1 Lineární závislost a nezávislost, báze, dimenze. Lineární zobrazení, jádro a obor hodnot, skalární a vektorový součin. (A0B01LAG)

1.1 Lineární prostor

Neprázdná množina $\mathcal{L}$ se nazývá lineární vektorový prostor nad tělesem $\mathbb{R}$, jestliže je splněno následujících deset podmínek.

1. Pro každé dva prvky $u, v \in \mathcal{L}$ je jednoznačně určen prvek $u + v \in \mathcal{L}$ nazývaný součet prvků u a v.
2. Pro každý prvek $u \in \mathcal{L}$ a pro každý prvek $\lambda \in \mathbb{R}$ je jednoznačně určen prvek $\lambda u \in \mathcal{L}$ nazývaný násobek prvku u prvkem λ
3. $u + v = v + u$ pro každé dva prvky $u, v \in \mathcal{L}$ (komutativita)
4. $(u + v) + w = u + (v + w)$ pro každé tři prvky $u, v, w \in \mathcal{L}$ (asociativita)
5. Existuje prvek $0 \in \mathcal{L}$. takový, že pro každý prvek $u \in \mathcal{L}$ platí $u + 0 = 0 + u = u$
6. Pro každý prvek $u \in \mathcal{L}$ existuje prvek $-u \in \mathcal{L}$ takový, že $u + (-u) = (-u) + u = 0$
7. $\lambda(u + v) = \lambda u + \lambda v$ pro každé dva prvky $u, v \in \mathcal{L}$ a pro každý prvek $\lambda \in \mathbb{R}$.
8. $(\lambda + \alpha)u = \lambda u + \alpha u$ pro každý prvek $u \in \mathcal{L}$ a pro každé dva prvky $\alpha, \lambda \in \mathbb{R}$
9. $(\lambda\alpha)u = \lambda(\alpha u)$ pro každý prvek $u \in \mathcal{L}$ a pro každé dva prvky $\alpha, \lambda \in \mathbb{R}$
10. $1u = u$ pro každý prvek $u \in \mathcal{L}$

1.2 Lineární podprostor

Neprázdná podmnožina W vektorového prostoru V nad tělesem T se nazývá podprostorem V, pokud pro libovolné vektory $u, v \in W$ a libovolný skalár $\lambda \in T$ platí:

- $a + b \in W$
- $\lambda a \in W$

Množina W je tedy uzavřená vzhledem k operacím sčítání vektorů a násobení vektoru skalárem.

1.3 Lineární kombinace

Nechť $\mathcal{L}$ je lineární prostor, $v_1, v_2, ..., v_n \in \mathcal{L}$ a $\lambda_1, \lambda_2, ..., \lambda_n \in \mathbb{R}$. Prvek $\lambda_1 v_1 + \lambda_2 v_2 + ... + \lambda_n v_n \in \mathcal{L}$ se nazývá lineární kombinace prvků $v_1, v_2, ..., v_n$ s koeficienty $\lambda_1, \lambda_2, ..., \lambda_n$.

- Lineární kombinace $\lambda_1 v_1 + \lambda_2 v_2 + ... + \lambda_n v_n$ se nazývá **netriviální**, pokud existuje $i \in \{1, 2, ..., n\}$ takové, že $\lambda_i \neq 0$
- Lineární kombinace $\lambda_1 v_1 + \lambda_2 v_2 + ... + \lambda_n v_n$ se nazývá **triviální**, jestliže $\lambda_i = 0$ pro každé $i = 1, 2, ..., n$

1.4 Lineární závislost a nezávislost

Prvky $v_1, v_2, ..., v_n$ množiny M se nazývají **lineárně závislé**, pokud existuje taková **netriviální lineární kombinace** těchto prvků, která vyhovuje vztahu

$$\sum_{i=1}^{n} a_i v_i = 0$$

kde a_i je skalár. V opačném případě jsou lineárně nezávislé.

- Pro **lineárně nezávislé** prvky je jediným řešením výše uvedeného vzorce triviální řešení, tedy $a_i = 0$
- Jsou-li prvky **lineárně závislé**, je možné nějaký z nich vyjádřit jako lineární kombinaci ostatních prvků

1.4.1 Příklad

Lineárně zavislá množina$M$ a koeficienty netriviální lineární kombinace a

$$M = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 4 & 0 \\ 3 & 6 & 1 \end{bmatrix}, a = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}$$

$$M \bullet a = 0$$

1.5 Báze

Lineární obal Mějme množinu M, která je podmnožinou vektorového prostoru V. Průnik všech podprostorů prostoru V, které obsahují množinu M se nazývá lineárním obalem množiny M.
Zjednodušeně - lineární obal množiny M je podprostor prostoru V. Co obsahuje? Všechny ty prvky, ke kterým se mohu dostat libovolnou lineární kombinací vektorů z množiny M.

$$\langle M \rangle = \left\{ \sum_{i=1}^{n} a_i u_i \mid u_i \in M, a_i \in \mathbb{R}, i = 1, 2, 3, ..., n \right\}$$

Báze vektorového prostoru V je nejmenší množina **lineárně nezávislých vektorů** taková, že její lineární obal je roven celému prostoru V. V konečně dimenzionálním prostoru dimenze n je bází každá množina obsahující n lineárně nezávislých vektorů.

- Obal báze prostoru V tvoří celý prostor V
- Vektory báze jsou **lineárně nezávislé**.
- Prostor může mít **více bází**. Všechny ale mají **stejný počet prvků**.

1.5.1 Příklad

$$B_1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, B_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Matice B_1 i B_2 tvoří bázi prostoru $\mathbb{R}^2$.

1.6 Lineární zobrazení

Pojmem **lineární zobrazení** (lineární transformace) se v matematice označuje takové zobrazení mezi vektorovými prostory U a V, které zachovává vektorové operace sčítání a násobení skalárem. Název lineární je odvozen z faktu, že grafem obecného lineárního zobrazení z reálných čísel do reálných čísel je přímka.

$$L(u + v) = L(u) + L(v)\ u \in U, v \in V$$
$$L(\alpha u) = \alpha L(u)\ u \in U$$

1.6.1 Matice linárního zobrazení

Nechť U a V jsou lineární vektorové prostory konečné dimenze nad tělesem R, $L : U \to V$ je lineární zobrazení. Mějme $u_1, u_2, ..., u_k$ bázi prostoru U, $dimU = k$ a $v_1, v_2, ..., v_n$ bázi prostoru V, $dimV = n$. Pro libovolný prvek $x \in U$ lze psát $x = \lambda_1 u_1 + \lambda_2 u_2 + ... + \lambda_k u_k$, tedy $\hat{x} = [\lambda_1, \lambda_2, ..., \lambda_k]^T$ je vektor koeficientů prvku x v bázi $u_1, u_2, ..., u_k$ prostoru

U. Zobrazením prvku x získáme prvek $y = L(x)$, který lze opět vyjádřit jako lineární kombinaci bázových vektorů $y = \eta_1 v_1 + \eta_2 v_2 + ... + \eta_k v_n$, tedy $\hat{y} = [\eta_1, \eta_2, ..., \eta_n]^T$.

Zobrazení L je lineární, proto platí

$$L(x) = L(\lambda_1 u_1 + \lambda_2 u_2 + ... + \lambda_k u_k) = \lambda_1 L(u_1) + \lambda_2 L(u_2) + ... + \lambda_k L(u_k)$$

$$L(x) = y = [v_1, v_2, ..., v_n] \cdot \hat{y}$$

$$L(u_i) = [v_1, v_2, ..., v_n] \cdot [\alpha_{1i}, \alpha_{2i}, ..., \alpha_{ni}]^T$$

$$[v_1, v_2, ..., v_n] \cdot \hat{y} = \lambda_1 [v_1, v_2, ..., v_n] \cdot [\alpha_{11}, \alpha_{21}, ..., \alpha_{n1}]^T + ... + \lambda_k [v_{1k}, v_{2k}, ..., v_{nk}] \cdot [\alpha_{1k}, \alpha_{2k}, ..., \alpha_{nk}]^T$$

po zkrácení

$$\hat{y} = \lambda_1 \cdot [\alpha_{11}, \alpha_{21}, ..., \alpha_{n1}]^T + ... + \lambda_k \cdot [\alpha_{1k}, \alpha_{2k}, ..., \alpha_{nk}]^T$$

$$\hat{y} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1k} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{k1} & \alpha_{k2} & \cdots & \alpha_{nk} \end{bmatrix} \cdot \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_k \end{bmatrix} = \mathbf{A}\hat{x}$$

$$\mathbf{A} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1k} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{k1} & \alpha_{k2} & \cdots & \alpha_{nk} \end{bmatrix}$$

kde **A** je **matice lineárního zobrazení** L v bázích $u_1, u_2, ..., u_k$ prostoru U a $v_1, v_2, ..., v_n$ prostoru V.

Vyjádřeno méně formálně: každý vektor z U si můžeme vyjádřit jako kombinaci bázových vektorů U. Pak se podíváme na koeficienty, kterými násobíme tyto bázové vektory, a chceme z nich dostat koeficienty bázových vektorů v prostoru V. Pokud těmito koeficienty vynásobíme bázové vektory V, dostaneme lineární zobrazení původního vektoru do prostoru V. Díky matici lineárního zobrazení můžeme tyto koeficienty získat.

1.7 Jádro a obor hodnot

Mějme lineárního zobrazení $L : U \to V$, které je vyjádřeno jako $\{A \bullet x = y \mid x \in U, y \in V, A \in \mathbb{R}^{\mathsf{m} \times \mathsf{n}}\}$. Množinu tvořenou všemi řešeními $A \bullet x = 0$ nazýváme **jádro** lineárního zobrazení L, nebo-li **nulový prostor** matice A.

$$Ker(L) = null(A) = \{x \in U \mid A \bullet x = 0\}$$

Obor hodnot zobrazení L (obraz matice A) je podprostor, který obsahuje zobrazení všech prvků z prostoru U.

$$Im(L) = rng(A) = \{A \bullet x \mid x \in U\}$$

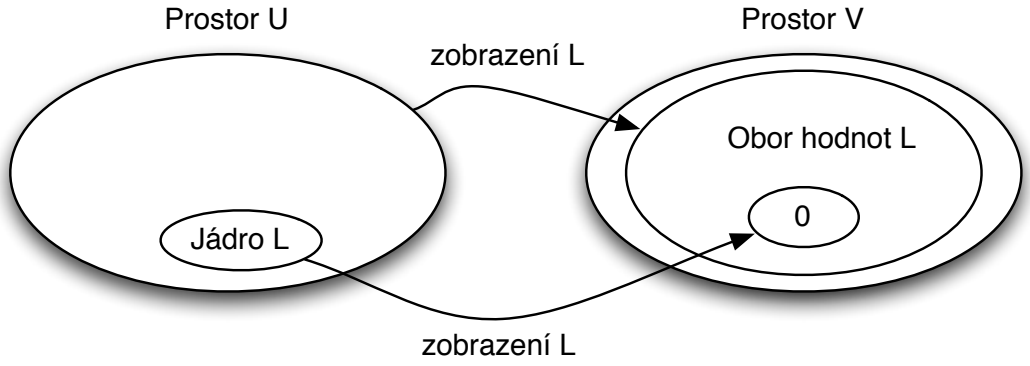


Figure 1.1: Vztah jádra a obrazu prostoru

- **Jádro** zobrazení L je podmnožinou prostoru U, ale **obor hodnot** zobrazení L je podmnožinou V
- Platí vztah $dim(Ker(L))+dim(Im(L)) = dim(U)$ jinak $dim(null(A))+dim(rng(A)) = n$ kde n je šířka matice A.

