsahuje timestamp posledniho pristupu — moc nefunguje při procházení velkých struktur.
  - zasobnik implementovany jako dvojity spojovy seznam. Pokud pouziju stranku tak se presune na vrchol. Nevic pouzivana stranka je na vrcholu, nejmin pouzivana je na dne. Vse je v O(1).
- NRU — Not Recently Used – stránky mají čítač přístupu a špinavosti, čítač přístupu se občas maže, pak se první vyhazují stránky nečtené a čisté, pak nečtené a špinavé atd.
- One Hand Clock – kruhový seznam, při nutnosti vyhodit stránku se pustí ručička, která nuluje access bit pokud je nastavený a vyhazuje stránku pokud je nulový.
- Two Hand Clock – první ručička maže accessed bity, druhá (v pevném odstupu za ní) vyhazuje stránky s nulovým bitem, odstup ručiček určuje agresivitu.

- LFU — Least Frequently Used — stranky maji citac pristupu. Hlavni nedostatek je ten ze stranka hodne pouzita minule a uz nikdy nepouzita v budoucnu ma malo sanci na vyhozeni.

NRU, one hand clock a two hand clock algoritmy muzou byt chapane jako aproximace LRU, ktery je aproximaci OPT. FIFO a LFU jsou na tom nejhure.

Inverzni strankovani

inverzni strankovani (inverted page tables) — prevazne u 64bit. architektur napr. UltraSPARC 64 a Power PC — kvuli velikosti virtualniho prostoru organizuje se adresovani pres ramce fyzicke pameti, ne pres stranky.

Problemy:

- Pristupuje se na virtualni adresu
  - naivni reseni — linearni prohledavani polozek (protoze neumime rychle urcit ktery fyzicky ramec odpovida virtualni adrese)
  - efektivni reseni — pouzije se hashovani — nejdrive sahneme na hash table a tak zjistime kde hledat polozku v page table.
- Sdileni pameti. Uz to nejde tak jako u primych page table — fyzicky ramec odkazuje na virtualni stranku jako 1-1. Pouzije se stejna virtualni adresa stranky ve vsech ASID (???). Anglicky receno: " ... allow the page table to contain only one mapping of a virtual address to the shared physical address.".

Working set – stránky, které proces právě používá. Když běží moc aplikací, tak se jejich working sets nevejdou do paměti, každé přepnutí procesu vede k mohutnému swapování a vše je v háji. Tomu se říká thrashing.

Segmentace

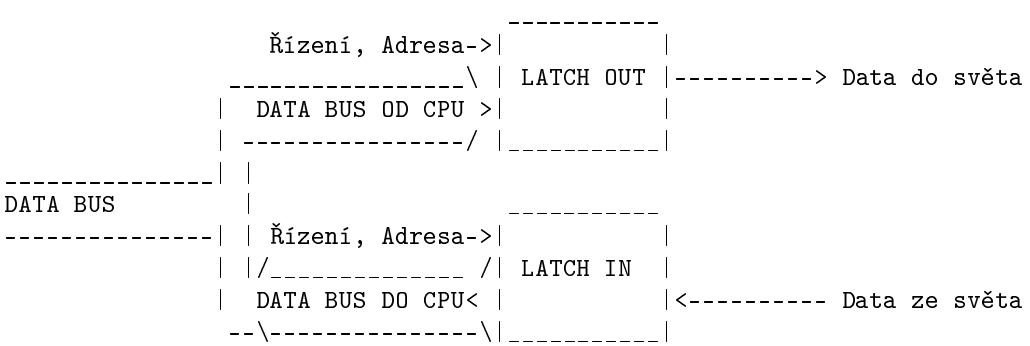
Paměť je rozdělena na segmenty. Program přistupuje na adresy rozdělené na segment/offset. Segmenty mohou být různě velké, a kromě umístení ve fyzické paměti mají i příznaky co na ně může sahat. Segmenty jdou přesouvat i zvětšovat aniž by o tom aplikace musela nezbytně vědět (stačí je jinak přeházet v paměti, číslo segmentu i offset v kódu se nemění).

Stránkování + segmentace: Kazdy proces ma iluzi pro sebe vyhrazeneho prostoru — segmentu. Pro hledani fyzicke stranky se pak pouziva i cislo segmentu prirazene procesu. MMU nejdrive v tabulce segmentu najde, kde zacina tabulka stranek pro dany segment a pak se pokracuje jako vyse.

25.6 Vstupně výstupní subsystémy, přerušení, DMA

- Art of Assembly — Chapter Seven The I/O Subsystem
- Všechno je připojeno vhodně na sběrnici (pomocí "latch" nebo flip-flop).
- Např. output port se může jevit jako místo v paměti, akorát má ještě spojení na vnější svět
- Jakmile je do latch zapsána datová posloupnost, zpřístupní je na sadě drátů co jdou z počítače.

Obecně jde Input i Output zařízení sdílející jednu adresu nakreslit následovně, na BUS je někde napojen CPU. Data nemusí jít úplně nutně do CPU, mohou jít třeba jinám (v režiji DMA)



Přerušení

see also Operační systémy (státnice)#Mechanismus přerušení v OS

Přerušení je událost, která mění sekvenci, ve které CPU vykonává instrukce; je generováno HW nebo SW. Je pouzivano pro praci s I/O, virtualni pameti, DMA, ...
Přerušení lze generovat:

- synchronně (trap) | vyvolala ho instrukce v běžícím programu, např. výpadek stránky
- asynchronně (interrupt) | vnější událost, např. dokončení V/V operace, porucha HW: : :
- výjimkou (exception) | nesprávné chování programu, např. dělení nulou

Postup při generování přerušení:

- 1) vznik příčiny, vyslání žádosti
- 2) rozhodnutí o přijetí či nepřijetí — přerušení lze maskovat — přerušení se uplatní až později
- 3) identifikace zdroje přerušení a určení adresy obslužného programu — rutina, ktera vyresi duvod preruseni
- 4) úschova aktuálního stavu procesoru na zásobník — je možno povolit víceúrovňové přerušení, ale je nutno zabránit zacyklení procesor obvykle při přijetí přerušení maskuje všechna přerušení
- 5) provedení obslužného programu
- 6) obnova stavu procesoru
- 7) návrat do přerušeného programu | obvykle pomocí zvláštní strojové instrukce, která obnoví to, co CPU uklidil

Programové přerušení se využívá ke zpřístupnění služeb OS, instrukce call se nepoužívá kvůli ochraně | vstupní body jsou při volání přes přerušení pevně dány.

DMA (Direct Memory Access)

see also Operační systémy (státnice)#DMA

- je způsob rychlého přenosu dat bez ucasti procesoru při I/O operacích přesunech dat v paměti, zobrazování, refreshi . . .
- Přenos není řízen strojovými instrukcemi, ale řadičem DMA.
- Přenos probíhá přímo mezi pamětí a I/O zařízením, procesor je odpojen od sběrnic.
- Jak se řadič DMA a procesor dohodnou, kdy provést přenos?
  - dávkový režim — řadič o sběrnice žádá, po dobu přenosu procesor na sběrnici nesahá, pak dostane od řadiče zpávu že je hotovo
  - kradení cyklů — řadič je schopen uspat procesor, provést přenos a pak ho probudit; náročné na hardware, nejde na dlouho
  - transparentní režim — řadič pozná, kdy procesor nepracuje se sběrnicemi, a v této době provede přenos — procesor o tom ani neví

25.7 Způsoby obsluhy periferií

Metody

- polling, periodické dotazování na stav zařízení
- Přerušení, zařízení samo zatahá procesor za nožičku, (většinou verkrze řadič přerušení)
- DMA, to je rozebráno výše samostatně

Abstrakce

- Většinou se programátorovi zařízení jeví jako stream nebo jako soubor.

Typy přístupu

- Adresované (Memmory mapped), jeví se jako nějaká část paměti

  - Přímé, adresa je součást instrukce
  - Nepřímé, součástí instrukce je adresa registru, kde je adresa dat

- Port mapped, k přístupu se používá specielní instrukce (wen:Input/output), `http://webster.cs.ucr.edu/AoA/Windows/HTML/IO.html`
  - Intel 80x86 IN and OUT instrukce

25.8 Vstupně-výstupní topologie

- daisy chain (řetězec počítač – zařízení – další zařízení – poslední zařízení)
  - SCSI se tak fyzicky zapojuje, i když elektricky je to sběrnice
  - MIDI má takovou formu (viz `http://www.root.cz/clanky/rozhrani-midi-na-osobnich-pocitacich/`)
- sběrnice (všechno visí na jednom drátu)
- stromová struktura (USB)

25.9 Sběrnice a jejich řízení (SCSI, USB, AGP, ...)

Dřív prostě dráty, obsahovaly následující linky:

- datové (přenáší data)
- adresové (kam mají data jít)
- řídící (režie přenosu)

Jde o sdílené médium, takže je potřeba nějak zajistit, aby nedocházelo ke kolizím.
přenos:

- synchronní (podle předem daných hodin)
- asynchronní (signál: teď bude zpráva, pak je zpráva)

Ke sběrnici se jednoduše přidávají další (i vcelku různá) zařízení, ale je to potenciální bottleneck (všechno na sběrnici se musí bavit stejnou rychlostí).
Přenos dat:

- s účastí CPU, které to řídí
- bez účasti CPU, řízeno řadičem (třeba DMA), nebo přímo zařízením (bus mastering)

Původně CPU přímo na sběrnici, později oddělena CPU/pamět na rychlejší, a za řadičem pomalejší zbytek. K rychlejším komponentám se pak přidala i grafická karta na vyhrazené sběrnici AGP nebo PCIe.

Sériové sběrnice

Přenáší najednou jen jeden bit (víc jen v nějakém složitějším kódování).
Výhody: jednodušší (nemusí sychronizovat mezi jednotlivými kanály), menší nároky na místo; dá se dosahovat vyšších přenosových rychlostí na kanál. Některé (PCI Express) začínají používat full-duplex p2p spojení, číž eliminují kolize.
Zástupci:

- USB
- FireWire
- SATA
- PCI Express

Paralelní sběrnice

Přenáši se najednou více bitů (asi ve více vodičích, tedy)
Zástupci:

- AGP
- ISA
- PCI
- ATA (aka IDE)
- SCSI

SCSI

- SCSI (Small Computer System Interface) je standardní rozhraní a sada příkazů pro výměnu dat mezi externími nebo interními počítačovými zařízeními a počítačovou sběrnicí. SCSI se vyslovuje „skazi “. Komunikace probíhá mezi **iniciátorem** a **cílem** pomocí příkazů.
- SCSI commands are sent in a Command Descriptor Block (CDB). The CDB consists of a one byte operation code followed by five or more bytes containing command-specific parameters.
- There are 4 categories of SCSI commands: N (non-data), W (writing data from initiator to target), R (reading data), and B (bidirectional). There are about 60 different SCSI commands.

AGP

- AGP (Accelerated Graphics Port) je PCI sběrnice, které běží na 66 MHz, a data přenáší na vzestupné i na sestupné hraně hodin, čimž zdvojuje rychlost přenosu. Obrázek v Ceresovo (`http://dsrg.mff.cuni.cz/~ceres/sch/osy/notes.php`) handoutech.

USB

- Mezi dvěma zařízeními vždz jen jeden kabel, jsou tam HUBy. Logicky je to tedy sběrnice, ale fyzicky tam nic sdíleného není. Jedná se o strom, v jeho kořenu je jedno zařízení připojené na vnitřní sběrnici, pracující v režimu MASTER. ve stromě může být celkem 127 jiných zařízení v režimu SLAVE.
- Řešení napájení z USB
- Zařízení, která se dají připojit na USB spadají do tříd. Tj, OS po připojení a resetu zařízení umí rozpoznat z jaké třídy a podtřídy zařízení je a podle toho zvolit ovladač. Ten je stejný bez ohledu na výrobce.