1.8 Skalární a vektorový součin

Skalární součin definujeme mezi dvěma vektory. Výsledkem skalárního součinu je reálné číslo, není to vektor. Máme-li dva vektory $u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ a $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$, pak jejich skalární součin je roven:

$$u^T \bullet v = \mid u \mid\mid v \mid \cos\alpha$$

kde α je velikost úhlu mezi vektory u a v.

Vektorový součin je binární operace vektorů v trojrozměrném vektorovém prostoru. Výsledkem této operace je vektor, který je kolmý k oběma původním vektorům. Velikost tohoto vektoru je rovna obsahu rovnoběžníku tvořeného původními vektory. Spočítá se

$$u \times v = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} = w$$

a platí

$$u^T \bullet w = 0,\ v^T \bullet w = 0$$

1 Matice, determinant, inverzní matice, vlastní čísla a vlastní vektory matice. Soustavy lineárních rovnic. (A0B01LAG)

1.1 Matice

Matice je obdélníkové či čtvercové schéma čísel nebo nějakých matematických objektů - prvků matice (též elementů matice). Obsahuje obecně m řádků a n sloupců. Hovoříme pak o matici typu $m \times n$

Zápis matice

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = (a_{ij})_{m,n}$$

1.2 Determinant

V lineární algebře je determinant zobrazení, které přiřadí každé **čtvercové** matici $\mathbf{A}$ skalár $det(\mathbf{A})$.

1.2.1 Výpočet

$$det(\mathbf{A}) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

vysvětlivky

- S_n- množina všech permutací čísel $1, 2, ..., n$, kde n je šířka (i výška) matice A; pro $n = 3$ tedy $S_n = \{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$
- $sgn()$ - funkce vracející $+1$ nebo -1 (znaménko permutace). To se odvíjí od počtu prohození dvou sousední prvků v permutaci, abychom dostali základní řadu $[1, 2, ..., n]$. Pokud σ je sudá permutace, vrací funkce $+1$, naopak pokud je lichá, vrací -1. Např. pro $[2, 1, 3]$ musíme prohodit jen 2 s 1, abychom dostali [1,2,3], jedná se tedy o lichou permutaci a funkce $sgn()$ vrací -1. U permutace $[3, 2, 1]$ je prohození více: $[3, 2, 1] \rightarrow [3, 1, 2] \rightarrow [1, 3, 2] \rightarrow [1, 2, 3]$, celkem 3 prohození, proto funkce $sgn()$ vrací -1. Jedná se o totéž, čemu profesor Pták říkal „leftdown".

- $a_{i,\sigma(i)}$ takto nastavené indexy zaručí, že do součinu vybereme vždy právě jeden prvek z každého sloupce a řádku

1.2.2 Vlastnosti

- Při výměně dvou řádků nebo dvou sloupců se znaménko determinantu změní na opačné
- Z předchozí vlastnosti plyne, že pokud má matice $\mathbf{A}$ dva stejné řádky nebo dva stejné sloupce, tak musí platit $det(\mathbf{A}) = -det(\mathbf{A}) = 0$
- Hodnota determinantu se nezmění, zaměníme-li řádky za sloupce $det(\mathbf{A}) = det(\mathbf{A}^T)$
- Jestliže jeden řádek (sloupec) lze vyjádřit jako lineární kombinaci ostatních řádků (sloupců), je determinant nulový.
- Matice se značí pojmem **singulární**, když $det(\mathbf{A}) = 0$. Pokud $det(\mathbf{A}) \neq 0$, pak se jedná o matici **regulární.**

1.3 Inverzní matice

Inverzní matice k dané matici je taková matice, která po vynásobení s původní maticí dá jednotkovou matici.

$$\mathbf{A} \bullet \mathbf{A^{-1}} = \mathbf{A^{-1}} \bullet \mathbf{A} = 1$$

Obě rovnosti znamenají, že inverzní matice může existovat jen pro čtvercovou matici. U obdélníkové matice mluvíme o tzn. pseudoinverzi.

1.3.1 Výpočet

$$a_{i,j} = \frac{(-1)^{i+j} \mid \mathbf{A}_{j,i} \mid}{\mid \mathbf{A} \mid}$$

kde $\mid \mathbf{A}_{j,i} \mid$ je subdeterminant získaný z matice $\mathbf{A}$ vynecháním j-tého řádku a i-tého sloupce, $\mid \mathbf{A} \mid$ je determinant matice $\mathbf{A}$.

1.4 Vlastní čísla a vlastní vektory matice

Jako **vlastní vektor** dané transformace označujeme nenulový vektor, jehož směr se při transformaci nemění. Koeficient, o který se změní velikost vektoru, se nazývá **vlastní číslo** (hodnota) daného vektoru. Vyjádřeno vzorcem

$$\mathbf{Av} = \lambda \mathbf{v}$$

kde $\mathbf{A}$ je matice transformace, $\mathbf{v}$ je vlastní vektor, který je na obou stranách stejný, λ je vlastní číslo, nebo-li onen koeficient, kterým je třeba vlastní vektor vynásobit. Definice totiž říká, že se při transformaci nemění jen směr, velikost se ale lišit může.

Rovnici můžeme upravit

$$(\mathbf{A} + \lambda\mathbf{E})\mathbf{v} = 0$$

kde $\mathbf{E}$ je jednotková matice. Aby platila rovnice a my dostali netriviální řešení $\mathbf{v} \neq 0$, musí platit

$$det(\mathbf{A} + \lambda\mathbf{E}) = 0$$

$$\begin{vmatrix} a_{11} - \lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda \end{vmatrix} = 0$$

Při výpočtu tohoto determinantu dostaneme na levé straně polynom. Ten se nazývá charakteristický polynom matice $\mathbf{A}$ a jeho kořeny jsou vlastní čísla matice $\mathbf{A}$. Výsledkem je tedy vždy n vlastních čísel, z nichž některá se mohou opakovat.

1.5 Soustavy lineárních rovnic

Soustava m lineárních rovnic s n proměnnými může být zapsána ve tvaru $\mathbf{Ax} = \mathbf{b}$, kde $\mathbf{A} \in \mathbb{R}^{\mathsf{m}\times\mathsf{n}}, \mathbf{x} \in \mathbb{R}^{\mathsf{n}}, \mathbf{b} \in \mathbb{R}^{\mathsf{m}}$. Pokud je $\mathbf{b}$ nulový vektor, mluvíme o homogenní soustavě lineárních rovnic.

Zajímá nás, jestli má soustava řešení, případně kolik. Může nastat jedna z těchto situací

- soustava nemá **žádné** řešení
- soustava má **jedno** řešení
- soustava má **nekonečně** mnoho řešení

1.5.1 Frobeniova věta

Nehomogenní soustava lineárních algebraických rovnic má řešení pouze v případě, že hodnost matice soustavy $h(\mathbf{A})$ je rovna hodnosti rozšířené matice soustavy $h([\mathbf{A}\ \mathbf{b}])$.

Dejme tomu, že máme matici $\mathbf{A} \in \mathbb{R}^{\mathsf{4}\times\mathsf{3}}$, tedy 3 neznámé, ale 4 rovnice. Je zřejmé, že taková matice může mít hodnost maximálně $h(\mathbf{A}) = 3$. Řekněme, že má skutečně hodnost 3, potom se ptáme jakou hodnost má rozšířená matice $[\mathbf{A}\ \mathbf{b}] \in \mathbb{R}^{\mathsf{4}\times\mathsf{4}}$. Ta už může mít hodnost 4, a pokud ji skutečně má, potom soustava nemá řešení, protože nalevo je jedna rovnice lineární kombinací zbývajících, ale napravo má řešení, které není odpovídající lineární kombinací pravých stran. Naopak, pokud $[\mathbf{A}\ \mathbf{b}]$ má stále hodnost 3, jenom se nám potvrdilo, že jedna rovnice je v soustavě zbytečná, můžeme ji vynechat a rovnice má řešení.

1.5.2 Počet řešení

Soustava lineárních rovnic má právě jedno řešení, když $h(\mathbf{A})$ je rovno počtu neznámých; pokud je $h(\mathbf{A})$ menší než počet neznámých, je řešení nekonečně mnoho.

3 Vlastnosti celých čísel (dělitelnost, prvočísla) a Eukleidův algoritmus. Binární relace, zejména ekvivalence a uspořádání, a jejich reprezentace. Počítání modulo. (A4B01DMA)

3.1 Vlastnosti celých čísel

- celá čísla Z se skládají z přirozených čísel, nuly a záporných celých čísel
- množina je uzavřena na operaci sčítání, odčítání a násobení

3.1.1 Dělitelnost

- **Definice:** Nechť a, b∈ Z. Řekneme, že a dělí b, značeno a|b, jestliže existuje k ∈ Z takové, že b = k · a. V takovém případě říkáme, že a je faktor b a že b je násobek a. Také říkáme, že b je dělitelné číslem a. Pokud toto není pravda, tak píšeme a∤b.
- Číslo d∈N je **společný dělitel** (common divisor) čísel a, b, jestliže d|a a d|b.
- **největší společný dělitel** (greatest common divisor), značeno gcd(a, b) je největší prvek množiny jejich společných dělitelů, pokud je alespoň jedno z a,b nenulové.
- Číslo d∈N je **společný násobek** (common multiple) čísel a, b, jestliže a|d a b|d.
- **nejmenší společný násobek** (least common multiple), značeno lcm(a, b) je nejmenší prvek množiny jejich společných násobků, pokud jsou obě a,b nenulové.
- lcm(a, 0) = lcm(0, b) = 0
- gcd(0, 0) = 0
- **lcm(a, b) · gcd(a, b) = |a| · |b|**
- čísla a, b ∈ Z jsou **nesoudělná**, jestliže gcd(a, b) = 1

3.1.2 Prvočíslo

- je přirozené číslo, které je beze zbytku dělitelné **právě dvěma různými přirozenými čísly**, a to číslem **jedna** a **sebou samým** (tedy 1 není prvočíslo)
- Přirozená čísla různá od jedné, která nejsou prvočísla, se nazývají **složená čísla**.

3.1.3 Eukleidův algoritmus

Lze jím vypočítat **největšího společného dělitele** dvou přirozených čísel.

- **vychází z lemmatu**: Nechť a, b ∈ N, nechť q, r∈N_0 splňují a = qb + r a 0 ≤ r < b. Pak platí následující: d ∈ N je společný dělitel a, b právě tehdy, když je to společný dělitel b, r.
- gcd(a, b) = gcd(b, r)
- opakovaně hledáme gcd pro dvojici b, r místo a, b

3.1.3.1 příklad: Chceme najít gcd(408, 108)

Máme 408 = 3 · 108 + 84 (408 mod 108 = 84), proto gcd(408, 108) = gcd(108, 84).
Máme 108 = 1 · 84 + 24, proto gcd(408, 108) = gcd(108, 84) = gcd(84, 24).
Máme 84 = 3 · 24 + 12, proto gcd(408, 108) = gcd(108, 84) = gcd(84, 24) = gcd(24, 12).
Máme 24 = 2 · 12 + 0, proto gcd(408, 108) = gcd(108, 84) = gcd(84, 24) = gcd(24, 12) = gcd(12, 0) = 12.

3.2 Binární relace

Definice: Nechť A,B jsou množiny. Libovolná podmnožina R ⊆ A × B se nazývá relace z A do B. Jestliže (a, b)∈ R, pak to značíme aRb a řekneme, že a je v relaci s b vzhledem k R. Jestliže (a, b) /∈ R, pak řekneme, že a není v relaci s b vzhledem k R.

Druhy relací:

- R je reflexivní, jestliže pro všechna a ∈ A platí aRa. např. "je stejný"
- R je symetrická, jestliže pro všechna a, b ∈ A platí (aRb ⇒ bRa). "je sourozencem"
- R je antisymetrická, jestliže pro všechna a, b ∈ A platí ([aRb ⋀ bRa] ⇒ a = b).
- R je tranzitivní, jestliže pro všechna a, b, c ∈ A platí ([aRb ⋀ bRc] ⇒ aRc). "je vyšší; A je vyšší než B, B je vyšší než C ⇒ A je vyšší než C"

3.2.1 Ekvivalence

Definice: Nechť R je relace na nějaké množině A. Řekneme, že R je ekvivalence, jestliže je **reflexivní, symetrická a tranzitivní**.

3.2.1.1 Třída ekvivalence

Každá ekvivalence rozdělí množinu A na systém disjunktních množin, které pak nazýváme třídy ekvivalence.

Definice: Nechť R je relace ekvivalence na nějaké množině A. Pro a∈A definujeme třídu ekvivalence prvku a (equivalence class of a) vzhledem k R jako $[a]_R = \{b \in A; aRb\}$.

Příklad: Mějme ekvivalenci R na množině celých číslech Z definovanou takto: [a, b] ∈ R právě tehdy, když |a| = |b|. Pak:

Z[0] = {0}. Nula je v relaci pouze s nulou.

Z[1] = {−1, 1}. Jednička je v relaci s jedničkou a s minus jedničkou, protože |1| = |−1|.

Z[2] = {−2, 2}. Dvojka je v relaci s dvojkou a s minus dvojkou.

Z[3] = {−3, 3}. ...

3.2.2 Částečné uspořádání

Definice: Nechť R je relace na nějaké množině A. Řekneme, že R je částečné uspořádání, jestliže je **reflexivní, antisymetrická a tranzitivní**. V tom případě řekneme, že dvojice (A, R) je částečně uspořádaná množina.

Příklad: Relace ≤ je uspořádání na přirozených, celých, racionálních i reálných číslech.

Relace ⊆ je uspořádání na třídě všech množin (na univerzální třídě).

Relace dělitelnosti | (a dělí b) je uspořádáním na přirozených číslech

Relace "Být potomkem" je uspořádáním na množině osob.

3.2.2.1 Hasseův diagram

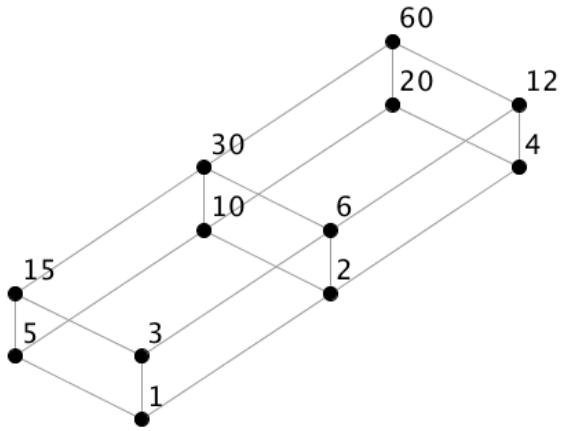
- Uspořádané množiny můžeme zakreslit pomocí Hasseova diagramu.
- vrcholy představují prvky množiny
- hrana mezi vrcholy (a, b) nám říká, že a < b a zároveň neexistuje c takové, že a < c < b. Tedy mezi prvky a a b už žádný jiný prvek není. Přitom musí platit, že v grafu je vrchol a níže než vrchol b.

Příklad: Dělitelé čísla 60: A = {1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60}. Uspořádání podle dělitelnosti:

3.3 Počítání modulo

Definice Nechť n∈N. Řekneme, že čísla a, b∈Z jsou **kongruentní modulo** n, značeno a ≡ b (mod n), jestliže n|(a−b).

Nechť n ∈ N. Pro čísla a, b ∈ Z jsou následující podmínky ekvivalentní:



- a ≡ b (mod n)
- existuje k ∈ Z takové, že a = b + kn
- mod n = b mod n, tj. jsou si rovny zbytky po dělení číslem n.

3.3.1 vlastnosti

Nechť n ∈ N, uvažujme a, b, u, v ∈ Z takové, že a ≡ u (mod n) a b ≡ v (mod n):

- a + b ≡ u + v (mod n)
- a − b ≡ u − v (mod n)
- ab ≡ uv (mod n)

4 Kombinatorika (kombinatorická čísla, princip inkluze a exkluze); Využití matematické indukce; rekurzivní vztahy (řešení rovnic, odhad náročnosti algoritmů) (A4B01DMA)

4.1 Kombinatorika

4.1.1 Kombinační číslo

Definice
Nechť $k \leq n \in \mathbb{N}_0$. Definujeme jejich **kombinační číslo** nebo **binomický koeficient** jako

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Čteme to „n nad k“.

Úprava

Výraz z definice je nepraktický, protože faktoriály jsou velice drahé na výpočet. Proto bývá lepší nejprve zkrátit jeden z faktoriálů ze jmenovatele se začátkem faktoriálu v čitateli. Máme pak

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdots (n-k+2) \cdot (n-k+1)}{k!} = \frac{n \cdot (n-1) \cdots (k+2) \cdot (k+1)}{(n-k)!}.$$

4.1.2 Kombinatorické případy

Uvažujme množinu o n různých prvcích.
(i) Je $n!$ způsobů, jak je seřadit (neboli je $n!$ permutací).
(ii) Jestliže na pořadí záleží a opakování není povoleno, pak je $\frac{n!}{(n-k)!} = \binom{n}{k} \cdot k!$ různých způsobů, jak vybrat k prvků z této množiny.
(iii) Jestliže na pořadí záleží a opakování je povoleno, pak je n^k různých způsobů, jak vybrat k prvků z této množiny.
(iv) Jestliže na pořadí nezáleží a opakování není povoleno, pak je $\binom{n}{k}$ různých způsobů, jak vybrat k prvků z této množiny.
(v) Jestliže na pořadí nezáleží a opakování je povoleno, pak je $\binom{n+k-1}{k}$ různých způsobů, jak vybrat k prvků z této množiny.

4.1.2.1 shrnutí

	bez opakování	s opakováním
s pořadím (variace)	$\frac{n!}{(n-k)!}$	n^k
bez pořadí (kombinace)	$\binom{n}{k}$	$\binom{n+k-1}{k}$

4.1.2.2 příklady

Variace s opakováním Kolik je možno vytvořit osmimístných hesel (password) skládajících se z písmen a číslic? Každý znak je nezávislý jev, který je možno udělat 26 + 10 = 36 způsoby, proto je možno vytvořit 36^8 hesel.

Permutace Kolik permutací písmen ABCDEFGH obsahuje slovo DECH? Toto se udělá jednoduchým trikem, prostě se DECH vezme jako jeden celek, který se spolu s ostatními čtyřmi písmenky permutuje, takže celkem permutujeme pět věcí. Možností je tedy 5! = 120.

Kombinace Uvažujme binární řetězce o délce 8. Kolik z nich obsahuje přesně tři jedničky? Zde vybíráme, na které pozice jedničky dáme, a na pořadí výběru nezáleží (říct, že jedničky mají být na pozicích 1, 2 a 6, vyjde nastejno jako říct, že mají být na pozicích 2, 6 a 1). Takže vybíráme z osmi míst, bez opakování a bez pořadí, tedy $\binom{8}{3} = 56$ řetězců.

Kombinace s opakováním Kolik různých balíčků bonbónů (ty jsou tam volně ložené) je možné vytvořit, když do balíčku vybíráme 10 bonbónů ze tří druhů, přičemž od každého druhu je k dispozici dostatek kusů? Vybíráme desetkrát z tříprvkové množiny, výběr můžeme opakovat a na pořadí nezáleží, protože bonbóny se pak stejně budou v balíčku volně míchat. Je to ta nejobtížnější ze čtyř základních situací, proto si vzorec pamatujeme: Je možné udělat $\binom{3+10-1}{10} = 66$ různých balíčků.

4.1.3 Princip inkluze a exkluze

Jsou-li A_i pro $i = 1, 2, \ldots, n$ konečné množiny, pak

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{n-1} \left|\bigcap_{i=1}^{n} A_i\right|$$
$$= \sum_{k=1}^{n} (-1)^{k-1} \sum_{1 \le i_1 < i_2 < \cdots < i_k \le n} |A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}|.$$

pro dvě množiny tedy: $|A \cup B| = |A| + |B| - |A \cap B|$

4.1.4 Binomická věta

4.1.4.1 Pascalova identita

Pro všechna $k \leq n \in \mathbb{N}_0$ platí

$$\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}.$$

grafické znázornění - Pascalův trojúhelník

$$\begin{array}{ccccccccccc}
 & & & & \binom{0}{0} & & & & & & \\
 & & & \binom{1}{0} & & \binom{1}{1} & & & & & \\
 & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & \\
 & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & \\
\binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & & \\
\binom{5}{0} & & \binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \binom{5}{4} & & \binom{5}{5}
\end{array}
\qquad
\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & \\
 & & & 1 & & 1 & & & & & \\
 & & 1 & & 2 & & 1 & & & & \\
 & 1 & & 3 & & 3 & & 1 & & & \\
1 & & 4 & & 6 & & 4 & & 1 & & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

4.1.4.2 Binomický rozvoj

Pro každé $n \in \mathbb{N}$ a všechna $x, y \in \mathbb{R}$ platí

$$\begin{aligned}(x+y)^n &= \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k \\ &= x^n + nx^{n-1}y + \frac{n(n-1)}{2}x^{n-2}y^2 + \cdots + \frac{n(n-1)}{2}x^2y^{n-2} + nxy^{n-1} + y^n.\end{aligned}$$

4.2 Matematická indukce

= metoda dokazování matematických vět a tvrzení

4.2.1 Slabý princip matematické indukce

Nechť $n_0 \in \mathbb{Z}$, nechť $V(n)$ je vlastnost celých čísel, která má smysl pro $n \geq n_0$.
Předpokládejme, že jsou splněny následující předpoklady:
(0) $V(n_0)$ platí.
(1) Pro každé $n \in \mathbb{Z}$, $n \geq n_0$ je pravdivá následující implikace: Jestliže platí $V(n)$, pak platí i $V(n+1)$.
Potom $V(n)$ platí pro všechna $n \in \mathbb{Z}$, $n \geq n_0$.

Pokud tedy chceme dokázat univerzální platnost nějaké vlastnosti V, stačí dokázat pravdivost tvrzení (0) a (1).

4.2.1.1 Demonstrace na příkladu žebříku

Obecně Tzv. základní krok (0) říká, že umíme vylézt na první příčku žebříku. Tzv. indukční krok (1) říká, že když už někde jsme, tak umíme vylézt o příčku výš. Podstatný je ten obecný kvantifikátor v (1), indukční krok je splněn pro libovolné místo na žebříku.

Matematicky Vezmeme pro jednoduchost n0 = 1. Podle základního kroku platí V (1). Indukční krok dává pro volbu n = 1 pravdivou implikaci V (1) ⇒ V (2), my už ovšem ze základního kroku víme, že V (1) platí, tudíž podle této implikace platí i V (2). Pak zase můžeme použít indukční krok s n = 2, kde z pravdivosti V (2) dostaneme pravdivost V (3). Další použití indukčního kroku (s n = 3) dá pravdivost V (4), pak V (5) a tak dále.