25.10 Mezipočítačová komunikace

pres site s pouzitim nejake sitove architektury — dnes prevazne TCP/IP.

Stejne vrstvy sitove architektury na ruznych pocitacich spolu “jako” komunikuji (rovnobezna komunikace). Ve skutecnosti kazda vrstva krome fyzicke vyuziva sluzeb te pod ni. Fyzicka vrstva umi posilat 0 a 1 nekam.

Kazda dve zarizeni komunikuji pomoci nejakeho (predem znameho) protokolu — “jazyka” s nejakou gramatikou — napr. IP, TCP, UDP, HTTP, FTP, ...

Komunikujici strany si predavaji PDU — protocol data unit, ktery ma vzdy dve slozky (hlavicku a obsah), i kdyz se muze jmenovat ruzne (packet, ramec, http data request, ...). Hlavicka obsahuje rezijni informace, napr. druh dat, adresa odesilatele a prijemce, poradove cislo, ...

25.11 Sériové a paralelní kanály

Asi viz #Sběrnice a jejich řízení (SCSI, USB, AGP, ...)

RS-232, Centronics, USB

- Seriovy port RS-232C, seriova sběrnice, viz `http://www.root.cz/clanky/komunikace-pomoci-serioveho-portu-rs-23`
  - Nemaj vyvedený samostatný hodinový signál pro synchronizaci
  - Posílá se start a stop bit, mezi ně posloupnost bitů, velké nároky na přesnost krystalky na obou stranách
  - Nutné jen dva datové vodiče, (tři s “nulákem”), volitelně a řídícími vodiči pro HW řízení toku, handshaking.
  - Simplexní, poloduplexní či plně duplexní asynchronní přenos dat

- Paralelní port, (Centronics rozhraní, LPT Port), viz `http://www.root.cz/clanky/paralelni-port-a-rozhrani-centro`
  - Konektor s dvaceti pěti piny, dvě řady (13+12), označení DB25
  - Osm datových vodičů
  - Čtyři řídicí vodiče
  - Pět stavových vodičů, kterými připojené zařízení posílá počítači zpět informaci o svém stavu
  - Připojen přes kartu např na PCI sběrnici

25.12 Modemy

see also wen:Modem

Modem je zarizeni pouzivane k prevodu digitalniho signalu na analogovy, jeho kodovani pro prenos pres metalicke medium a konverze zpet do digitalni podoby na druhe strane linky.

MOdulator a DEModulator, odtud modem. Modulovani je "nanaseni" signalu nesouciho informaci na nejakou nosnou vlnu, ktera se dobre siri mediem. Samotna nosna (typicky sinusoida) nenese informaci. Informace se uklada az do zmen teto vlny tzv. modulovanim.

- baseband — modulují do změn napětí/proudu. Unipolární (0V-5V), bipolární(-5V-5V), Manchester(náběžná hrana, sestupná hrana), Diferenciální Manchester(byla změna, nebyla změna).
- broadband — Modulace na nosnou vlnu, $y = A \cdot sin(\omega t + \varphi)$, tj tři možnosti co měnit (amplituda, frekvence, posunutí=fáze)

Pouziva se frekvencni, amplitudova a fazova modulace. Presneji kombinace techto tri. Napr. QAM — kvadraturni amplitudova modulace je kombinaci 3 ruznych amplitud a 12 fazovych posunu, coz tvori celkem 36 stavu. Pro snazsi rozpoznatelnost se ale pouziva jen 16 z nich s nejvyssi vzajemnou "vzdalenosti". Prenosova rychlost = modulacni rychlost (kolikrat za sekundu se zmeni signal) * log2 n. Kde n je pocet rozpoznatelnych stavu (16 u QAM)

Baud = počet změn stavu za sec != bps (bit per second)... modemy totiz mohou mit vic stavu.

Shanonuv theorem

$$C = B \log_2 \left(1 + \frac{S}{N}\right)$$

where

C is the channel capacity in bits per second;

B is the bandwidth (signal processing) of the channel in hertz;

S is the total signal power over the bandwidth, measured in watt or volt^2;

N is the total noise power over the bandwidth, measured in watt or volt^2; and

S/N is the signal-to-noise ratio (SNR) or the carrier-to-noise ratio (CNR) of the communication signal to the Gaussian noise interference expressed as a linear power ratio (not as logarithmic decibels]]).

Media

- drátová (kabely a optika)
- bezdrátová (wifi, bluetooth,...)
  - narrowband režim (malý rozsah frekvencí, velký odstup signálu a šumu)
  - spreadspectrum režim (velký rozsah frekvencí)
    - Frekvency hopping (změny frekvence)
    - DSSS (Direct Sequece Spread Spectrum), na každý bit sekvence bitů (chip) a z ní se pak něco vyXORuje (WiFi, GPS)

Multiplexing

- Frekvenční
- Časový
- Kódový na DSSS (chiping kódy)

Realita

- 33,6 kbps, protože šířka pásma elefonní linky je 3,1 KHz, AT příkazy
- ISDN (kde je všecko digitální, modem není modem)
- ADSL, subkanály, na každém je nosná frekvence, QAM

25.13 Topologie sítí

see also wen:Network topology

- point-to-point – permanentní nebo spojované okruhy (klasická telefonní síť)
- sběrnice – drát na kterém jsou všichni (původní Ethernet)
- kruh (Token Ring, FDDI – kruh může být koncentrovaný ve středu hvězdy – see wen:Media Access Unit)
- strom (Ethernet 10BASE-T a ty po něm, 100VG-AnyLAN – alternativní varianta 100Mbit Ethernetu)
- hvězda (jednotlivý segment nového Ethernetu)
- klika (každý s každým – asi nějaké bezdráty)
- mesh (ad-hoc wireless sítě – 802.11s, OLPC laptopy mají něco takového)

25.14 Přístupové metody

Resi problem pristupu ke sdilenemu prenosovemu mediu. resi se na urovni linkove vrstvy (podvrstvy MAC v RM ISO/OSI).

Varianty reseni, nekolik ruznych pristupu:

- deterministicke (rizene) vs. nedeterministicke (nerizene)
- centralizovane vs. distribuovane
- vylucujici kolize / zpracovavajici kolize / bez detekce i napravy

Srovnani:

- nerizene metody funguji lepe v mensich sitich s mensi vytizenosti media (LAN)
- rizene funguji lepe ve vetsich sitich s vetsi vytizenosti (paterni site).

Rizene centralizovane

arbitr vyzyva (polling) nebo je dotazovan (RTS/CTS – Request To Send / Clear To Send).

- Vyhody: arbitr muze menit strategie
- Nevyhody: vyssi rezie, single point of failure
- Priklad: 100VG-AnyLAN (pouze stromova topologie s hlavnim korenem)

Rizene distribuovane

jasna pravidla, vsechny uzly rovnocenne,
varianty:

- metody logickeho kruhu (tokenring) — siti putuje opravneni vysilat (token), ktery si uzly pravidelne predavaji.
- rezervacni metody (siti putuje specialni rezervacni ramec, ktery se pravidelne vyhodnocuje vsemi uzly). Ramec muze mit podobu bitmapy, ktera reprezentuje ktery uzel ma zajem o vysilani.
- Vyhody: distribuovanost — nestoji a nepada s jednim uzlem
- Nevyhody: jednotlive uzly toho musi umet vice
- Priklad: FDDI, TokenRing

Nerizene distribuovane

Metoda vysilej co chces (radiem), kdyz nedojde potvrzeni vysilej znovu. Vzniklo na Hawaiskych ostrovech, kde nebyla zadna pouzitelna infrastruktura.

- Vyhody: jednoduchý protokol
- Nevyhody: velka rezie na kolize a znovu posilani
- Priklad: wen:ALOHAnet

CSMA/CD, CSMA/CA a no-stress metody

"Poslechne" si (Carrier Sense), zda nekdo vysila a pokud ne, tak zacne — malo kolizi (ale muzou byt — Multiple Access) vzhledem k vysoke rychlosti sireni signalu. Pokud nekdo vysila, tak pocka az skonci a pak zkusi s pravdepodobnosti p (v zavislosti na metode, pro Ethernet p=1) vysilat znovu. Pokud dojde ke kolizi (Colision Detection), tak se chvili vysila JAM signal pro utvrzeni kolize. Ruzne metody pak muzou zkusit hned vysilat znovu, nebo chvili mlcet. V Ethernetu se oba (vsechny) kolizni uzly odmlci na nahodne zvolenou dobu a pak zacnou vysilat znovu (pokud dojde opet ke kolizi, tak se zdvojnasobi interval, ze ktereho si voli nahodnou dobu k odmlceni).

Na podobném principu funguje i CSMA/CA, která se snaží předcházet kolizím. Poslouchá, počká, nechá rozestup a pak vysílá.

"No-stres" metoda nic neřeší, pokud nepřijde potvrzení, tak je prostě neco špatně a pošle se to znova.

- vyhody: mohou byt velmi efektivni — mala rezie u nizke zateze
- nevyhody: nezarucuji vysledek (mohou byt kolize do te doby, nez to uzel vzda), pri vyssi zatezi prenosoveho media se velmi zvysuje rezie
- Priklad: Ethernet

Kolizni domena — kam az se siri kolize — repeatery siri, bridge, routery ji ohranicuji
Celkove se ocekava nizke vyuziti prenosoveho media a tim i malo kolizi

Bezdratove pristupove metody

- IEEE 802.11 — "casta" ztrata ramcu kvuli spatnemu signalu, neni mozne zjistit ze nastala kolize behem vysilani (radio je poloduplex). Vzdy ceka na ACK ramec, aby se ujistil ze prenos probehl uspesne. Problem "skryte" stanice (A chcevysilat na B ale nevidi ze na nej vysila C protoze signal z C nedosahuje) a predsunute stanice (B chce vysilat na C ale na nekoho vysila A takze medium je 'obsazeno' i kdyz na C by vysilat mohl — ale nevi ze A na C nevidi a tedy nejde o kolizi).
  - Distributed coordination function (DCF), CSMA/CA (Colision Avoidance). 0-perzistentni metoda — pokud nekdo vysila odmlci se na nahodnou dobu. Stale se muze dojit ke kolizi, napr. problem skryteho uzlu (tzn. neni 100% CA metoda). Povinna metoda dle IEEE 802.11
  - RTS/CTS `http://www.marigold.cz/wifi/doku.php/problem_skryteho_uzlu`. Snazi resit problem skryteho uzlu. Uzel X vysila Request-to-Send signal, uzly co jsou v dosahu zachyti signal RTS, nastavi svuj network allocation vector (stopky) a vyslou Clear-to-Send signal ktery zachyti i ty uzly ktere jsou mimo dosah uzlu X.
  - Point coordination function (PCF). Centralizovana metoda rizeni pristpu access pointem (AP).
- Bluetooth — po navazani spojeni preskakuje rychle mezi vysilacimi frekvencemi a tim si "zarucuje" malou pravdepodobnost kolize (u prenosu hlasu nevadi sem tam nedorucenost)

25.15 Síťové technologie — ATM, FDDI, FastEthernet, beztrátové technologie

ATM

see also wen:Asynchronous Transfer Mode

Prenosova technologie pouzivana v nekterych paternich sitich, převážně telekomunikačních společností. Je orientovana na QoS, je draha, jen spojovana, nespolehlivá, nema broadcast, ale zase nemá omezenou prenosovou rychlost. Puvodem ze sveta spoju, ale respektuje cast. i svet pocitacu. Je to bitová roura, žádné potvrzování a řízení toku.

ATM pracuje s bunkami dat. Maji vzdy 53B, z toho 5 je hlavicka, 48 naklad.