4.2.1.2 Příklad

Dokažte indukcí: Pro n∈N je V (n) tvrzení, že $1 + 3 + 5 + \cdots + (2n - 1) = n^2$.

(0) Nechť n = 1. Vlastnost V (1) zní 1 = 1, což je pravda.

(1) Nechť n∈N je libovolné. Předpokládejme, že $1 + 3 + 5 + \cdots + (2n - 1) = n^2$ pro naše konkrétní n platí také $1+3+5+\cdots+(2(n+1)-1) = (n+1)^2$, tedy že $1 + 3 + 5 + \cdots + (2n + 1) = (n + 1)^2$

$1 + 3 + 5 + \cdots + (2n + 1) = 1 + 3 + 5 + \cdots + (2n - 1) + (2n + 1) = [1 + 3 + 5 + \cdots + (2n - 1)] + (2n + 1) = n^2 + (2n + 1) = (n + 1)^2$.

4.3 Rekurzivní vztahy

Definice: Rekurentní vztah či rekurzivní vztah pro posloupnost $\{a_k\}$ je libovolná rovnice typu $F(a_n, a_{n-1}, a_{n-2}, \ldots, a_0) = 0$, kde F je nějaká funkce.

Např. podstata problému Hanojských věží se dá vyjádřit vztahem $H_n - 2H_{n-1} - 1 = 0$

4.3.1 Lineární rekurentní rovnice

Lineární rekurentní rovnice, popřípadě **lineární rekursivní rovnice řádu** $k \in \mathbb{N}_0$ je libovolná rovnice ve tvaru

$$a_{n+k} + c_{k-1}(n)a_{n+k-1} + \cdots + c_2(n)a_{n+2} + c_1(n)a_{n+1} + c_0(n)a_n = b_n \quad \text{pro všechna } n \geq n_0,$$

kde $n_0 \in \mathbb{Z}$, $c_i(n)$ pro $i = \{0, \ldots, k-1\}$ (tzv. **koeficienty** rovnice) jsou nějaké funkce $\mathbb{Z} \mapsto \mathbb{R}$, přičemž $c_0(n)$ není identicky nulová funkce, a $\{b_n\}_{n=n_0}^{\infty}$ (tzv. **pravá strana rovnice**) je pevně zvolená posloupnost reálných čísel.

Jestliže $b_n = 0$ pro všechna $n \geq n_0$, pak se příslušná rovnice nazývá **homogenní**.

Řešení

Nechť je dána lineární rekurentní rovnice

$$a_{n+k} + c_{k-1}(n)a_{n+k-1} + \ldots + c_1(n)a_{n+1} + c_0(n)a_n = b_n \quad \text{pro všechna } n \geq n_0.$$

Jako její **řešení** označíme libovolnou posloupnost $\{a_n\}_{n=n_0}^{\infty}$ takovou, že po dosazení odpovídajících členů do dané rovnice dostáváme pro všechna n pravdivý výrok.

4.3.2 Charakteristická rovnice

Definice

Nechť je dána lineární rekurentní rovnice s konstantními koeficienty

$$a_{n+k} + c_{k-1}a_{n+k-1} + \ldots + c_1a_{n+1} + c_0a_n = b_n \text{ pro všechna } n \geq n_0.$$

Její **charakteristický polynom** (**characteristic polynomial**) je definován jako polynom

$$p(\lambda) = \lambda^k + c_{k-1}\lambda^{k-1} + \ldots + c_1\lambda + c_0.$$

Kořeny charakteristického polynomu se nazývají **charakteristická čísla**, popřípadě **vlastní čísla** dané rovnice (**characteristic numbers/roots** or **eigenvalues**).

K získání charakteristických čísel potřebujeme vyřešit rovnici $\lambda^k + c^k{-}1\lambda^k{-}1 + \ldots + c_1\lambda + c_0 = 0$, které se také říká charakteristická rovnice

4.3.2.1 Příklad

Najdeme obecné řešení rovnice $a_n + 3 - a_n + 2 - a_n + 1 + a_n = 0$pro všechna n ≥ -2
Charakteristický polynom je p(λ) =$\lambda_3 - \lambda_2 - \lambda + 1$
p(λ) = $(\lambda - 1)(\lambda - 1)(\lambda + 1) = (\lambda - 1)^2(\lambda + 1)$.
báze řešení:

$$\left\{\{1^n\}_{n=-2}^{\infty}, \{n1^n\}_{n=-2}^{\infty}, \{(-1)^n\}_{n=-2}^{\infty}\right\}$$

obecné řešení pro pro u, v, w ∈ R:
$\{u \cdot 1^n + v \cdot n1^n + w \cdot (-1)^n\}_{n=-2}^{\infty} = \{u + vn + w(-1)^n\}_{n=-2}^{\infty}$

z takového řešení lze odhadnout asymptotickou složitost alogritmu

4.3.3 Master theorem

- = algoritmus pro určování asymptotické složitosti algoritmů
- zejména "rozděl a panuj" $f(n) = a \cdot f(\frac{n}{b}) + g(n)$.

Algoritmus

(The Master theorem)
Předpokládejme, že neklesající nezáporná funkce f na $\mathbb{N}$ splňuje rovnici $f(n) = a \cdot f(\frac{n}{b}) + cn^d$ na množině $M = \{b^k;\ k \in \mathbb{N}\}$, kde $b \in \mathbb{N}$ splňuje $b \geq 2$ a $a, c \in \mathbb{R}$, $d \in \mathbb{N}_0$ jsou konstanty splňující $a \geq 1$ a $c > 0$. Pak platí následující:
(i) Jestliže $a > b^d$, tak $f(n) = \Theta(n^{\log_b(a)})$.
(ii) Jestliže $a = b^d$, tak $f(n) = \Theta(n^d \log_2(n))$.
(iii) Jestliže $a < b^d$, tak $f(n) = \Theta(n^d)$.

Společná část - otázka č. 5

Imperativní programování, software, překladač, interpret, vnitřní forma, programovací jazyky, syntaxe, sémantika, proměnné, výrazy, vstup, výstup, řídící struktury, jednoduché datové typy, přiřazení, funkce, procedury, parametry, rozklad problému na podproblémy, princip rekurze a iterace (A0B36PR1)

June 2, 2012

1 Základní pojmy

1.1 Informatika

Informatika:
- zabývá se zpracováním informací nejen na počítačích
- studuje výpočetní a informační procesy z hlediska hardware i software – „technická informatika“
- je součástí teorie informací, věda spojující aplikovanou matematiku a elektrotechniku za účelem kvantitativního vyjádření informace

1.2 Software

V informatice sada všech počítačových programů v počítači. Software zahrnuje:
– operační systém (zajišťuje běh programů)
– aplikační software (pracuje s ním uživatel)
– další (knihovny, middleware, BIOS, firmware apod.)

1.2.1 Knihovna

Knihovna je v informatice označení pro soubor funkcí a procedur (v objektovém programování též objektů, datových typů a zdrojů), který může být sdílen více počítačovými programy. Knihovna usnadňuje programátorovi tvorbu zdrojového kódu tím, že umožňuje použít již vytvořený kód i v jiných programech. Knihovna navenek poskytuje své služby pomocí API (aplikační rozhraní), což jsou názvy funkcí (včetně popisu jejich činnosti), předávané parametry a návratové hodnoty. Knihovny lze rozdělit podle vazby na program, který je používá, na statické a dynamické.

1.2.1.1 Statické knihovny

Statická knihovna tvoří s přeloženým programem kompaktní celek. V závěrečné fázi překladu programu ze zdrojového kódu do strojového kódu jsou volané funkce (a další jimi kaskádově volané funkce) připojovány linkerem (odtud označení linkování) do výsledného spustitelného souboru. V horším případě je k programu připojena celá knihovna bez ohledu na to, jaké části program skutečně využívá. Výsledný spustitelný soubor tak v sobě obsahuje všechny části, které jsou nutné pro jeho běh. Staticky slinkovaný spustitelný soubor proto typicky při spuštění nepotřebuje žádné další soubory, takže je ho možné překopírovat na jiný systém a jednoduše spustit (tj. bez instalace, tak, jak je známá například pro běžné programy v Microsoft Windows). Př. .lib, .a.

1.2.1.2 Dynamické knihovny

Dynamické knihovny nejsou (na rozdíl od statických knihoven) k výslednému spustitelnému souboru přidávány. V závěrečné fázi překladu programu ze zdrojového kódu do strojového kódu jsou odkazy na volané knihovní funkce pomocí linkeru (odtud označení linkování) zapsány do speciální tabulky symbolů, která je připojena k výslednému spustitelného souboru. Pro chod programu (tj. spuštění výsledného spustitelného souboru) je pak nutné mít k dispozici též příslušné dynamické knihovny. Pokud dynamická knihovna využívá ke své činnosti jiné dynamické knihovny, vzniká řetězec závislostí a všechny potřebné knihovny musí být při spuštění programu přítomny. Př. .dll, .o, .so.

1.3 Hardware

Zahrnuje všechny fyzické součásti počítače

- čistě elektronická zařízení (procesor, paměť, display)
- elektromechanické díly (klávesnice, tiskárna, diskety, disky, jednotky CD-ROM, páskové jednotky, reproduktory) pro vstup, výstup a ukládání dat.
- Počítač se skládá z procesoru, operační paměti a vstupně-výstupních zařízení.

1.4 Algoritmus

Postup při řešení určité třídy úloh, který je tvořen seznamem jednoznačně definovaných příkazů a zaručuje, že pro každou přípustnou kombinaci vstupních dat se po provedení konečného počtu kroků dospěje k požadovaným výsledkům.

- **hromadnost** - měnitelná vstupní data
- **determinovanost** - každý krok je jednoznačně definován
- **konečnost a resultativnost** - pro přípustná vstupní data se po provedení konečného počtu kroků dojde k požadovaným výsledkům

1.5 Program

Program je předpis (zápis algoritmu) pro provedení určitých akcí počítačem zapsaný v programovacím jazyku.

1.6 Programovací jazyk

Programovací jazyk je prostředek pro zápis algoritmů, jež mohou být provedeny na počítači.

1. **strojově orientované**

- **strojový jazyk** = jazyk fyzického procesoru

- **asembler** = jazyk symbolických adres

1. **vyšší jazyky**

- **imperativní** (příkazové, procedurální)
- **neimperativní** (např. funkcionální)

1.6.1 Imperativní programování

Hlavní rysy imperativních jazyků (např. C, C++, Java, Pascal, Basic, ...)
- zpracovávané údaje mají formu datových objektů různých typů, které jsou v programu reprezentovány pomocí proměnných resp. konstant
- program obsahuje deklarace a příkazy
- deklarace definují význam jmen (identifikátorů)
- příkazy předepisují akce s datovými objekty nebo způsob řízení výpočtu

1.7 Syntaxe

Souhrn pravidel udávajících přípustné tvary dílčích konstrukcí a celého programu.

1.8 Sémantika

Udává význam jednotlivých konstrukcí.