Nabizi vyssim vrstvam nekolik trid sluzeb:

- CBR — constant bit rate — garantuje celou kapacitu — emuluje drat — rezervuje max, vhodne pro tel. ustredny, nekomprimovana multimedia
- VBR — variable br — garantuje co, co prenos prave potrebuje — rezervuje max — nevyuzite pasmo vraci (narozdil od CBR), vhodne pro komprimovana multimedia
- ABR — available br — rezervuje min
- UBR — negarantuje nic — best effort — co zbyde

Nabizi virtualni okruhy — snazsi routovani pro ustredny, presmerovani, pokud cesta vypadne, nevyhodou je nutnost dvojiho protokolu aby to fungovalo (jeden pro komunikaci s koncovymi uzly, jeden pro vnitrni komunikaci ustreden mezi sebou).

Z hlediska vrstev je ATM nad fyzickou vrstvou a nad sebou jeste potrebuje AAL — (ATM Adaptation Layer), jejiz hlavnim ukolem je rozsekavat data na kusy po 44-48B pro prenos pres ATM. AAL ma casti AAL1-4 zhruba odpovidajici tridam sluzeb (viz vyse). AAL5 je specializovan na pocitacove prenosy (velke datagramy, nespolehlive, burstmod).

VPI a VCI — virtual path se přepisuje při přechodu přes ústřednu, VCI ne.

- vyhody: rychla, snazi se byt univerzalni
- nevyhody: tezkopadna, draha, IP na ATM není moc efektivní
- Priklad: paterni sit ADSL Ceskeho Telecomu (O2)

FDDI

see also wen:Fiber distributed data interface

Fiber Distributed Data Interface — data prenasena optikou. Nejstarsi vysokorychlostni (100Mbps) technologie, vhodna pro paterni site. Topologii dvojity ring, druhy pro zalohovani, pokud funguji oba, tak max. prenosova rychlost 200Mbps. V praxi se pouziva dvojity kruh stromu, kdy do kruhu jsou zapojeny treba routery a k nim stromem obyc. pocitace plus dalsi aktivni prvky.

Pouziva token passing pro rizeni pristupu. Moznost provozovat na dlouhe vzdalenosti (100km), pro pripojeni na kratke vznikla z ekonomickych duvodu varianta CDDI (Copper DDI). Existuje i novejsi FDDI2, ale ani ta se moc nepouziva. Celkove jde o do budoucna pravdepodobne mrtvou technologii, používá se spíš Fast Ethernet nebo Gigabit Ethernet.

Dva typy uzlů, co se připojují

- DAS oba okruhy jimy přiímo prochází
- SAS zapojené pomocí konektoru

Optické přemostění, pokud je připojená stanice vypnutá

FastEthernet

dnes verze ethernetu pracujici 100Mbps a vice (Gb Ethernet, 10Gb Ethernet), puvodne jen 100BaseT — prvni 100MBps CSMA/CD standard. Vychazi z 10Mbps, jen vse zrychluje a zkracuje intervaly. Verze, ktera i meni mnohe veci dostala nazev 100VG-AnyLAN.

Ethernet pouziva 48bitove adresy — MAC adresy. Puvodne mely byt nemenne, dnes je vetsinou lze menit. Prvni tri byty jsou OUI (organizationally unique identifier) — identifikator organizace, dalsi tri identifikator zarizeni. Mely by byt unikatni na celem svete (coz zjevne nelze zarucit, kdyz je uz lze uzivatelsky menit).

Ethernet je vrstvou na urovni sitoveho rozhrani. Ethernetove ramce je treba rozlisovat na ramce na urovani MAC (nizsi podvrstva) a LLC (vyssi podvrstva). Tj. Linková vrstva ISO-OSI se rozpadá pro ethernet na LLC (Logical Link Control) a MAC (Media Access Controll) podvrstvu Ramec ma 22B hlavicku a az 42-1500B payload.

Struktura packetu

```
-------------------------------------------------------
|PREAMBULE| DEST | SOURCE | TYP | PAYLOAD | CHECKSUM |
-------------------------------------------------------
```

100base-tx kodovani — misto Manchesteru (Ethernet) se zaclo pouzivat kodovani 4b/5b (4 informace, 5 preneseno, tj vždy se posílá pětice s alespoň dvěma jedničkama). Zpetna "kompatibilita" — vsechny sitove prvky ktere umi 100Mb se dokazou prepnout i na 10Mb. Puvodne poloduplex, ale rozsiren na plne duplexni (vysilani muze probihat obema smery) za predpokladu zadnych opakovacu a jen switchu krome koncovych stanic — uz neni pouzivano sdilene medium — kazda stanice ma primy spoj na switch a tim odpada nutnost CSMA/CD.

Rozpoznání 10 vs 100 MB ethetnetu na základě FastLinkPulse, 17-33 pásků o 100 ns (který desetimegabit detekuje jako jeden pulz?)

Mozna delka kabelu uz neni omezena technologii na 200m, ale jen utlumem kabelu.

Gigabit ethernet

Optikou nebo kroucenou dvoulinkou (kat. 5, 4 pary). Varianty poloduplex (s CSMA/CD) i fullduplex (ten se stava technologii vhodnou i jinde nez v LAN). Jiz jen dvoubodove spoje (pres switche se realizuji ruzne topologie).

10Gb ethernet — predpoklada se jen optika a jen fullduplex.

bezdrátové technologie

Je k dispozici omezeny rozsah pouzitelnych frekvenci.

- licencovane (nabizi operatori) provozovatel ma exkluzivni prava na vysilani, zadne ruseni, vydava CTU — 900MHz, 1.8GHz GSM, UMTS, CDMA
- nelicencovane, musi jen respektovat omezeni o max. vykonu; jednotlivi provozovatele si mohou rusit signal — wifi (2,4; 5GHz), WiMAX, bluetooth

druhy deleni bezdratovych siti

- podle druhu mobility (wifi, wimax vs GSM, CDMA vs Iridium; satelitni telefon — primo "bts")
- podle dosahu — cordless, wireless, satelitni
- podle druhu prenosu (1:1, 1:n)

WLAN standardy IEEE 802.11

- 802.11a – 5GHz, 54Mbit/s
- 802.11b – WiFi, 2.4GHz, 11Mbit/s
- 802.11g – 2.4GHz, 54Mbit/s

Rámce v 802.11

- řídící (např ACK)
- administrační (např authentizační)
- datové

Přístupové metody

- DCF, Distributed Coordination Function, založená na CSMA-CA
- PCF Point Coordination Function, koordinace založená na koordinátorovi
  - cez AP, mix periód DCF a CFP (vtedy rozosiela pakety staniciam, ze mozu vysielat)

BSS, prepojene cez (Wireless-)DS; spojenim BSS => ExtendedSS (mobilita)

- SSID, BSSID

rámce: PLCP (rychlost prenosu) | Mac (RTS/CTS, ACK) | management (probe, asociacia,autentizacia) | data frames

4 adresy v MAC (sndr, rcvr + bity to DS, from DS)

kolizie

- 0-perzistencia (CSMA/CA — ACK)

prekryvajuce sa frekv. pasma

DHSS (bit => napr. bakerov kod; vo wifi nie kvoli multiplexu — ale kvoli sumu)

OFDM (ortho. freq. division multiplex — prekryvajuce sa pasma, v kazdom pasme iny nosny signal)

PBCC

25.16 RM ISO/OSI

see also wen:OSI model

Referencni model ISO/OSI (International Organisation for Standardisation / Open Systems Interconnection)

Pokus vytvorit univerzalni sitovou architekturu standardizacni agenturou. Reseni od "zeleneho stolu". Nebylo plne dodelano, protokoly se dodelavaly az pozdeji postupne. Zacal "velkyma ocima", ze kterych bylo postupne ustupovano kvuli neimplementovatelnosti celeho modelu. Ze zacatku hodne prosazovan statnimi institucemi, ktere ve svych zakazkach vyzadovaly kompatibilitu.

RM ISO OSI obsahuje sedm vrstev, kazda vrstva ke svemu fungovani pouziva vrstvy primo pod ni na stejnem pocitaci, mezi pocitaci fakticky komunikuje jen fyzicka vrstva.

1. Fyzicka — prenasi bity. Kvuli tomu resi kodovani, modulaci, synchronizaci, ... na jeji urovni se rozlisuje paralelni a seriovy prenos, synchronni a asynchronni,...
2. Linkova — prenasi ramce k primym sousedum. Kvuli pretizenosti se rozpadla na dve podvrstvy — LLC a MAC. Zajistuje synchronizaci na urovni ramcu (detekce zacatku a konce ramcu). Muze fungovat spojovane i nespojovane, spolehlive i nespolehlive, best eff. i QoS. Ridi tok dat a pristup ke sdilenemu mediu
   - MAC dělá kanál na sdíleném médiu, adresování, řeší kolize
   - LLC multiplexing, řízení toku (kdy vysílat aby se druhý konec neucpal)
3. Sitova vrstva — prenasi datagramy pomoci smerovani do cele site. Muze pouzivat ruzne routovaci algoritmy. Je to nejvyssi vrstva, kterou musi mit smerovace (routery). Routing — hledani cesty k adresatovi, forwarding — poslani packetu tou cestou.
4. Transportni vrstva — muze pridavat sluzby — napr. nizsi vrstvy (patrici nekomu jinemu) nabizi jen nespolehlivou komunikaci a transportni vrstva doda (pripadne zvysi) spolehlivost. Pr. TCP (spolehlivy) nad IP (nespolehlivy) (ten priklad neni z ISO OSI, ale ze transportni vrstvy rodiny TCP/IP). Transportni vrstav pridava "delitelnost" uzlu — uz nestaci jen adresa, ale je treba i port pro identifikaci adresata.
5. Relacni vrstva — muze zajistovat sifrovani, podporu transakci, sessions. Je to nejvice kritizovana soucast RM ISO/OSI. V TCP/IP neni zastoupena a řeší si to aplikace. `http://www.earchiv.cz/a92/a225c110.php3`
6. Prezentacni vrstva — prevod dat do/z tvaru vhodneho k prenaseni — napr. kodovani (ascii, ebcdic, little endian vs big endian) linearizace (dvourozmerne pole -> 1D),...
7. Aplikacni vrstva — puvodne mela obsahova aplikace, ale nakonec jen jadro aplikaci, ktere ma smysl standardizovat — napr. prenos el. posty.

nevyhody modelu: vyroben od zeleneho stolu, prilis slozity, nektere funkce opakovany v ruznych vrstvach, upřednostnuje spojovane spolehlive sluzby sveta spoju, maly duraz na propojovani siti.

25.17 Aktivní prvky (bridge, routery)

Směrovač (router)

Zarizeni fungujici na urovni sitove vrstvy. Propojuje dve nebo vice siti (ma dve nebo vice sitovych rozhrani — adres). Funguje v prostredi prepojovani packetu. Stara se o smerovani packetu (routing) i jejich posilani (forwarding). Sklada se z prepojovaciho pole a smerovaciho procesoru. Procesor urcuje smer dalsiho putovani packetu, pres prepojovaci pole se packet fyzicky prenese z vstupni fronty do vystupni fronty.

Smerovac pro urceni dalsi cesty pouziva smerovaci tabulky (routing tables), pripadne je nepouziva u "nouzovych" metod typu flood nebo random. Smerovaci tabulka obsahuje trojice jake site, jakou cenou (v nejake predem stanovene metrice), pres jakeho primeho souseda.

Adaptivni algoritmy prubezne upravuji smer. tabulky, neadaptivni ne.

metody smerovani:

- flood — posli vsem krome zdroje — pro aktualizacni informace, pripadne ve vojenskych sitich.
- random — posli komukoli — kdyz se mi zacinaji zahlcovat buffery a nejde to poslat tam, kam to ma jit
- distribuovane smerovani — Nejcasteji uzly spolupracuji na hledani optimalni cesty. Dnes prevazne link state routing metoda.
  - OSPF (open shortest path first) protokol. Poslu vsem! vzdalenosti k primym sousedum a kazdy uzel si z nich pocita nejkratsi cesty sam. Ani tohle neni kvuli velikosti tabulek dostatecne pro opravdu VELKE site.
  - hiearchicke smerovani — rozdelim sit na bloky, ze kterych se nebudou sirit detailni smerovaci informace, ale jen informace typu informace ve smyslu: "pres mne jsou dostupné síte X.Y.Z až X.Y.W". Tuto sumarizaci provadi "hranicni smerovac". Pres paterni sit je nutno pouzit exterior gateway protokol (BGP), v "moji" siti nejaky interior gateway p. (třeba OSPF, nebo něco statického)

Most (bridge), přepínač (switch)

Zarizeni na linkove vrstve. spojujici jednotlive pocitace do "site". Muze byt zapojeno k sobe i vice switchu. Jednotlive site od sebe oddeluji routery. Bufferuje data a tim muze spojovat jednotlive segmenty site, ktere nepracuji stejnymi rychlostmi. Pracuje transparentne (na urovni linkove vrstvy), zastavuje kolize Ethernetu, podporuje broadcast.

Metodou zpetneho uceni se uci topologii nejblizsiho okoli (jen sit — po router). Nejdrive nevi nic a co dostane broadcastuje vsem. Zapamatuje si adresy tech co mu neco poslou a kdyz jim je neco v budoucnu smerovano, tak uz to posle jen jim. Problem moznych cyklu v siti (od A mi to prislo pres B i C) resi inteligentni switche vyhledanim kostry.

Pro posilani v ramci site se muze pouzivat source routing (tedy ne v Ethernetu; neplest s routing v sitove vrstve) — ramec ma v hlavicce navigaci — uplny itinerar uzlu, pres ktere vede cesta k cili. Ten se zjisti specialnim pruzkumnym packetem, ktery se pred tim poslal floodem a vratil se.

Rozdil mezi switchem a bridgem je v tom, ze switch umi jen stejne/podobne technologie (prepoji site 10 a 100mbit ehternet), kdezto bridge je pomalejsi ve sve funkci, ale za to treba spoji token ring a ethernet.

Opakovač (repeater), hub

Zarizeni fyzicke vrstvy, jen hloupy zesilovac signalu — co mu prijde z jedne strany vysle na druhou (nebo na všechny ostatní v případě hubu), a tim "prodluzuje" sit. Mnozstvi repeaterů zapojitelnych v Ethernetu je omezene (max 5 segmentů mezi dvěma stanicemi), aby se stihly detekovat kolize. Dulezite na repeateru je, ze uz se nepouziva, protoze se ze sdileneho media (sbernice — koax) preslo na point2point (kroucene dvoulinky do switche).

25.18 Síťový model TCP/IP, IPv6

Rodina protokolů (asi 100) TCP/IP, na které je postavený dnešní Internet. Ucelena predstava o sitove architekture a poctu a ukolech vrstev. Oproti RM ISO/OSI obsahuje jen ctyri vrstvy.

Vznikalo pomalu, postupnym pridavanim v akademickem prostredi, ale prosadilo se i v komercnim. Dnes je to nejrozsirenejsi sitova architektura fungujici nad jakoukoli linkovou technologii.

Protokoly byly vyvijeny jako definitivni reseni pro provoz vznikajiciho Internetu. Mely nahradit prozatimni Network Control Protocol Arpanetu — site financovane ministertvem obrany USA (Vint Cerf — otec internetu). Filozofie – musi to byt decentralizovane a nejak fungovat, kdyz cast fungovat prestane (bude znicena treba). Nebylo pozadovano zabezpeceni, mobilita... TCP/IP vymysleno tak, aby sly ruzne site s ruznymi technologiemi pripojovat k ARPANETu (a tim tvori Internet).

Vrstvy

1. Aplikacni vrstva — jednotne zaklady aplikaci — email, prenos souboru, http, xmpp, etc
2. Transportni vrstva — jednotne transportni protokoly — TCP a UDP
3. Sitova vrstva — prenosovy protokol IP
4. Vrstva sitoveho rozhrani — TCP/IP nedefinuje — at si tady je co chce, kdyz to bude poskytovat sluzby potrebne pro fungovani IP protokolu. napr. Ethernet, Token Ring, ATM, ...

porovnani s RM ISO/OSI:

- aplikacni vrstva ˜ aplikacni + prezentacni + relacni
- transportni vrstva ˜ transportni v.
- sitova v. (IP vrstva) ˜ sitova v.
- v. sitoveho rozhrani ˜ linkova + fyzicka

IP

Protokol IP – nespolehlivy, nespojovany, best effort, 32bitove adresy (IPv4). Vyssi vrstvy vidi pouze MTU — maximum transfer unit — velikost ramce pro danou technologii pod IP. Pokud ji nerespektuji, dochazi k fragmentaci (IP to rozdeli, ale to je plytvani).

IP adresy logicky dvouslozkove adresa site + adresa uzlu. Predpokladaji se dva typy uzlu v siti — koncove (pc, tiskarny,servery,...) a routery, ktere spojuji jednotlive "site". Adresa site byla puvodne vyjadrena bud 8 bity (trida A), 16 (trida B) nebo 24 (trida C) a jednotlivym zadatelum se pridelovaly jednotlive tridy.

S rozvojem pocitacu zacinaji adresy dochazet, coz se resi:

- Subnetting — deleni jiz pridelenych trid.
- CIDR – uz jde pridelovat jakoukoli 2^n mnozinu adres ne jen tri ruzne tridy.
- privatni IP adresy (10.*; 192.168.*) — jsou videt jen v ramci jedne site, ne pres router, do zbytku světa připojené přes NAT nebo různé proxy
- IPv6

ostatní protokoly a TLA

- TCP — spojovany spolehlivy (emuluje toto chovani nad nespojovanym nespolehlivym IP) protokol v transportni vrstve (vyuziva jej napr. SMTP, FTP, Telnet, HTTP)
- UDP — nespojovany nespolehlivy protokol v transportni vrstve (DHCP, RPC, NFS, větší část DNS), víceméně jen obal IP s porty
- RFC — public domain standardy TCP/IP, vznikaji az na zaklade fungujici technologie (firmy dnes predkladaji ke standardizaci sve technologie)
- Internet Socieety -> IAB (ridi standardizaci, vydava RFC)
- Webove standardy vydava W3C, kopie jsou vydavany jako RFC

Packet header analysis IP header Ethernet frames
TCP/IP obecně má:

- vyhody: efektivni, nevnucuje rezii, kterou nepotrebujeme
- nevyhody: nehodi se na nektere nove technologie (VOIP, Video on Demand), ktere by potrebovaly QoS, nikoli BestEffort (lze resit prioritizaci — MPLS nebo rezervaci RSVP – zajistuje rezervaci zdroju "pod" IP); neposkytuji zadne zabezpeceni (ze zadani)

Srovnání s RM ISO/OSI

TCP/IP	ISO/OSI
prijme jedn. reseni, pak pridava	zacne na velkem, pak ubira, protoze nezvlada
prijima jen realizovane veci	neni nutne overeni realizovatelnosti
standardy jsou volne dostupne	standardy jsou prodavany (draze)

IPv6

Naslednik IPv4 (ten prevazne pouzivany nyni). Asi nejdulezitejsi zmenou je rozsireni adresoveho prostoru z 32 bitu na 128 a tim vyreseni problemu s nedostatkem adres "na vzdy".

Obsahuje ale i jine nove moznosti:

- multicast přímo ve specifikaci
- mobilita (dnes zatím nepoužito)
- link-local adresy
- IPSec – šifrování autentizace přímo ve specifikaci
- autokonfigurace (postup pro získání link-local adresy, pak router solicitation, ...)

Nejedna se o zasadni zmeny od IPv4, spise, podle filosofie protokolu TCP/IP o vylepseni, ktere si "zadaji uzivatele".
Další odlišnosti:

- nemá checksum v hlavičce (kde se neustále mění TTL/hop limit), místo toho spoléhá na vrstvy okolo. routery tak nemusejí neustále počítat měnící se součty
- pakety se nefragmentují na cestě, ale jen u odesílatele, předpokládá se použití Path MTU discovery

Adresace v ipv6

z wcs:ipv6 Adresy IPv6 se dělí do tří kategorií[1]1RFC 2373 — 'Architektura adresování IPv6':

- unicast adresy
- multicast adresy
- anycast adresy

Unicast adresa reprezentuje jednotlivé síťové rozhraní. Paket zaslaný na unicast adresu je doručen konkrétnímu počítači. Následující typy adres jsou IPv6 unicast adresy:

- globální unicast adresy
- adresy místní linky
- adresy místní stránky
- unikátní lokální IPv6 unicast adresy
- speciální adresy

Multicast adresy jsou používány k definování množiny rozhraní obvykle patřících různým uzlům, nikoli pouze jednomu. Paket zaslaný na multicast adresu je protokolem doručen všem rozhraním určeným touto adresou. Multicast adresy mají prefix FF00::/8 a jejich druhý oktet určuje dosah adresy, tzn. rozsah v jakém je multicast adresa zviditelněna. Běžně využívány jsou rozsahy místní linky (0x2), místní stránky (0x5) a globální (0xE). Anycast adresy jsou také přiřazeny více než jednomu rozhraní, patřící rozdílným uzlům. Nicméně paket vyslaný na anycast adresu je obvykle doručen pouze jednomu z členských rozhraní, typicky „nejbližšímu“ vzhledem k představě směrovacího protokolu o vzdálenosti. Anycast adresy nemohou být snadno identifikovány, mají strukturu bežné unicast adresy a liší se pouze zaváděním do směrovacího protokolu na více místech v síti.

Speciální adresy

Existuje množství adres které mají speciální význam v IPv6:

Místní linka

- `::/128` — adresa samých nul je nespecifikovaná adresa a je používána pouze v software.
- `::1/128` — adresa místní smyčky je adresa pro localhost. Pokud aplikace vyšle paket na tuto adresu, IPv6 paket je vrácen zpět na téhož hosta (odpovídá 127.0.0.1 v IPv4).
- `fe80::/10` — prefix místní linky udává platnost adresy pouze v místní fyzické lince. Analogické autokonfigurační IPv4 adrese 169.254.0.0/16.

Místní stránka

- `fc00::/7` — unikátní lokální adresy (ULA) jsou směrovatelné pouze v množině spolupracujících stránek. Adresy zahrnují 40 bitové pseudonáhodné číslo minimalizující nebezpečí konfiktu při sloučení stránek či úniku paketů.

IPv4

- `::ffff:0:0/96` — tento prefix se používá pro IPv4 mapované adresy (viz Mechanismy přechodu níže).
- `2002::/16` — pro adresování tunelu IPv6 v IPv4.

Multicast

- `ff00::/8` — použití pro multicast adresy

Užito v příkladech, neschváleno, či zastaralé

- `::/96` — nulový prefix se používal pro IPv4 kompatibilní adresy. Nyní zastaralý.
- `2001:db8::/32` — použito v dokumentaci. Kdekoliv je dán příklad IPv6 adresy, měl by mít tento prefix.
- `fec0::/10` — prefix místní stránky udává platnost adresy pouze uvnitř místní organizace. Použití bylo v září 2004 odmítnuto dle RFC 3879 a systémy nesmí podporovat tento speciální typ adres.