1.9 Interpret

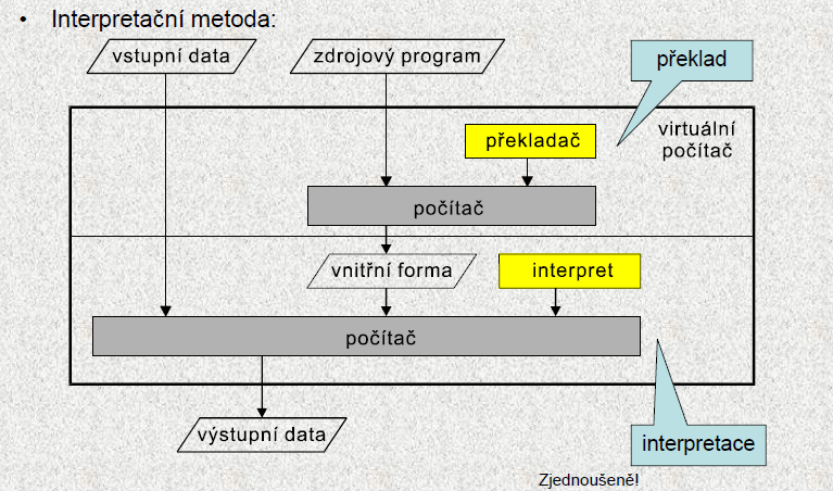
Interpret je v informatice speciální počítačový program, který umožňuje **přímo vykonávat (interpretovat)** zápis jiného programu v jeho zdrojovém kódu ve zvoleném programovacím jazyce. Program proto není nutné převádět do strojového kódu cílového procesoru, jako je tomu v případě překladače. Interpret tak umožňuje programování kódu, který je snadno přenositelný mezi různými počítačovými platformami.

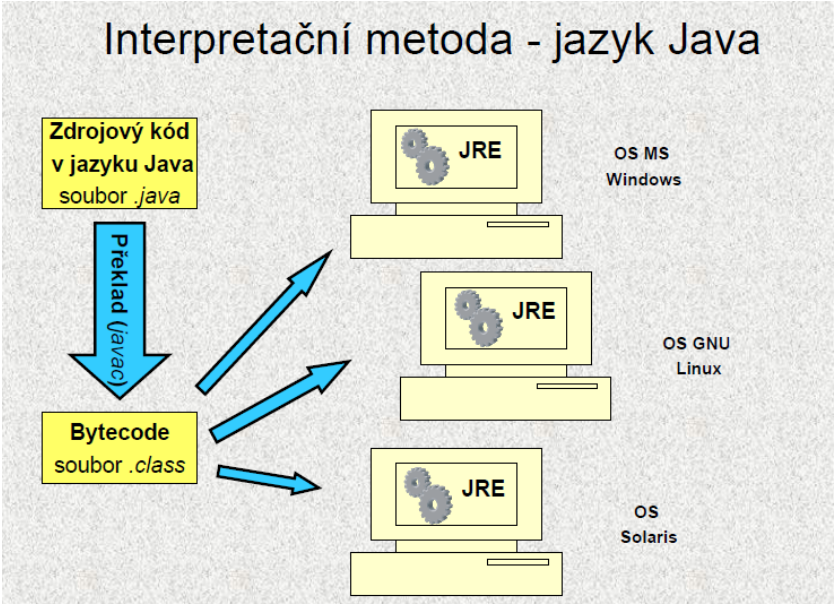
1. provádějí přímo zdrojový kód (unixový shell, COMMAND.COM nebo interprety jazyka BASIC)

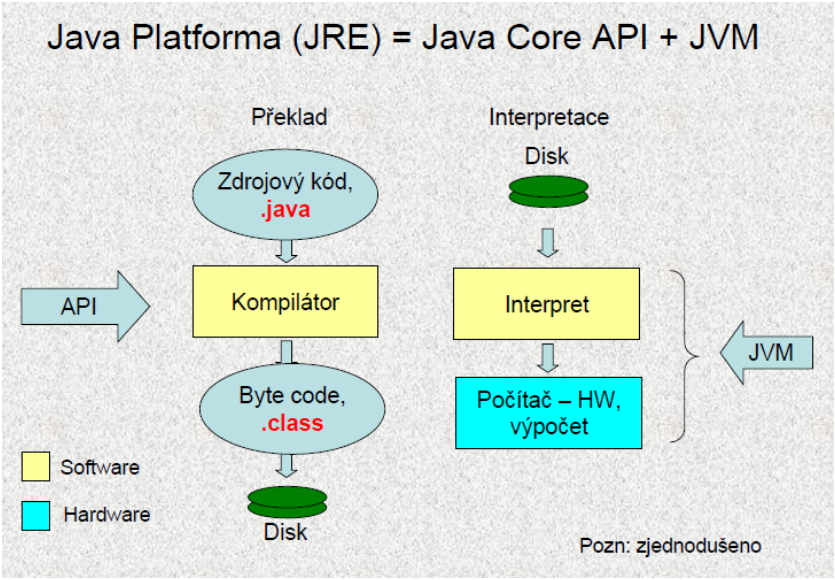
2. přeloží zdrojový kód do efektivnějšího mezikódu, který následně spustí (Perl, Python nebo MATLAB)

3. přímo spustí předem vytvořený předkompilovaný mezikód, který je produktem části interpretu (UCSD Pascal a Java - zdrojové kódy jsou kompilovány předem, uloženy ve strojově nezávislém tvaru, který je po spuštění linkován a interpretován nebo kompilován v případě použití JIT).

- Interpretační metoda:





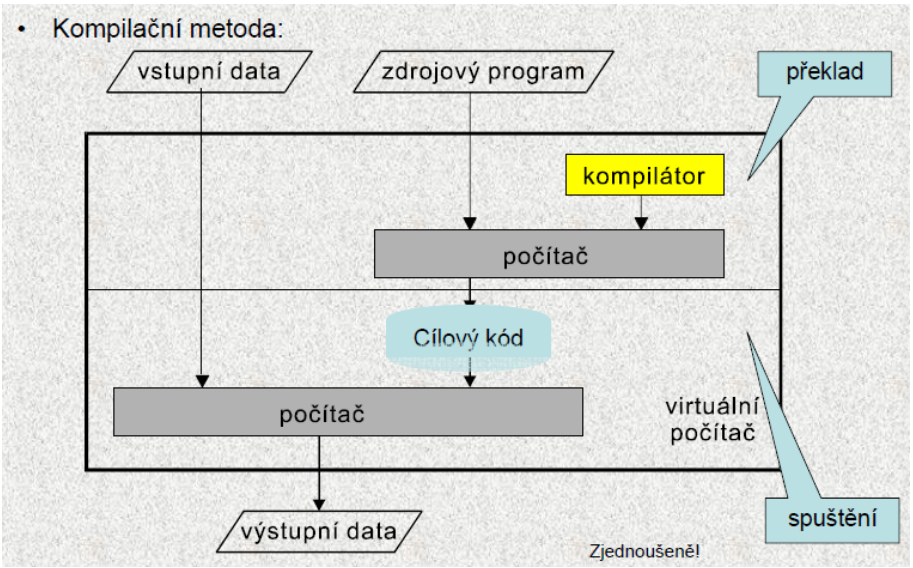


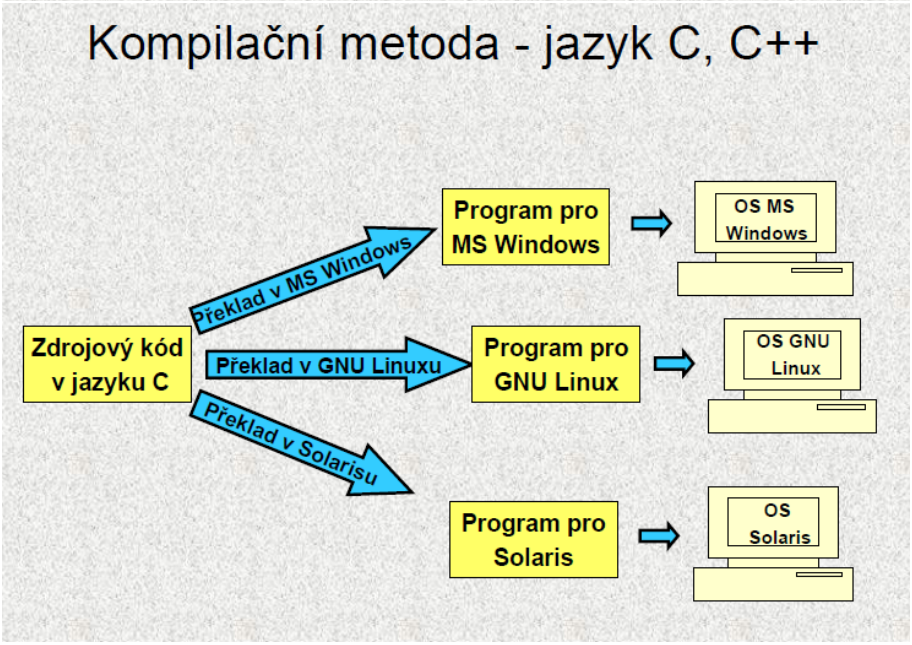
1.9.1 Vnitřní forma

Při interpretaci se zdrojový program přeloží do vnitřní formy. Ta není strojovým jazykem fyzického procesoru, ale je jazykem virtuálního počítače. Provedení programu ve vnitřní formě na konkrétním počítači zajistí interpretr.

1.10 Kompilátor = překladač

Překladač (též kompilátor) je v nejčastějším smyslu slova softwarový nástroj používaný programátory pro vývoj softwaru. Kompilátor slouží pro překlad algoritmů zapsaných ve vyšším programovacím jazyce do jazyka strojového, či spíše do strojového kódu. Z širšího obecného hlediska je kompilátor stroj, respektive program, provádějící překlad z nějakého vstupního jazyka do jazyka výstupního.


• Kompilační metoda:
vstupní data
zdrojový program
překlad
kompilátor
počítač
Cílový kód
počítač
virtuální počítač
výstupní data
spuštění
Zjednodušeně!


Kompilační metoda - jazyk C, C++
Zdrojový kód v jazyku C
Překlad v MS Windows
Překlad v GNU Linuxu
Překlad v Solarisu
Program pro MS Windows
Program pro GNU Linux
Program pro Solaris
OS MS Windows
OS GNU Linux
OS Solaris

2 Části programovacích jazyků

2.1 Proměnná

V imperativním programování je proměnná „úložiště“ informace (tedy vyhrazené místo v paměti - v některých jazycích se ovšem v průběhu výpočtu může místo, kde je proměnná uložena, měnit). Proměnná nebo (v beztypových jazycích) její hodnota má typ.

Proměnná je datový objekt, který je označen jménem a je v něm uložena hodnota nějakého typu, která se může měnit.

2.1.1 Přiřazení

Pro přiřazení hodnoty proměnné slouží přikazovací příkaz. Má tvar <proměnná> = <výraz>;

V jazyku Java zavedeme proměnnou deklarací int promenna = 0;

Přiřazovací příkaz: promenna = 37;

Konstantě nelze přiřadit hodnota: MAX = 32 -> CHYBA!!

Jazyk JAVA má **silnou typovou kontrolu**, následující nelze: boolean b; b = 1; -> CHYBA!!

2.1.1.1 Přiřazovací operátory

- <proměnná> = <proměnná> <OP> <výraz> e.g. x = x +1
- <proměnná> <OP>= <výraz> x += 1

2.1.2 Přidělení paměti proměnným

Přidělením paměti proměnné rozumíme určení adresy umístění proměnné v paměti počítače.

2.1.3 Typy proměnných

- lokální proměnné funkcí - pamět přidělena při volání funkce, po jejím skončení uvolněna, pamět na zásobníku
- statické proměnné tříd - pamět přidělena při zavedení kódu třídy do paměti (JVM), až do konce programu

2.2 Primitivní datové typy

Datový typ (zkráceně jen typ) specifikuje:

• **množinu hodnot** - pro int: -2147483648 .. 2147483647

• **množinu operací,** které lze s hodnotami daného typu provádět (pro int je to například: relační operace ==, !=, >, >=, <, <=, jejichž výsledkem je hodnota typu boolean)

V jazyce JAVA je třeba deklarovat, s jakým typem dat budeme pracovat. Překladač tuto deklaraci hlídá. V důsledku reprezentace dat v počítači vznikají nepřesnosti (1.00 + 2.00 se nemusí rovnat 3.00). Čísla jsou aproximována. Čím větší exponent (tzn. čím dále od nuly), tím je větší mezera mezi aproximacemi. Kolem nuly je čísel nejvíce. Reprezentace dat v počítači (int, double apod.) bude podrobněji rozebrána pravděpodobně v nějaké otázce z APO.