[1]

V protokolu IPv6 nejsou vymezeny žádné rozsahy adres pro broadcast — aplikace používají multicast ke skupině all-hosts

Zónové rejstříky

Určitý problém představují adresy lokální linky pro systémy s více než jedním rozhraním. Jelikož každé rozhraní může být připojeno k jiné síti a všechny adresy se zdají být na stejné podsíti, objevuje se nejednoznačnost neřešitelná směrovacími tabulkami Například host A má dvě rozhraní a těmto rozhraním jsou při aktivaci automaticky přiřazeny adresy místní linky : fe80::1/64 a fe80::2/64. Pouze jedno z rozhraní je připojeno do stejné fyzické sítě jako host B s adresou fe80::3/64. V případě že host A se pokusí o spojení s fe80::3, jak určí rozhraní které má použít? O tom pak rozhodují zónové rejstříky

25.19 Přenosové služby počítačových sítí

Spolehlivé a nespolehlivé

spolehliva sluzba ten, kdo data prenasi zodpovida za doruceni (proto je nutna detekce chyb, zadosti o opetovny prenos).

nespolehliva sluzba je ji jedno, zda se to doruci nebo ne. Se spolehlivosti je spojena nenulova rezie, ktera muze byt nezadouci. Napr. multimedialni prenos potrebuje dodavat pravidelne hodne dat a pripadna ztrata casti z nich boli mene nez zdrzeni vetsiny kvuli znovuposlani.

best effort typ prenosu, kdy se sit "snazi" a pokud to uz nejde, tak jsou pozadavky stejnomerne kraceny. pr. IP

Quality of Service Obecné oznacení pro variantu, kdy prenosová sí dokáže rozlišovat mezi jednotlivými prenosy a nabízet jim ruznou "kvalitu prenosu" (QoS). Muze nabizet garanci sluzeb (resi se rezervaci zdroju) a take nemusi.

Spojované a nespojované

spojovana komunikace

1. nejdrive se navaze spojeni v rámci navázání spojení je nalezena (a vyznacena) trasa penosu
2. probiha komunikace
3. spojeni je ukonceno, pripadne zdroje vraceny

- komunikace je stavova (alespoň neni spojeni / je spojeni). je treba osetrit prechod mezi jednotlivymi stavy a nestandardni situace (jeden ucastnik spadne apod.)
- zachovava poradi (packety se nemohou zprehazet, kdyz cestuji stejnou cestou)
- analogie — telefonni hovor (vytocim, mluvim, polozim)

Prepojovani okruhu – zpusob spojovaneho prenosu. Jednotlive uzly site "vyriznou" z dostupne kapacity tolik, o kolik si komunikujici strany reknou (a taky to pak nauctuji, rezervace je drahá). Vyrabi iluzi jednoho dratu mezi zdrojem a prijemcem, ktery ma vsude stejnou kapacitu. Odhadnutelne zpozdeni na ceste, vyuziva se ve svete telekomunikaci, protoze je vhodne pro multimedia, malo ve svete pocitacu (snad jen seriove komunikace).

nespojovaná komunikace

nenavazuji spojeni, neoveruji, ze druha strana existuje, neni treba ukoncovat spojeni. komunikace probiha formou posilani zprav — datagramu

- bezestavova
- každý datagram obsahuje v hlavicce plnou adresu prijemce
- nezarucuje poradi doruceni
- trasa pro datagram je hledana vzdy znovu
- analogie — listovni posta

Prepojovani packetu – nic se nevyhrazuje, stale je snaha vyuzit vsechny dostupne zdroje (best effort). Pouziva se prakticky ve vsech sitich, vhodne pro datove prenosy. Datagramy museji obsahovat adresy odesilatele a prijemce. Ma smysl jen blokovy prenos, proudovy se emuluje. Diky slozitejsi logice zpracovani v uzlech ma vetsi prenosove zpozdeni nez prepojovani okruhu a je nerovnomerne. Uzly funguji na principu Store & Forward. Muze fungovat spojovane (virtualni okruhy) i nespojovane (datagramova sluzba).

25.20 Přenos a sdílení dat

není moc jasné co tahle otázka vlastně znamená, následují nějaké spekulace

Z hlediska (lowlevel) techniky prenosu dat

proudovy prenos / blokovy prenos

- Proudovy prenos (stream)
  - predavano po jednotlivych b[iy]tech.
  - Zadna hlavicka, prijemce je ten na druhe strane kanalu. Predpoklada se spojovana komunikace.
- Blokovy prenos
  - prenaseno po blocich. Blok je nazyvan podle vrstvy, ve ktere je prevod realizovan,zpusobu prenosu nebo velikosti bloku:
    - packet — síťová vrstva
    - rámec (frame) — linkova vrstva
    - zpráva — aplikacni vrstva
    - datagram — obecny nespojovane prenaseny blok (ramec je datagram)
  - Velikost bloku je promenna, ale shora omezena

Podle synchronizace

- synchronni — hodinky zdroje a prijemce jsou stale aktualizovany (treba v ethernetu se posilaji data jen kazdou druhou dobu, ty druhe "kazde druhe (sudy/lichy) jsou pro synchronizaci)
- asynchronni — postrada jakoukoli synchronizaci, potrebuje 3hodnotovou logiku (“1”, “0”, “nic”), nepouziva se
- arytmicky — terminologicky zamenovan s asynchronnim — muze velke mezery mezi vysilanim, zesynchronizuji se vzdy na zacatku vysilaneho bloku (ktery ma jen 4-8b, coz vydrzi synchronizovane) “start” bitem.

Podle směrovosti

- simplex — jendosmerny kanal
- duplex — obousmerny,
- semiduplex — lze prenaset v obou smerech, ale ne najednou

Z hlediska prenosu dat pomoci aplikacnich protokolu

Kazdopadne musi obe strany dodrzovat nejaky dohodnuty protokol.

- pro prenos souboru — FTP — na soubory se divame jinak, jsou-li ulozeny lokalne nebo vzdalene. Uzivatel musi explicitne kopirovat data z jedne lokaci do druhe (get/put). Funguje typem klient/server. active/passive(vsechna spojen navazuje klient — kvuli fw) mod. port serveru 21 — prikazy, 20 — data, u klienta nahodny.
- pro sdileni souboru — NFS, AFS, SMB (wokenni sdileni) — soubory lze vnimat stejne (byt muzou mit ruznou dobu pristupu), o stehovani z jednotlivych lokaci se stara system.

Soubory jde posílat přenášet obří hromadou dalších protokolů. Pod sdílení se asi kromě síťových filesystémů dají zahrnout i FTP, web, různé content distribution networks a p2p věci jako je BitTorrent.

25.21 Elektronická pošta

služba, je moznost realizovat ji interne ruzne (smtp vs MS mail vs X.400). V internetu se pouziva SMTP. Krome protokolu pro prenos zprav mezi servery a klientem (SMTP), je treba dodrzovat jeste spravny format (rfc822 — hlavicka, telo, pripadne prilohy), zajistit zpusob stahovani zprav ze schranky na serveru (imap, pop3), rozsireni pro napr. narodni abecedy, urceni typu prilohy (MIME).

Kazda zprava ma:

- header — komu, od koho, predmet, datum poslani, ...
- telo — obsah
- prilohy

Přenos zprávy

1. sestavení zprávy, příkaz k odeslání
2. upload (odeslání) zprávy na poštovní server, pomocí SMTP
3. přenos zprávy mezi poštovními servery
   - pomocí protokolu SMTP
   - zpráva končí v poštovní schránce (mailboxu) příjemce
4. stažení zprávy z poštovní schránky do poštovního klienta
   - pomocí POP3/IMAP
5. čtení přijaté zprávy, v rámci poštovního klienta příjemce

- MX záznamy v DNS — pro kazdou domenu je definovan jeden nebo vice serveru, ktere pro ni prijimaji postu
- SMTP garantuje prenaseni jen 7bitovych znaku (ASCII), pro prenos 8bitovych je treba je prevest do 7bitu — jak to udelat resi MIME.
- IMAP vs POP3 — IMAP dovoluje pracovat s daty na serveru, je urcen pro stala pripojeni typu pevna linka/adsl. POP3 umi zpravy stahovat na lokal — vhodne pro modemisty

Spam

Jak na spam:

- Bayes filtry
- DNSBL
- greylisting
- SPF (Sender Policy Framework)

25.22 Služby zpřístupnění informací

Po Gopheru dnes WWW.

WWW

Sada HTML stránek a dalších médií okolo nich propojených odkazy. Data se přenášejí pomocí protokolu HTTP/HTTPS. Funguje na architektuře klient/server.

HTTP je bezestavový protokol (jen request/response), stavovost se do něj doplňuje typicky přes cookies. HTTP request je nejčastěji GET/POST/HEAD, ale jsou i PUT/DELETE a pár dalších (see wen:Hypertext Transfer Protocol#Request methods).

V odpověd přijde nějaký ze status kódů (200, 404, 302, ...) a obsah.

Proxy, P2P

proxy proxy server je jakysi prostrednik mezi klientem a serverem, ktery vyrizuje pozadavek. Klient posle pozadavek proxy serveru (o cemz uzivatel ani nemusi vedet a typicky si nemuze vybrat, zda proxy server pouzije ci nikoli) a ten jej vyridi bud ze sve cache nebo dotazem na dany server a pak preposlanim odpovedi. Pouziva se z duvodu bezpecnostnich (jediny bod site pristupny z internetu, vsechny lokalni pocitace jsou az "za nim"), vykonovym (cacheovanim muze usetrit spoustu prenosove kapacity), pripadne jinym (napr. povoleni jen urcitych stranek — pokus o pristup na jine proxy server nedovoli)

Peer to peer site (Peer2Peer, P2P) vymenne datove site (bittorrent, KaZaa, Napster, DirectConnect...), kde si data predavaji dva(vice) koncovych uzivatelu, nejsou stahovana z zadneho centralniho serveru.

25.23 Bezpečnost síťového přístupu, zabezpečené protokoly

Tady je stranka, ktera slusne pokryva toto tema. Jasne a do hloubky napsane: RAD security

Bezny sitovy provoz je tvoren v "dobre vire". Kdokoli je na trase, tak muze do zprav nakukovat, i je menit. To samozrejme neni pro nektere typy komunikaci dostatecne. Aby byla sitova komunikace bezpecna, mela by zajistovat tyto veci:

- identifikace (zjisteni identity ucastniku — ten druhy je tim, za koho se vydava),
- autorizace (overeni pristupovych prav — nelez/nesahej kam nemas)
- duvernost (sifrovani komunikace — nikdo si to neprecte)
- integritu (taky sifrovani — nikdo to nezmeni).

Protokoly pro bezpecny prenos informaci — z datoveho toku neni poznat, co je prenaseno. Resi se sifrovanim. Pro "uplne" (kazda sifra muze byt prolomena) bezpecne pouziti je nutne identifikatory ucastniku (fingerprinty, jmena-/hesla) prenest bezpecnou cestou (rucne lokalne opsat, nikoli stahnout nezabezpecene z webu). Bezpecnost zavisi na bezpecnosti pouzite sifry a korektni implementaci algoritmu na obou stranach.

Identita ucastniku a navazani spojeni se resi pomoci asymetrickeho sifrovani, vlastni vymena pak pomoci symetrickeho(pretoze symetricke su rychlejsia nez asymetricke). Identita je dana public klicem ucastnika (nebo jeho fingerprintem). To se typicky deje jen pro server a uzivatel svou identitu serveru "prokazuje" pozdeji pomoci jmena a hesla.

SSL, secure socket layer protokol v rodine tcp/ip slouzici k bezpecnemu provozu jinych protokolu. Je zalozen na sifrovani a muze pouzit nekolik ruznych algoritmu. Pouziti se sestava ze tri kroku: 1. Dohodnuti se na algoritmech, 2. autentikace ucastniku pomoci verejnych klicu (certifikátů), serveru povině a klienta volitelně 3. vymena dat pomoci symetrickych sifer. Myšlenaka byla vytvořit zabezpečené sockety typu berkley, původně hlavně pro protokol HTTP.

https — secure obdoba http protokolu jde o vymenu dat beznym http protokolem provozovanym nad SSL (secure socket layer)

ssh (secure shell) pro bezpecne terminalove spojeni se se vzdalenym serverem, dá se použít pro tunelování protokolů na TCP/IP, např HTTP.

sftp secure file transfer protocol, je nad ssh

FTP jde pomocí SSH zabezpečit řídící spojení, to je nad TCP, ale datové ne, to je nad UDP.

Do SSL/TLS se dá balit i SMTP, POP3, XMPP, ... Pro paranoiky jde i Tor a tak.

25.24 Překlad adres

NAT — network address translation — technika prekladu sitovych adres, ktery se deje na firewallu nebo routeru. Pri pouziti se vzdy projevi tyto cile/dusledky:

- setreni jmeneho prostoru IPv4 — privatni adresy mohou byt pouzity pro spoustu pocitacu za NATem.
- sdileni internetoveho pripojeni — internet provider poskytuje pouze jednu sitovou adresu a my chceme pripojit vice pocitacu.
- bezpecnost — pocitace za NATem nemaji verejnou adresu a neda se na ne z internetu poslat request, protoze skonci na NAT routeru.

Krome techto "vlastnosti", ktere se projevi pri kazdem pouziti NATu, lze tuto techniku pouzit i pro dosazeni jinych cilu:

- LoadBalancing — nekolik fyzickych serveru za jednou ip adresou a "NATovadlo :)", ktere rozdeluje pozadavky
- Failover — server a jeho backup za jednou ip adresou. Kdyz server selze, tak se zacne pouzivat backup, aniz by o tom kdokoli vedel.

priklad — dnesni domaci routery, iptables.

Rozliseni, ktery pocitac v privatni casti ma dostat odpoved na svuj pozadavek se dela pomoci cisel portu (proto nekdy nazyvano PAT) — router pro pozadavek pocitace prideli nejaky port a "z nej" to potom posle do internetu. Kdyz prijde odpoved na tento port, tak router pak vi, komu to ma preposlat. Preposilani obema smery se deje prepisovanim adres v hlavickach a ponechani tela (payload) packetu. Proto protokoly, ktere maji adresu i v tele (IPSec) nebo navazují dodatečná spojení směrem ke klientovi (active mód FTP) nefunguji za NATem (bez explicitni pomoci od NATovadla).

Routeru se dá nastavit aby některé požadavky na sebe přeposílal daným počítačům ve vnitřní síti (port forwarding). Toho se dá využít ke zpřístupnění některých zdrojů ve vnitřní síti ven (třeba pro p2p sítě ;-). Protokol UPnP umožňuje tyto tunely vytvořit zevnitř bez speciálního nastavování routeru.

25.25 Firewally

Firewall je zařízení slouzici ke kontrole a případnému povolení/zakázání nějakého síťového provozu. V podstatě odděluje vnější síť (typicky Internet) od vnitřní sítě (LAN nebo i jeden počítač). Lze řešit v HW i SW.

Ve firemní síti firewall třeba zakazuje veškerý HTTP provoz všemu kromě proxy serveru, a zabraňuje navazování spojení zvenku. Dále se filtruje třeba SMTP, nebo některý druh přístupu pouze z důvěryhodných sítí, ...

V jednom PC je firewall softwarove který kontroluje příchozí a odchozí spojení i na úrovni aplikací – můžu povolit spojení ven prohlížeči, ale už ho třeba zakázal malwaru.

25.26 Certifikáty

Digitální certifikát: obecné označení pro údaje, týkající se určitého subjektu a stvrzené jiným subjektem, který se zaručuje za jejich pravost (tzv. certifikační autoritou — u nas 1.CA, ve svete VeriSign). Nejčastěji je v certifikátu obsažen veřejný klíč vlastníka certifikátu, který má být veřejně přístupný (ale mohou zde být obsaženy i další údaje).

Údaje v certifikátu jsou chráněny pomocí asymetrických šifrovacích technik — jsou zašifrovány privátním klíčem vydavatele certifikátu (certifikační autority), a mohou být dešifrovány s použitím veřejného klíče certifikační autority (který je veřejně známý). Význam a věrohodnost certifikátu jsou závislé jak na věrohodnosti samotné certifikační autority, tak i na způsobu, jakým tato autorita získává a ověřuje údaje, které svým certifikátem stvrzuje.

Treba pro certifikat pouzitelny pro digitalni podpis pro komunikaci se statni spravou musite jit osobne s obcankou do 1.CA. Nikdo jiny zatim nema "licenci" na to, aby tyto certifikaty vydaval. Naproti tomu Verisign vydava vice "druhu" certifikatu, podle toho jak byla overena identita.

Pro uplnou funkci je treba pouzivat i casove znamky (timestamp). Jinak bych u cehokoli mohl tvrdit, ze to bylo podepsano nekym jinym mym certifikatem az potom, co jsem jej prohlasil za neplatny a vygeneroval si jiny.

Ověření certifikátů se tedy dáj zřetězovat. Řekněme, že certifikát autority A má každý, např už jako součást prohlížeče nebo OS. A podepíše certifikát autoritám AA, BB, CC a do certifikátu je zaznamenenáno, kdo certifikát podepsal (jak sehnat certifikát podepisovače). AA nebo BB nebo CC podepíše certifikáty AAA, BBB, CCC, DDD, EEE, a tak dále. Pokud jsme schopni sehnat všechny certifikáty v tomto stromě od listu ke kořeni, tak jsem schopni ověřit, zda je certifikát v listu důvěryhodný (je tak důvěryhodný jako nejméně důvěryhodná autorita na cestě do kořene)

25.27 VPN

viz http://cs.wikipedia.org/wiki/Virtu%C3%A1ln%C3%AD_priv%C3%A1tn%C3%AD_s%C3%AD%C5%A5

Z fyzickeho pohledu jde o součást jiné sítě. Z logického pohledu se jedná o samostatnou sit. Muze mit vlastni adresovaní, do jinych siti pristup jen pres branu aj.

Duvod vzniku: uzivatel chce mit vlastni sit, ale nevyplati se mu ji fyzicky budovat. Jeji "simulace" je levnejsi. Provoz ve VPN je sifrovan, aby byl bezpecny. poskytuje typicky sluzby identifikace (zjisteni identity), autentizace (overeni pristupovych prav) u kazdeho vstupu uzivatele (pripojeni z domova, z jine pobocky) pres verejnou sit (napr. internet). Dale zajistuje duvernost (sifrovani komunikace — nikdo si to neprecte) a integritu (taky sifrovani — nikdo to nezmeni, aniz by se to poznalo).

Priklady pouziti:

- uzivatel z domu do firemni site, nekolik pobocek dohromady se tvari jako jedna sit
- ve svete telekomunikaci se vyskytuji sluzby jako "volani zdarma v ramci firmy", jejiz telefony tvori VPN (coz jsou teda asi spis jen jina cisla a tarifikace nez specialni sifrovani)

Kapitola 26

Geometrické modelování a výpočetní geometrie

26.1 Rozsah látky

Seznam oficiálních státnicových otázek:

Projektivní rozšíření afinního prostoru, homogenní souřadnice, afinní a projektivní transformace v rovině a v prostoru, kvaterniony v reprezentaci 3D orientace, diferenciální geometrie křivek a ploch, základní spline funkce, kubické spliny C2 a jejich vlastnosti, interpolace kubickými spliny, Bézierovy křivky, Catmull-Rom spliny, B-spline, de Casteljaův a de Boorův algoritmus, aproximační plochy, plochy zadané okrajem, Bezierovy plochy, plátování, B-spline plochy, NURBS plochy, základní věty o konvexitě, kombinatorická složitost konvexních mnohostěnů, návrh geometrických algoritmů a jejich složitost, Voroného diagram a Delaunayova triangulace, konvexní obal, lokalizace, datové struktury a algoritmy pro efektivní prostorové vyhledávání.

26.2 Projektivní rozšíření afinního prostoru, homogenní souřadnice, afinní a projektivní transformace v rovině a v prostoru

Afinní prostor:

- A_n — neprázdná množina bodů
- W_n — vektorový prostor (zaměření)
- $f : A_n \times A_n \to W_n$
- $f(A,B) + f(B,C) = f(A,C)$
- $\forall P \in A_n, \forall X \in A_n : f_P(x) = f(P,X)$ je bijektivní

Běžně $A_n = R^n, W_n = R^n, f(A,B) = B - A$.
Soustava souřadnic
Repér: pevný bod O + báze zaměření
Transformace souřadnic
Lineárně nezávislé body
$B_0, B_1, \ldots, B_n$ jsou LN $\Leftrightarrow (B_1 - B_0), (B_2 - B_0), \ldots, (B_n - B_0)$ jsou LN
Afinní prostor dimenze n je jednoznačně určen n+1 body:

$$X = B_0 + \sum \beta_i (B_i - B_0) = B_0 + \sum \beta_i B_i - \sum \beta_i B_0 = B_0 \underbrace{(1 - \sum \beta_i)}_{\beta_0} + \sum \beta_i B_i$$

```
Afinní kombinace bodů (barycentrické souřadnice)
\(X = \beta_0 B_0 + \beta_1 B_1 + \ldots + \beta_n B_n\), \(\sum \beta_i = 1\)
Konvexní kombinace bodů - navíc požadavek \(\beta_i \geq 0, \forall i\)
```

26.3 Kvaterniony v reprezentaci 3D orientace

- $Q = q_0 + q_1 i + q_2 j + q_3 k = (q_0, (q_1, q_2, q_3)) = (q_0, \overrightarrow{q})$
- $i^2 = j^2 = k^2 = -1$
- $ij = k, ji = -k, \ldots$

Operace s kvaterniony

- $Q \cdot P = (q_0, \overrightarrow{q})(p_0, \overrightarrow{p}) = (q_0 p_0 - \overline{qp}; q_0\overline{p} + p_0\overline{q} + (\overline{q} \times \overline{p})$ — komutativní pokud shodné vektorové části, jinak pouze asociativní a distributivní
- $Q^* = (q_0, -\overrightarrow{q})$
- $Q \cdot Q^* = (q_0^2 + \overline{qq}; q_0\overline{q} - q_0\overline{q} + \overline{0}) = q_0^2 + q_1^2 + q_2^2 + q_3^2$
- $\|Q\| = \sqrt{Q \cdot Q^*}$
- $Q \cdot Q^{-1} = 1$

$$Q^* \cdot Q \cdot Q^{-1} = 1$$

$$Q^* \cdot Q \cdot Q^{-1} = Q^*$$

$$Q^{-1} = \frac{Q^*}{\|Q\|^2}$$

- Věta: $\|Q \cdot P\| = \|Q\| \cdot \|P\|$

Dk: $\|Q \cdot P\| = \sqrt{(Q \cdot P) \cdot (Q \cdot P)^*} = \sqrt{(Q \cdot P) \cdot (P^* \cdot Q^*)} = \sqrt{Q \cdot P \cdot P^* \cdot Q^*} = \sqrt{\|P\| Q \cdot Q^*} = \sqrt{\|P\|^2 \|Q\|^2} = \|Q\| \cdot \|P\|$

Jednotkové kvaterniony

- $\|Q\| = 1$
- tvoří multiplikativní podgrupu
- $Q^{-1} = Q^*$
- Jednotkový kvaternion je tvaru: $Q = (cos(\alpha), sin(\alpha) \cdot \overline{a}), \|\overline{a}\| = 1$

Rotace

Interpolace rotace

1. Lineární Eulerova interpolace
2. Kvaterniony — LERP
3. SLERP — sférická lineární interpolace

26.4 Diferenciální geometrie křivek a ploch

26.5 Základní spline funkce

26.6 Kubické spliny C2 a jejich vlastnosti, interpolace kubickými spliny

26.7 Bézierovy křivky

26.8 Catmull-Rom spliny

26.9 B-spline

26.10 De Casteljaův a de Boorův algoritmus

26.11 Aproximační plochy, plochy zadané okrajem, Bezierovy plochy, plátování, B-spline plochy, NURBS plochy

26.12 Základní věty o konvexitě, kombinatorická složitost konvexních mnohostěnů

26.13 Návrh geometrických algoritmů a jejich složitost

26.14 Voroného diagram a Delaunayova triangulace

26.15 Konvexní obal, lokalizace, datové struktury a algoritmy pro efektivní prostorové vyhledávání

26.16 Materiály

- …

Kapitola 27

Analýza a zpracování obrazu, počítačové vidění a robotika

27.1 Rozsah látky

Seznam oficiálních státnicových otázek:

Matematický model obrazu, 2D Fourierova transformace a konvoluce, vzorkování a kvantování obrazu, změna kontrastu a jasu, odstranění šumu, detekce hran, inverzní a Wienerův filtr, určení vzájemné polohy snímků, problém korespondence bodu a objektu, odstranění geometrických zkreslení snímků, detekce hranic objektů, detekce oblastí, příznaky pro popis a rozpoznávání 2D objektů, momentové invarianty, wavelety a jejich použití, statistická teorie rozpoznávání, klasifikace s učením (Bayessův, lineární a k-NN klasifikátor), klasifikace bez učení (hierarchické a iterační shlukování), počítačové vidění, úvod do počítačové robotiky, plánování cesty mobilního robota.