2.2.1 JAVA - Primitivní datové typy

1. **byte:** Datový typ byte obsahuje 8bitové dvakrát doplněné celé číslo se znaménkem. Jeho minimální hodnota je −128 a maximální 127. Datový typ byte může být použit pro ušetření paměti ve velkých polích, kde se může ve speciálních případech hodit každý bit. Taktéž může být náhradou za typ int, kde mohou jeho limity pomoct objasnit váš kód; takto je hodnota limitována víc než údajem v dokumentaci.

2. **short:** Datový typ short je 16bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −32 768 a maximální 32 767. Stejně jako u typu byte jsou zde stejné případy použití: typ short můžete použít k ušetření paměti při vytváření velkých polí u náročných aplikací.

3. **int:** Datový typ int je 32bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −2 147 483 648 a maximální 2 147 483 647 (2^31). Pro celočíselné hodnoty je tento typ tou základní volbou, pokud se nedostanete do výše popsaných situací. Tento datový typ bude dostatečně velký pro většinu čísel ve vaší aplikaci, pokud však budete potřebovat širší rozsah hodnot, použijte místo něj typ long.

4. **long:** Datový typ long je 64bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −9 223 372 036 854 775 808 a maximální 9 223 372 036 854 775 807 (2^63). Používejte tento datový typ, pokud vám nebude stačit rozsah hodnot poskytovaný datovým typem int.

5. **float:** Datový typ floatje 32bitové IEEE 754 číslo s pohyblivou desetinou čárkou. Stejně jako u typu byte a short používejte typ float (místo double), pokud potřebujete ušetřit paměť ve velkých polích. Tento datový typ byste nikdy neměli používat pro přesné hodnoty, jako jsou finanční částky. Pro tento případ budete muset použít třídu java.math.BigDecimal.

6. **double:** Datový typ double je 64bitové IEEE 754 číslo s pohyblivou desetinou čárkou. Pro desetinná čísla je tento datový typ tou základní volbou. Stejně jako typ float se double nehodí pro přesné hodnoty, jako jsou finanční částky.

7. **boolean:** Datový typ boolean má povoleny pouze 2 hodnoty: true a false. Používejte tento datový typ, abyste specifikovali, zda je podmínka pravdivá (true) nebo nepravdivá (false). Tento datový typ obsahuje 1 bit informací, ale jeho velikost není přesně definována.

8. **char:** Datový typ char je jednoduchý 16bitový Unicode znak (proto jsou odstraněny problémy s různými jazyky). Jeho minimálni hodnou je u0000 (nebo 0) a maximální uffff (nebo 65535). Konverze znaků probíhá v JAVA většinou automaticky podle nastavení jazyka OS.

Primitivní či základní datové typy

Typ	Bitů	Rozsah		Obal.třída
Celočíselný typ				
byte	8	-128 ... 127		Byte
short	16	-32768 ... 32767		Short
int	32	-2147483648 ... 2147483647		Integer
long	64	-9223372036854775808 ... 9223372036854775807		Long
Reálný typ, IEEE 754 (NaN, infinity)				
float	32	2^{-149} ... $(2-2^{-23})*2^{127}$		Float
double	64	2^{-1074} ... $(2-2^{-52})*2^{1023}$		Double
Znaky, UCS2				
char	16	'\u0000' to '\uffff' 0 ... 65535		Character
Logický typ				
boolean	1/8	true false		Boolean
Pomocný prázdný typ				
void				

byte +-*/ byte ... int
short +-*/ short ... int
int +-*/ int ... int (pozor na přetečení)
long +-*/ long ... long

2.2.2 Typové konverze

Typová konverze je operace, která hodnotu nějakého typu převede na hodnotu jiného typu.

- implicitní - např. int na double (když se očekává hodnota double, ale je tam INT, dojde k implicitní konverzi) - implicitní konverze je bezpečná

- explicitiní - programátor musí explicitně označit, je potenciálně nebezpečná (může dojít ke ztrátě informace) - eg. double -> int, double d = 1E30; int i = (int)d; // i je 2147483647 asi 2E10

2.3 Výrazy

Výraz se skládá z operandů a operátorů. **Operandem** může být konstanta, proměnná, volání metody nebo opět výraz. **Operátory** udávají, co se má provést s jednotlivými hodnotami operandů.

Operátory a jejich priorita

priorita	operátor	typ operandu	asociativita	operace
1	++	aritmetický	P	pre/post inkremetace
	--	aritmetický	P	pre/post dekrementace
	-	aritmetický	P	unarní plus/minus
	~	celočíselný	P	bitová inverze
	!	logický	P	logická negace
	(typ)	libovolný	P	přetypování
2	*, / ,%	aritmetický	L	násobení, dělení, zbytek
3	-	aritmetický	L	odečítání
	+	aritmetický, řetězový	L	sčítání, zřetězení
4	<<	celočíselný	L	posun vlevo
	>>	celočíselný	L	posun vpravo
	>>>	celočíselný	L	posun vpravo s doplňováním nuly
	<,>, <=, >=	aritmetický	L	porovnání
	instanceof	objekt	L	test třídy
	==, !=	primitivní	L	rovno, nerovno
7	&	celočíselný nebo logický	L	bitové nebo logické AND
8	^	-"-	L	bitové nebo logické XOR
9	|	-"-	L	bitové nebo logické OR
10	&&	logický	L	logické AND vyhodnocované zkráceně
11	||	logický	L	logické OR vyhodnocované zkráceně
12	? :	logický	P	podmíněný operátor
13	=, -=, *=, /=, %=, ==, &=, ^=, |=	libovolný	P	přiřazení

Pozor na asociativitu. Odčítání je asociativní zleva, mocnění zprava.
Přikazovací příkaz může být výraze: y = x = x + 6, tj. y = (x = (x + 6));
Operace && a || se vyhodnocují zkráceným způsobem, druhý operand se nevyhod-

nocuje, když lze výsledek určit již z prvního operandu.

2.4 Výstup

Pro výpis dat na obrazovku se v Javě používá příkaz System.out.println(parametr); = výpis na standardní výstup. Metoda je přetížená, jako její parametr lze zadat všechny primitivní datové typy i String.

2.4.1 Formátovaný výstup

Př. System.out.printf("Cislo Pi = %6.3f %n ", Math.PI);
Specifikace formátu %[$indexParametru][modifikátor][šířka][.přesnost]konverze

- konverze - povinný parametr
- typ celé číslo d,o,x - dekadicky, oktalově a hexadecimálně
- typ double f je desetinný zápis; e,E vědecký s exponentem
- šířka - počet sázených míst, zarovnání vpravo
- .přesnost - počet desetinných míst
- modifikátor - v závislosti na typu konverze určuje další vlastnosti, například pro konverzi f (typ double) –> symbol + určuje, že má být vždy sázeno znaménko, –> symbol - určuje zarovnání vlevo, –> symbol 0 doplnění čísla zleva nulami.

Formátovaný výstup

```
package pr1_3;
public class FormatovanyVystup {
public static void main(String[] args) {
System.out.printf("Cislo Pi = %6.3f %n", Math.PI);
System.out.printf("Cislo Pi = %8.3f %n", Math.PI);
System.out.printf("Cislo Pi = %6.5e %n", Math.PI);
System.out.printf("Cislo Pi = %6.5g %n", Math.PI);
System.out.printf("Cislo Pi = %6d %n", 33);
System.out.printf("Cislo Pi = %4d %n", -33);
    }
}
```

```
Cislo Pi =  3,142
Cislo Pi =    3,142
Cislo Pi = 3.14159e+00
Cislo Pi = 3.1416
Cislo Pi =     33
Cislo Pi = -33
```

2.5 Vstup

Pro vstup dat zadaných na klávesnici poslouží třída Scanner. Je třeba vytvořit objekt třídy Scanner a napojit jej na standardní vstupní proud: Scanner sc = new Scanner(System.in);

- sc.nextInt() : přečte celé číslo z řádku zadaného klávesnicí (řádek je zakončen klávesou Enter, číslo je zakončeno mezerou nebo Enter) a vrátí je jako funkční hodnotu typu int
- sc.nextDouble() : přečte číslo z řádku zadaného klávesnicí a vrátí je jako funkční hodnotu typu double, jako oddělovač použijte . nebo čárku v závislosti na definovaném jazyku OS. Lze změnit pomocí před vytvořením Scanneru Locale.setDefault(Locale.ENGLISH);
- sc.nextLine() : přečte zbytek řádku zadaného klávesnicí a vrátí je jako funkční hodnotu typu String

2.6 Řídicí struktury

Řídicí struktura je programová konstrukce, která se skládá z dílčích příkazů a předepisuje pro ně způsob provedení. Tři druhy řídicích struktur:

1. 1. posloupnost, předepisující postupné provedení dílčích příkazů
2. 2. větvení, předepisující provedení dílčích příkazů v závislosti na splnění určité podmínky
3. 3. cyklus, předepisující opakované provedení dílčích příkazů v závislosti na splnění určité podmínky

2.6.1 Podmínka

Příkaz if (podmíněný příkaz) umožňuje větvení na základě podmínky.

- if (podmínka) příkaz1 else příkaz2
- if (podmínka) příkaz1

2.6.2 Cyklus

Základní příkaz cyklu, který má tvar: while (podmínka) příkaz (blok příkazů).

Příkaz cyklu do se od příkazu while liší v tom, že podmínka se testuje až za tělem cyklu. Tvar příkazu: do příkaz while (podmínka);

Poznámka k sémantice: příkaz do provede tělo cyklu alespoň jednou, nelze jej tedy použít v případě, kdy lze očekávat ani jedno provedení těla cyklu

Cyklus For: je často řízen proměnnou, pro kterou je stanoveno:

- jaká je počáteční hodnota

- jaká je koncová hodnota
- jak změnit hodnotu proměnné po každém provedení těla cyklu

2.6.2.1 Konečnost cyklů

Vstupní podmínku konečnosti cyklu lze určit téměř ke každému cyklu (někdy je to velmi obtížné, až nespočetné). Splnění vstupní podmínky konečnosti cyklu musí zajistit příkazy předcházející příkazu cyklu.

2.6.3 Switch

Příkaz switch (přepínač) umožňuje větvení do více větví na základě různých hodnot výrazu (nejčastěji typu int nebo char).

Sémantika (zjednodušeně):

- vypočte se hodnota výrazu a pak se provedou ty příkazy, které jsou označeny konstantou označující stejnou hodnotu
- není-li žádná větev označena hodnotou výrazu, provedou se příkazydef

2.7 Funkce

Funkce v programování je část programu, kterou je možné opakovaně volat z různých míst kódu. Funkce může mít argumenty (též parametry) – údaje, které jí jsou předávány při volání – a návratovou hodnotu, kterou naopak vrací.