27.2 Matematický model obrazu

Obrazová funkcia (spojitá), 2D:

$$f : U \subset \mathbb{R}^2 \to \mathbb{R}^n$$

$f : [x, y] \to [a_1, a_2, ..., a_n]$
(poloha bodu v rovine -> atributy obrazu (farba, priehladnost, ... — $\mathbb{R}^4$ pre 3553)))
Digiálny rastrový obraz:

$$I :< 0..m-1 > \times < 0..n-1 > \to \mathbb{R}^n$$

Digitalizácia pomocou filtru $\underline{d}$:

$$I_f(i,j) = \iint_{R^2} f(x,y)d(x-i, y-j)\mathrm{d}x\mathrm{d}y$$

$\underline{d}$ vyjadruje snímaciu charakteristiku digitalizačného zariadenia (fotočidlo, CCD prvok, ...)

27.3 2D Fourierova transformace a konvoluce

Spojité verze

- Dopředná Fourierova transformace: $F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-2\pi i(ux+vy)}dxdy$
- Zpětná Fourierova transformace: $f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{2\pi i(ux+vy)}dudv$
- Konvoluce: $(f*g)(x,y) = (g*f)(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(a,b)g(x-a,y-b)dadb$

Vlastnosti

- Convolution theorem: $\mathcal{F}\{f*g\} = \mathcal{F}\{f\}\cdot\mathcal{F}\{g\}$
- Linearita: $\mathcal{F}\{a\cdot f + b\cdot g\} = a\cdot\mathcal{F}\{f\} + b\cdot\mathcal{F}\{g\}$
- Shift theorem: $\mathcal{F}\{f(x-x_0, y-y_0)\}(u,v) = e^{-2\pi i(ux_0+vy_0)}F(u,v)$
- Rotace: $\mathcal{F}\{Rot(f)\} = Rot(\mathcal{F}\{f\})$

Diskrétní verze

- Dopředná Fourierova transformace: $F_{n,m} = \frac{1}{\sqrt{MN}} \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} f_{k,l} e^{-2\pi i(\frac{km}{M} + \frac{ln}{N})}$
- Zpětná Fourierova transformace: $f_{k,l} = \frac{1}{\sqrt{MN}} \sum_{m=0}^{N-1} \sum_{n=0}^{M-1} F_{n,m} e^{2\pi i(\frac{km}{M} + \frac{ln}{N})}$
- Konvoluce: $(f * g)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j) g(m-i, n-j)$

27.4 Vzorkování a kvantování obrazu

Matematický model vzorkování, Shannon theorem

$f(x,y) \cdot s(x,y) = d(x,y)$, kde f je původní funkce, s je vzorkovací fce (pole delta funkcí) a d je navzorkovaný obraz.

- $F(u,v) * S(u,v) = D(u,v)$
- $s(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(x - i\Delta x, y - j\Delta y)$
- $S(u,v) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \delta(u - i\frac{1}{\Delta x}, v - j\frac{1}{\Delta y})$

Fourierův obraz navzorkované funkce ($D(u,v)$) je tvořen do mřížky poskládanými spektry původní funkce s roztečemi $\frac{1}{\Delta x}$ a $\frac{1}{\Delta y}$. Dokážeme zrekonstruovat původní funkci pouze pokud se nám jednotlivá spektra neslijí a to platí jen pokud je původní funkce frekvenčně omezená a vzorkujeme s dostatečnou frekvencí:

$$\Delta x \leq \frac{1}{2W_u} \text{ a } \Delta y \leq \frac{1}{2W_v}, \text{ kde } W_u \text{ a } W_v \text{ jsou maximální frekvence v základních směrech.}$$

Potřebujeme dvakrát vyšší frekvenci než je maximální přítomná frekvence v původní fci.

Negativní projevy podvzorkování

- aliasing (stráta vysoko frekvenčnej informacie — hrany, detaily)
- Moiré efekt — falešné nízké frekvence

Kvantování

- Diskretizace oboru hodnot signálu — vždy ztrátové.
- Často se kvantizér navrhuje tak aby využíval vlastnosti lidského oka — např. nerozlišitelným jasovým úrovním se přiřazuje stejná hodnota

27.5 Změna kontrastu a jasu

- ekvalizace histogramu
- převodní funkce pro jasové úrovně (LUT — lookup table)
- gamma korekce

27.6 Odstranění šumu

Šum sa vyčísluje ako logaritmus pomeru signalu k šumu v decibeloch `dB` (Signal-to-Noise Ratio). Čím viac decibelov tým lepší odstup signálu od šumu -> kvalitnejší obraz.

$$\text{SNR} = 10log\frac{D(f)}{D(n)}[\text{dB}]$$

f — signál, n — šum

Modely šumu:

- Aditivní náhodný šum $g = f + n$
- Gaussovský bílý šum
- Impulsní šum (sůl a pepř)

Noise reduction: (nedám za to ruku do ohňa)

- bílý šum -> Priemerovanie v čase (prosté/vážené)
- impulsní šum -> Mediánový filter (pre pixel vyberáme intenzitu medianu v okolí), iné nelineárne filtre,
- low-pass filter (napr. Gauss) — zbaví vysokofrekvenčného šumu (rovnako ako aj každej inej vysokofrekvenčnej informácie — hrany)
- Rotujúce okno — pokus o odstranenie vysokofrekvenčného šumu a zachovania hran zároveň. Može vytvárať artefakty.
- Priemerovanie podél hran

27.7 Detekce hran

Lidské vnímání je založeno na detekci hran (edge detection), tedy změnou jasu hrany vidíme objekty. Toho se taktéž ve velké míře používá v segmentačních technikách pro zpracování obrazu. Mnoho metod segmentace právě vychází z detekce hran pro odlišení objektů v obraze. Hranu v obraze je charakterizovat gradientem, tedy velikostí a směrem. Existuje také mnoho filtrů pracující s detekcí hran v obraze a hrany hrají také klíčovou roli pro příznaky a posléze klasifikace podle vektorů příznaků. Mezi geometrické příznaky patří např. pravoúhlost, podlouhlost, kruhovost či vzdálenost pixelů od okraje, tedy hrany. Vektor příznaků, označme jej např. vc =(x1, x2,...,xn), kde xi je daný příznak. Tyto příznaky pak slouží jako vstupy (např. pro perceptron), a pomocí nich se klasifikuje výstup (třída).

27.8 Inverzní a Wienerův filtr

Předpokládáme, že známe funkci, která poškodila obraz.

Ideální případ — bez šumu:

$g(x,y) = (f * h)(x,y)$, kde h je funkce poškození, prostorově neměnná (stejná pro celý obraz).

Z Convolution theoremu dostaneme:

$$G = F \cdot H$$
$$F = G \cdot \frac{1}{H}$$

V praxi je však běžně přítomen i šum, který dekonvoluci stěžuje:

$g(x,y) = (f * h)(x,y) + n(x,y)$, kde n je aditivní šum, nezávislý na obrazové fci.

$$G = F \cdot H + N$$
$$F = G \cdot \frac{1}{H} - \frac{N}{H}$$

Z posledního výrazu je vidět, že šum bude nejvíce ovlivňovat výsledek na frekvencích, kde bude H téměř nulové.

Wienerův filtr

Wienerův filtr se snaží vypořádat se šumem a najít nejlepší opravu obrazu z hlediska nejmenších čtverců (matematicky správné, ale neideální pro člověka)

$$H_W(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + \frac{S_{nn}(u,v)}{S_{ff}(u,v)}}$$

27.9 Určení vzájemné polohy snímků, problém korespondence bodu a objektu

27.10 Detekce hranic objektů, detekce oblastí

27.11 Příznaky pro popis a rozpoznávání 2D objektů, momentové invarianty

27.12 Wavelety a jejich použití

http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html

http://pagesperso-orange.fr/polyvalens/clemens/wavelets/wavelets.html

http://cnx.org/content/m11140/latest/

27.13 Statistická teorie rozpoznávání, klasifikace s učením (Bayessův, lineární a k-NN klasifikátor), klasifikace bez učení (hierarchické a iterační shlukování)

27.14 Počítačové vidění

27.15 Úvod do počítačové robotiky, plánování cesty mobilního robota

27.16 Předměty

- Digitální zpracování obrazu
- Speciální funkce a transformace ve zpracování obrazu
- Rozpoznávání vzorů
- Úvod do mobilní robotiky
- Počítačové vidění a inteligentní robotika

27.17 Materiály

- Slajdy: Flusser J., Zitová B. `http://staff.utia.cas.cz/zitova/prednasky/`, Hlaváč V. `http://cmp.felk.cvut.cz/~hlavac/HlavacTeachPresentCz.htm`, Štanclová J. `http://www.ksi.mff.cuni.cz/~stanclova/`
- Flusser J., Zitová B., Image registration methods: a survey, AVČR `http://library.utia.cas.cz/prace/20030125.pdf`
- Gonzales R. C., Woods R. E., Digital Image Processing `http://www.imageprocessingbook.com/index_dip2e.htm`

Kapitola 28

2D počítačová grafika, komprese obrazu a videa

28.1 Rozsah látky

Seznam oficiálních státnicových otázek:

Výstupní grafická zařízení, plošné útvary — jejich reprezentace a množinové operace s nimi, kreslicí a ořezávací algoritmy v rovině, anti-aliasing, barevné vidění a barevné systémy, reprodukce barevné grafiky, rozptylování a půltónování, kompozice poloprůhledných obrázků, geometrické deformace rastrových obrázků, morphing, základní principy komprese rastrové 2D grafiky, skalární a vektorové kvantování, prediktivní komprese, transformační kompresní metody, hierarchické a progresivní metody, waveletové transformace a jejich celočíselné implementace, kódování koeficientů, komprese videosignálu, časová predikce — kompenzace pohybu, standardy JPEG a MPEG, snímání obrazu v digitální fotografii.

28.2 Výstupní grafická zařízení

- podľa trvanlivosti zobrazenia
  - zobrazovacie zariadenie (display, projector)
  - tiskové zariadenia (tiskárna, plotter, osvitová jednotka)
- podle barevných schopností
  - Č/B zobrazenie (2 barvy)
  - monochromaticke zobrazení (256 odstínů šedi)
  - Barevná paleta (pevná nebo nahrávaná: 16-1024)
  - plná barevnost ("true-color" — maximální barevné využití zobrazovací technologie: 16.7 mil. i více)
- rastrový/vektorový výstup
  - rastrový
    - jsou přímo adresovány jednotlivé pixely
    - data jsou závislá na rozlišení (a nelze je jednoduše škálovat)
  - vektorový
    - zobrazují se přímo složitější objekty (čáry, křivky, písmo)
    - data nejsou závislá na rozlišení (lze je škálovat až v zobrazovacom zařízení)
- podla technologie výstupu:
  - vektorový výstup (staré displeje, plotter, některé osvitové je dnotky)
  - rastrový výstup (displje, tiskárny)
- podle komunikace:
  - vektorové zařízení (urychlované video-adaptéry, plottery, PostScript)
  - rastrové zařízení (bežné video-adaptéry, tiskárny v grafickém režimu)

28.3 Plošné útvary — jejich reprezentace a množinové operace s nimi

28.4 Kreslicí a ořezávací algoritmy v rovině

Kreslení

Kreslení čar

- Kreslení čáry — DDA algoritmus (celočíselní)
  - vyhoda — snadná implementace
  - nevýhody — nutno počítat s velkou přesností (real, fixed point), jedno dělení, v cyklu zaokrouhlování

```
input: x_1, x_2, y_1, y_2, color : integer
1. \(x=x_1, y=y_1\)
2. \(dx=1; dy=\frac{y_2 - y_1}{x_2, x_1}\) (předpokládáme |y_2 - y_1| < |x2-x1|)
3. while (x<=x_2) \(x_{n+1}=x_n+dx, y_{n+1}=y_n+dy\); PutPixel(x, round(y), color);
```

- Kreslení čáry — Bresenhamův algoritmus
  - vyhody — rychlost, iba nasobenie a ščítaní, dá se implementovat bez if-ů
- Kreslení kružnica — Bresenhamův algoritmus
  - kreslí se jen $\frac{1}{8}$ kružnice (tj. while y>x) — zbytek se řeší symetrií

Odvození Bresenhamových algoritmů

- Rozkreslení pixelů, jejich polovin
- Určení základních vztahů pro dané tvary
  - Úsečka — $E' = E - \frac{dy}{dx} <= M = \frac{1}{2}$
  - Kružnice — $F(M) = M_x^2 + M_y^2 - R^2 > 0$
  - Elipsa — kreslí se po čtvrtinách, pozor, od chvíle, kdy dy=dx se posouvá v každém kroku y a ne x!!!
    - $F(M) = b^2 M_x^2 + a^2 M_y^2 - a^2 b^2 > 0$
- Odstranění zlomků, odvození inkrementů(resp. dekrementů)

Vyplňování n-úhelníka

- Seznam hran, dy, dxy (změna x při změně y o 1)
- Hrany orientované shora dolů, setříděné podle y, x, dxy
- Udržuje se seznam aktivních hran při vyplňování
- Zjednodušeně se vyplňují jen liché body

Vyplňování souvislé oblasti

Více druhů:

- Hraniční
- Stejné barvy (záplavové)
- 4-souvislé, 8-souvislé

Lépe využít frontu než zásobník — menší spotřeba paměti.
Řádkový algoritmus

Kreslení písma

Písmo zadáno dvěma různými způsoby:

- Vektorově
  - Pomocí úseček, oblouků, kružnic,...
  - Snadné neprofesionální škálování
  - Při kreslení je nutno resterizovat
- Rastrově
  - Zadáno v bitmapě
  - Snadné vykreslení pomocí BitBlt

Při vykreslování vhodné využití vzoru Flyweight. (Vektorové písmeno se převede jen jednou pro danou velikost, nechá se uloženo v cache)

Ořezávání

28.5 Anti-aliasing

Alias

Alias vzniká při rekonstrukci vzorkovaného singálu. Jedná se o dodatečnou (nežádanou) nízkofrekvenční informaci, která vzniká ve dvou případech:
1. Pokud je původní funkce frekvenčně neomezená — neexistuje maximální frekvence. => při diskrétním zobrazení spojité funkce nikdy nedostaneme přesnou reprezentaci.
2. Pokud je původní funkce frekvenčně omezená, tj. pokud existuje v jejím Fourierovském spektru maximální frekvence fmax. Pokud se taková funkce vzorkuje s frekvencí menší než tzv. Nyquistův limit -> fnyq = 2*fmax, vzniká alias.
Typický příklad — vykreslování šachovnice.

Antialiasing

Problém aliasu se typicky řeší zvýšením prostorového rozlišení na úkor barevného — použitím více odstínů té samé barvy. Pixely i kreslené útvary jsou plošné objekty => pixely rozsvítím s intenzitou úměrnou jejich pokrytí.
Provádí se pomocí supersamplingu — převzorkováním na vyšší rozlišení a následným průměrováním hodnot. Nevýhodou je ztráta informace u vysokofrekvenčních částí — ostré hrany se rozmažou.

- Pravidelné vzorkování — jen přenese problém o řád dál
- Stochastické vzorkování — problém se řeší umělým přidáním šumu
  - Poison disc
  - Jittering (roztřesení)
  - N-věží

28.6 Barevné vidění a barevné systémy, reprodukce barevné grafiky

Barevné vidění

oko — rohovka, duhovka, panenka, čočka, sklivec, sítnice. Žlutá skvrna
Tyčinky — intenzita světla — vnímání kontrastů, vidění za šera, v oku cca 120e6
Čípky — tři typy (r, g, b) — vnímání barev, soustředěny ve slepé skvrně, v oku cca 8e6

Barevné systémy

- RGB
- CMY(K) = 1 — RGB
- HSV (barvy na šestibokém jehlanu), HLS (barvy na dvou kuželech)
  - převod není prosté zobrazení, používá se algoritmus. viz Pelikánovy slidy

- YC_bC_r — používán při kódování JPEG
  - $Y = 0.3 * R + 0.6 * G + 0.1 * B$
  - $C_b = 0.56 * (B - Y)$
  - $C_r = 0.71 * (R - Y)$
- různé barevné palety s různými počty barev

Konstrukce adaptované palety

Paleta se může adaptovat pro daný obrázek (např. v GIFu). Metody:

- shora dolů
  - množinu použitých barev dělím tak dlouho, dokud nedostanu požadovaný počet
- zdola nahoru
  - sdružuji příbuzné barvy
- metoda shlukové analýzy
  - využívá se histogram
- Heckbertova metoda
  - opět histogram
  - dělí se nejdelší vážená hrana kvádru v místě těžiště

Reprodukce barevné grafiky

Pokud někdo tušíte, co spadá pod tuhle otázku, prosím doplňte.

28.7 Rozptylování a půltónování

Napodobení vjemu neexistujících barevných odstínů pro dané zařízení. Zvyšuji barevné rozlišení na úkor prostorového.

- Rozptylování
  - zobrazuje se bez zvětšení rozlišení (1:1)
- Půltónování
  - zobrazuje se se zvětšeným rastrovým rozlišením (1:N)

Typické příklady:

- černobílé tiskárny
- displaye s malou paletou barev
- obrázky s omezenou paletou

Půltónování

Jeden zdrojový pixel (rozsah hodnot 0-N^2) nakreslím jako čtverec NxN

- Inkrementální rastry
  - odstín hodnoty k obsahuje právě k černých teček
  - lze uložit do matice
- Pravidelný rastr
- Tečkový rastr
- Nutno předcházet pravidelnosti a aliasu — rastry se často otáčejí

Rozptylování

- Maticové rozptylování
  - libovolná půltónovací matice
  - nejčastěji matice pravidelného rastru
  - několik sousedních pixelů sdílí jednu matici
  - ztráta drobných detailů
  - zvýraznění vedlejších odstínů
- Náhodné rozptylování
  - pro lidské oko mnohem přirozenější
  - černobílé obrázky příliš zašumělé — lepší výsledky u více výstupních odstínů
- Distribuce chyby
  - zaokrouhlím hodnotu pixelu na nejbližší zobrazitelnou
  - rozdíl distribuuji mezi sousední pixely
    - různé způsoby distribuce
    - je třeba pomocný buffer
  - výhody
    - příjemné pro oko
    - dobrá kvalita
  - nevýhody
    - nutnost kreslit po řádkách
    - nemožnost vrátit ze zpátky
    - pomalé

28.8 Kompozice poloprůhledných obrázků

28.9 Geometrické deformace rastrových obrázků

28.10 Morphing

28.11 Základní principy komprese rastrové 2D grafiky

28.12 Skalární a vektorové kvantování

28.13 Prediktivní komprese

28.14 Transformační kompresní metody

28.15 Hierarchické a progresivní metody

28.16 Waveletové transformace a jejich celočíselné implementace

Jak funguje JPEG 2000

28.17 Kódování koeficientů

28.18 Komprese videosignálu, časová predikce – kompenzace pohybu

28.19 Standardy JPEG a MPEG

MPEG-1; MPEG-2; MPEG-3; MPEG-4;
Chroma subsampling
Macroblock

28.20 Snímání obrazu v digitální fotografii

28.21 Materiály

- Počítačová grafika I – stránka předmětu
- Pokročilá 2D počítačová grafika – stránka předmětu
- Speciální funkce a transformace ve zpracování obrazu – stránka předmětu

Kapitola 29

Realistická syntéza obrazu, virtuální realita

29.1 Rozsah látky

Seznam oficiálních státnicových otázek:

Metody reprezentace 3D scén, klasické zobrazovací algoritmy, výpočet viditelnosti, výpočet vržených stínů, modely osvětlení a stínovací algoritmy, rekurzivní sledování paprsku, textury, anti-aliasing, urychlovací metody pro ray-tracing, princip radiačních metod, výpočet konfiguračních faktorů, řešení radiační soustavy rovnic, hierarchické přístupy v radiačních metodách, fyzikální model šíření světla — zobrazovací rovnice, Monte-Carlo přístupy ve výpočtu osvětlení, hybridní zobrazovací metody, přímé metody ve vizualizaci objemových dat, generování izoploch, schéma grafického akcelerátoru, předávání dat do GPU, textury v GPU, programování GPU, základy jazyka Cg, pokročilé techniky práce s GPU, SW a HW prostředky pro virtuální realitu, vlastnosti jazyka VRML, struktura scény, typy uzlů (datové typy, trikové uzly), tvorba statické scény VRML, dynamické a interaktivní scény VRML, práce se skripty, rozhraní EAI, víceuživatelská virtuální realita.

29.34 Víceuživatelská virtuální realita

29.35 Materiály

- Počítačová grafika I – stránka předmětu
- Počítačová grafika II – stránka předmětu
- Hardware pro počítačovou grafiku – stránka předmětu
- Virtuální realita – stránka předmětu

Příloha A

Document Information & History

A.1 History

This book was created on the Wikibooks project and developed on the project by the contributors listed in Appendix ??, page ??. For convenience, this PDF was created for download from the project. The latest Wikibooks version may be found at `http://wiki.matfyz.cz.org/wiki/`.

A.2 PDF Information & History

This PDF was complied from LaTeX on 22. srpna 2011, based on the 22 August 2011 Wikibooks textbook. The latest version of the PDF may be found at `http://en.wikibooks.org/wiki/Image:.pdf`.

A.3 Authors

Andree, Che, Dmajda, Feci, Hardwire, Ivokabel, Johny, Kvido, Luky, Michalisek, Premil, Rajjo, Ramirez, Realyze, Segabond, Srakyi, Stevko, Tuetschek, Ty-Dyt, Valera, Wlk, and anonymous contributors.