Deklaraci funkce tvoří hlavička funkce a tělo funkce

• Hlavička funkce v jazyku Java má tvar static typ jméno(specifikace parametrů) kde

- typ je typ výsledku funkce (funkční hodnoty)
- jméno je identifikátor funkce
- specifikací parametrů se deklarují parametry funkce, každá deklarace má tvar typ_parametru jméno_parametru (a oddělují se čárkou)
- specifikace parametrů je prázdná, jde-li o funkci bez parametrů

• Tělo funkce je složený příkaz nebo blok, který se provede při volání funkce

• Tělo funkce musí dynamicky končit příkazem return x; kde x je výraz, jehož hodnota je výsledkem volání funkce

2.7.1 Parametry

Parametry funkce slouží pro předání vstupních dat algoritmu, který je funkcí realizován. Parametr může být výraz.

2.8 Procedura

Funkce, jejíž typ výsledku je void, nevrací žádnou hodnotu.

2.9 Rozklad problému na podproblémy

Postupný návrh programu rozkladem problému na podproblémy:

- zadaný problém rozložíme na podproblémy
- pro řešení podproblémů zavedeme abstraktní příkazy
- s pomocí abstraktních příkazů sestavíme hrubé řešení
- abstraktní příkazy realizujeme pomocí procedur (void)

2.10 Rekurze

V imperativním programování rekurze představuje opakované vnořené volání stejné funkce (podprogramu).

2.10.1 Rekurzivní algoritmus

Rekurzivní algoritmus předepisuje výpočet „shora dolů“ v závislosti na velikosti (složitosti) vstupních dat:

- pro nejmenší (nejjednodušší) data je výpočet předepsán přímo
- pro obecná data je výpočet předepsán s využitím téhož algoritmu pro menší (jednodušší) data

Výhoda: jednoduchost a přehlednost (to je diskutabilní).
Nevýhoda: Nevýhodou může být časová náročnost způsobená např. zbytečným opakováním výpočtu.

2.10.2 Rekurzivní funkce

Rekurzivní funkce (procedury) jsou přímou realizací rekurzivních algoritmů. Použití: faktoriál, fibonacciho posloupnost (začíná 1,1,2,3,5,8...). Fibonacci rekurzí (složitost exponenciální 2^n):

```
static int fib(int i) {
   if (i<2) return 1;
   return fib(i-1)+fib(i-2);
}
```

Fibonacci iteračně (složitost 3n):

```
static int fib(int n) {

    int i, fibNMinus2=1;
    int fibNMinus1=1, fibN=1;
    for (i=2; i<=n; i++) {
```

```
            fibNMinus2 = fibNMinus1;
            fibNMinus1 = fibN;
            fibN = fibNMinus1 + fibNMinus2;
        }

    return fibN;
    }
```

2.11 Iterace

Iterace v programování znamená opakované volání funkce v počítačovém programu. Zvláštní formou iterace je rekurze. Pro naše účely lze chápat iteraci jako opakované provádění bloku příkazu, typicky pomocí cyklu.

Společná část - otázka č. 6

Principy objektového přístupu, třída jako: programová jednotka, zdroj funkcí, datový typ; struktura objektu, konstruktory, přetěžování, instance třídy, hierarchie tříd, dědění, kompozice; abstraktní třídy, polymorfismus, rozhraní, rozhraní jako typ proměnné, typ interface.

June 14, 2012

1 Objektové programování

Objektově orientované programování (zkracováno na OOP) je metodika vývoje softwaru, založená na následujících myšlenkách, koncepci:

1. Objekty – jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty si pamatují svůj stav a navenek poskytují operace (přístupné jako metody pro volání).

2. Abstrakce – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

3. **Zapouzdření** – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem" jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

4. Skládání – Objekt může obsahovat jiné objekty.

5. Delegování – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

6. **Dědičnost** – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).

7. **Polymorfismus** – odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle implementace. U polymorfismu podmíněného dědičností to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy, neboť rozhraní třídy je podmnožinou rozhraní podtřídy. U polymorfismu nepodmíněného dědičností je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní.

Objektový přístup programování trochu jinak:

- Modelování problému jako systému spolupracujících tříd

- Třída modeluje jeden koncept
- Třídy umožní generování instancí, objektů příslušné třídy
- Jednotlivé objekty spolu spolupracují, „posílají si zprávy“,
- Třída je „vzorem“ pro strukturu a vlastnosti generovaných objektů
- Každý objekt je charakteristický specifickými hodnotami svých atributů a společnými vlastnostmi třídy

1.1 Třída

Třída je základem uživatelského programu v jazyku Java (nebo obecně v OOP). Třída = koncept. Každá třída je reprezentována svými prvky, objekty dané třídy. Každá třída je charakterizována svými vlastnostmi, svými funkčními možnostmi a svými parametry. Třída tedy popise nějaký “objekt” reálného problému a udržuje jeho parametry a poskytuje metody pro práci s ním. Třída v bodech:

- musí být deklarována hlavní funkce main (NOTE: bylo v přednášce, ale není to pravda, třídy bez třídy main se běžně používá, např. jako knihovna). Funkce main musí být statická, volá se ještě před vytvořením nějaké instance, slouží pro spuštění programu, testování.
- jsou deklarovány další (static) funkce (či procedury) třídy, případně i v jiných třídách: projekt, package
- mohou být deklarovány statické proměnné, které jsou použitelné jako nelokální proměnné ve funkcích dané třídy
- program probíhá spuštěním příkazů metody main

Třída je definována množinou identifikátorů, které mají třídou definovaný význam:
– data: proměnné, konstanty (členské proměnné, datové složky, atributy)
– metody: pracovně funkce a procedury

1.1.1 Třída = předpis vs. Objekt = instance třídy

Třída jako šablona pro generování konkrétních instancí třídy, tzv. objektů. Jednotlivé instance třídy (objekty) mají stejné metody, ale nacházejí se v různých stavech – Stav objektu je určen hodnotami instančních, členských proměnných. Schopnosti objektu jsou dány instančními metodami třídy. V jazyku Java lze objekty (instance tříd) vytvářet pouze dynamicky pomocí operátoru new a přistupovat k nim pomocí referenčních proměnných (podobně jako u pole). Třída bez vytvořené instance (objektu) může „pracovat“ pouze „staticky“ (mohou být použity jen její statické metody či proměnné – procedurální přístup). Statické proměnné = společné pro všechny instance třídy.

1.1.2 Konstruktor třídy

Konstruktory = speciální metody pro generování instancí tříd, konkrétních objektů.

- vytvoří objekt
- případně nastaví vlastnosti objektu
- jméno konstruktoru je totožné se jménem třídy (jediná metoda začínající velkým písmenem)
- volání pomocí operátoru new, např. malyObdelnik = new Obdelnik(2,5);
- neosahuje návratový typ - nic nevrací, vytváří objekt
- není-li konstruktor vytvořen, je vygenerován implicitní konstruktor s prázdným seznamem parametrů – je-li konstruktor deklarován, implicitní zaniká

1.1.2.1 Přetěžování (nejen konstruktoru třídy)

Umožňuje objektům volání jedné metody se stejným jménem, ale s jinou implementací. Provádí se to tak, že se deklaruje více metod se stejným názvem, které se mohou lišit různým počtem, typem argumentů popř. jejich pořadím. Umožňuje volat vytvoření instance s různými parametry - různým počtem i typem parametrů. Př.:

```
public class Obdelnik {


    // vsechny konstruktory lze pouzit zvenci pro vytvoreni objektu
    // kontruktor pro volani se vsemi tremi vlastnostmi,
    Obdelnik(int s, int v, Color c){
        sirka = s;
        vyska = v;
        barva = c;
    }
    // kontruktor pouze s rozmery - vyuziva prvni konstruktor pomoci this
    Obdelnik(int s, int v){
        this(s,v,Color.BLACK);
    }
    // prazdny konstruktor
    Obdelnik() {
        this(0,0,Color.BLACK);
    }

}
```

Konstruktor dále:

- Jméno konstruktoru je totožné se jménem třídy
- Konstruktor nemá návratovou hodnotu (ani void)
- Předčasně lze ukončit činnost konstruktoru return
- Konstruktor má parametrovou část jako metoda - může mít libovolný počet a typ parametrů
- V těle konstruktoru použít operátor this, odkaz na příslušný konstruktor s týmž počtem, pořadím a typem parametrů - nepíše se jméno třídy
- Konstruktor je zpravidla vždy public (!) – //třída java.lang.Math jej má private – proč? Protože je designována jako utilitní třída, která poskytuje statické metody (a atributy) a nevyžaduje vytvoření instance

1.1.3 Třída jako datový typ

Příkladem třídy jako datového typu je třída Complex

- hodnotami typu jsou komplexní čísla tvořená dvojicemi čísel typu double (reálná a imaginární část)
- množinu operací tvoří obvyklé operace nad komplexními čísly (absolutní hodnota, sčítání, odčítání, násobení a dělení)

1.1.4 Statické vs. Instanční metody

Třída může definovat dva druhy metod:

- statické metody – metody třídy, procedury a funkce
- instanční metody – metody objektů

Metody obou druhů mohou mít parametry a mohou vracet výsledek nějakého typu. Statická metoda označuje operaci (dílčí algoritmus, řešení dílčího podproblému), jejíž vyvolání (provedení) obsahuje jméno třídy, jméno metody a seznam skutečných parametrů jméno_třídy.jméno_metody(seznam skutečných parametrů). Statickým metodám třídy odpovídají v jiných jazycích procedury (nevracejí žádnou hodnotu) a funkce (vracejí hodnotu nějakého typu). Instanční metoda označuje operaci nad objektem (instancí (!)) dané třídy, jejíž vyvolání obsahuje referenční proměnnou objektu, jméno metody a seznam skutečných parametrů referenční_proměnná.jméno_metody(seznam skut. parametrů).

- Statickým metodám třídy budeme i nadále říkat procedury a funkce
- Instančním metodám budeme zkráceně říkat metody

1.1.5 Operátor this

Každý objekt má implicitní operátor this, který obsahuje odkaz na "svou" instanci

- Hodnotou operátoru this je odkaz na objekt pro který byla metoda zavolána (implicitní parametr odkazu na objekt)
- Umožňuje přístup k vlastním instančním proměnným v instančních metodách
- Používá se v přetížených konstruktorech

Podobně funguje operátor this i pro metody:

- pokud je instanční metoda volána z jiné instanční metody té samé třídy, potom se volá pomocí operátoru this (operátor se může vynechat)
- pokud se volá metoda z jiného kontextu, uvádí se před jejím jménem přístup k příslušné instanci (tečka notace) např. předané parametrem
- nelze volat ze statické metody - u ní by nebylo jasné, kam this odkazuje

1.2 Objekt

Objekt - datový prvek, instance třídy, dynamicky vytvořen podle „vzoru“- třídy (viz sekce Třída). Objekt si pamatuje svůj stav (v podobě dat čili atributů) a zveřejněním některých svých operací (nazývaných metody) poskytuje rozhraní, jak s ním pracovat. Objekt je strukturován tzn. skládá se z jednotlivých položek tzv. atributů. Objekt = heterogenní objekt skládající se z položek různého typu (x pole, pole se skládá z položek stejného typu). Složky objektu:

- atributy objektu (datové složky, členské proměnné (member variables)) - proměnné objektu, definují typ a jména vlastností objektu
- metody

Hodnota objektu je strukturovaná, tzn. skládá se dílčích hodnot, které mohou být obecně různého typu (heterogenní datová struktura – na rozdíl od pole). Objekt je tedy abstrakcí paměťového místa skládajícího se z částí, ve kterých jsou uloženy dílčí hodnoty - nazývají se položkami objektu (složkami, atributy, instančními proměnnými, fields, attributes). Položky objektu jsou označeny jmény, která mohou (ale nemusí) být třídou zveřejněna, zásadně se nezveřejňují. Pro manipulaci se zavádějí settery, gettery.

1.3 Zapouzdření

Zapouzdření v objektech znamená, že k obsahu objektu se nedostane nikdo jiný, než sám vlastník. Navenek se objekt projeví jen svým rozhraním (operacemi, metodami) a komunikačním protokolem. (Př. private proměnné -> pro manipulaci metody: nejjednodušší např. settery, gettery). Důležité pojmy:

- **Skládání objektů:** udržování odkazů na jiné objekty (objekt obsahuje jiné objekty).
- **Delegování:** Využívání služeb jiných objektů.

Zapoudření = Daný stav objektu je přístupný nebo měnitelný pouze prostřednictvím rozhraní poskytovaného objektem. Důsledkem zapouzdření je autorizovaný přístup k datům, při kterém zajistíme, že s daty objektu nebude možné z vnějšku třídy manipulovat jinak než pomocí metod této třídy, které tvoří komunikační rozhraní třídy. Zapouzdření je zajištěno pomocí modifikátorů:

1. public - lze je volat odkudkoli
2. protected - lze volat ze stejného package nebo odovozené třídy
3. neurčený - lze volat ze stejného package; nelze volat z odvozené třídy ležící v jiném package
4. private - lze je volat pouze z metod téže třídy

1.4 Dědičnost

Dědičnost je mechanismus umožňující

- rozšiřovat datové položky tříd (také modifikovat)
- rozšiřovat či modifikovat metody tříd

Dědičnost umožní

- vytvářet hierarchie tříd
- „předávat“ datové položky a metody k rozšíření a úpravě
- specializovat, „upřesňovat“ třídy

Dvě základní výhody dědění

- Dědění má praktický význam v znuvupoužitelnosti programového kódu
- Dědičnost je základem polymorfismu

Java dědí pouze od jediného předka. Mnohonásobné dědění se řeší pomocí rozhraní. Nejvyšší třída je Object, všechny třídy dědí třídu Object (metody: equals (standardní implementace neporovnává objekty, ale reference), toString, hashCode (stejné objekty by měly generovat stejný hashCode), clone..)

1.4.1 Příklad: Obdélník je případ kvádru (ne naopak)

Obdélník je „kvádrem“ s nulovou hloubkou

- Potomek se deklaruje pomocí klauzule extends
- Obdelnik převezme proměnné sirka, vyska, hloubka metody hodnotaSirky, delkaUhlopricky
- Konstruktor se dědí, parametr hloubka se nastaví do nuly
- Objekty Obdelnik mohou využívat proměnné sirka, vyska a hloubka, metody hodnotaSirky a delkaUhlopricky

```
public class Kvadr2 {

    public int sirka;
    public int vyska;
    public int hloubka;
    public Kvadr2(int sirka, int vyska, int hloubka) {
        this.hloubka = hloubka;
        this.sirka = sirka;
        this.vyska = vyska;
    }
    public int hodnotaSirky() {
        return sirka;
    }
    public double delkaUhlopricky() {

        double pom = (sirka * sirka) + (vyska*vyska) + (hloubka * hloubka);
        return Math.sqrt(pom);
    }

}
class Obdelnik2 extends Kvadr2{

    public Obdelnik2(int sirka, int vyska) {
        super(sirka, vyska, 0);
    }

}
```

1.4.2 Hiearchie tříd

Třída Tpod, která je podtřídou třídy Tnad, dědí vlastnosti nadtřídy Tnad a rozšiřuje je o nové vlastnosti; některé zděděné vlastnosti mohou být v podtřídě modifikovány. Pro instanční metody to znamená:

1. každá metoda třídy Tnad je i metodou třídy Tpod, v podtřídě však může mít jinou implementaci (může být zastíněna - override = Podtřída obsahuje metodu se stejným názvem i stejnými parametry, metody nadtřídy zastíní vlastní implementací. Př. typicky toString())
2. v podtřídě mohou být definovány nové metody

Pro strukturu objektu to znamená:

- instance třídy Tpod mají všechny členy třídy Tnad a případně další

Pro referenční proměnné to znamená:

- proměnné typu Tnad může být přiřazena reference na objekt typu Tpod
- na objekt referencovaný proměnnou typu Tnad lze vyvolat pouze metodu deklarovanou ve třídě Tnad; jde-li však o objekt typu Tpod, metoda se provede tak, jak je dáno třídou Tpod
- hodnotu referenční proměnné typu Tnad lze přiřadit referenční proměnné typu Tpod pouze s použitím přetypování, které zkontroluje, zda referencovaný objekt je typu Tpod

Vztah nadtřída – podtřída je tranzitivní = jestliže je x nad třídou y a y je nad třídou z, pak je x nadtřídou z

1.5 Kompozice

Obsahuje-li deklarace třídy členské proměnné objektového typu, pak mluvíme o kompozici objektů. Kompozice vytváří hierarchii objektů, nikoli však dědičnost. Například třída, která obsahuje jako členskou proměnnou integer, ale hlavně i například Datum (další, jiný objekt). (Kompozice označována jako struktura HAS ("má", dědičnost jako ISE "je"). Kompozice objektů je tvořena atributy objektového typu, pouze je skládá

1.6 Polymorfismus

V programovacím jazyce se jedná o možnost volat stejné metody u různých objektů, aniž bychom věděli, jakého přesně jsou typu. Navíc může mít stejná metoda u různých objektů odlišný význam. To je možné díky tomu, že vždy známe společného předka těchto různých objektů. Tím může být třída, abstraktní třída nebo rozhraní.

```
// soubor Osoba.java
public class Osoba {

    public int fce () {
        return 10;
```

```
    }
}
// soubor Zamestnanec.java
public class Zamestnanec extends Osoba {

    public int fce () {
        return 20;
    }

}
// jiny soubor
Zamestnanec a = new Zamestnanec();
Osoba b = new Zamestnanec();
Osoba c = new Osoba();
System.out.println(a.fce());    // vypíše 20
System.out.println(b.fce());    // vypíše 20
System.out.println(c.fce());    // vypíše 10
```

1.7 Abstraktní třídy

V některých situacích je výhodné vytvořit jedinou bázovou třídu pro více tříd odpovídajících konkrétním objektům, i když tato samotná bázová třída žádnému konkrétnímu objektu neodpovídá. Může ovšem nést některá data a poskytovat metody, které jsou odvozeným třídám společné. Taková třída se pak nazývá abstraktní a je označena klíčovým slovem abstract. Překladač jazyka Java pak zajistí, že instanci abstraktní třídy nelze operátorem new přímo vytvořit, mohou se vytvářet pouze instance konkrétních tříd.

Abstraktní třída může deklarovat některé společné metody a poskytovat jejich základní implementaci. Pokud odvozená třída takovou metodu nepředefinuje, pak se pro její instance použije implementace poskytnutá v bázové třídě.

Mohou však nastat i situace, kdy skutečně vyžadujeme, aby odvozené třídy určitou metodu vždy definovaly. Takovou metodu pak také nazýváme abstraktní a označujeme klíčovým slovem abstract, navíc u ní není uvedeno tělo a hlavička metody je zakončena středníkem. Pokud odvozená třída některou abstraktní metodu neimplementuje, musí být také označena jako abstraktní. Tím je zajištěno, že instance konkrétních tříd mají všechny metody implementované.

```
abstract class Obrazec {

    public Obrazec(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public abstract double obvod();
```

```
    public abstract double obsah();
    protected double x;
    protected double y;

}
```

1.8 Rozhraní

Druhou velmi podobnou konstrukcí jsou rozhraní (klíčové slovo interface). Základním rozdílem je, že interface obsahuje pouze konstanty a metody bez těla (v tomto případě je neoznačujeme slovem abstract). Rozhraní je kontrakt, který specifikuje operace, které má třída splňovat, a který se již nezabývá tím, jak toho bude konkrétně dosaženo.

Velkou výhodou rozhraní oproti abstraktním třídám je, že každá třída může implementovat až mnoho rozhraní, avšak vždy maximálně jednu třídu.

Zatímto pro dědění tříd (a dědení interfaců mezi sebou) využíváme v hlavičce klíčové slovo extends, tak pro implementaci rozhraní používáme slovo implements.

```
public interface Visualni {

    public void vykresli (Graphics kam);

}
```

1.8.1 Rozhraní jako typ proměnné

V Javě lze definovat proměnnou typu reference na rozhraní, ve které může být uložena libovolná třída, která toto rozhraní implementuje. Jména rozhraní lze používat jako referenční datové typy stejným způsobem jako jména tříd.

```
Visualni visualniObjekt = new VisualniKruznice();
visualniObjekt = new VisualniCtverec();
```

1.9 Výčtové typy - enum

Výčtové typy jsou speciální třídy zavedené pro větší bezpečí a pohodlí, v nejjednodušší variantě se definují (příklad):

```
enum Den {SUN, MON, TUE, WED, THU, FRI, SAT;}
for ( Den d : Den.values( ) )

    System.out.println( d.ordinal( )+ " " +d.name( ) );
```

Společná část - otázka č. 7

Spojové seznamy, lineární seznamy, obecné spojové struktury, stromy, jejich vlastnosti, binární stromy, implementace (A0B36PR1)

June 2, 2012

1 Pojmy

- Lineární spojová struktura (spojový seznam) - každý prvek má nanejvýš jednoho následníka
- Nelineární spojová struktura (strom) - každý prvek může mít více následníků
- Binární strom - každý prvek (uzel) má nanejvýš dva následníky

1.1 (Abstraktní) datový typ

Počet složek:

- neměnný = statický datový typ, počet položek je konstantní, pole, řetězec, třída
- proměnný = dynamický datový typ, počet složek je proměnný, mezi operace patří vložení, odebrání určitého prvku

Typ položek, dat:

- homogenní = všechny položky stejného typu
- nehomogenní = různého typu

Existence bezprostředního následníka

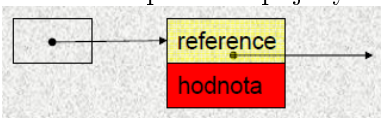
- lineární = existuje [např. pole, fronta, seznam,...]
- nelineární = neexistuje [strom, tabulka,...]

2 Spojové seznamy

Lineární seznam (také lineární spojový seznam) je dynamická datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázany vzájemnými odkazy pomocí ukazatelů nebo referencí. Aby byl seznam lineární, nesmí existovat cykly ve vzájemných odkazech.

Lineární seznamy mohou existovat jednosměrné a obousměrné. V jednosměrném seznamu odkazuje každá položka na položku následující a v obousměrném seznamu odkazuje položka na následující i předcházející položky. Zavádí se také ukazatel nebo reference na aktuální (vybraný) prvek seznamu. Na konci (a začátku) seznamu musí být definována zarážka označující konec seznamu. Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o kruhový seznam.

Základním prvkem spojových struktur je dvojice:



- reference - odkaz(y) na další prvek spojové struktury, definuje operátor new – znázorňujeme šipkou
- hodnota – informace libovolného typu

Nejjednodušší typ spojové struktury – jednosměrné spojové seznamy.

Spojové struktury jsou mocným implementačním prostředkem pro ADT, viz další přednáška o ADT a předmět Algoritmizace:

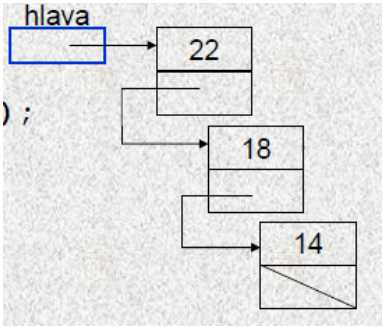
1. – Fronty
2. – Zásobníky
3. – Stromy
4. – Grafy

```
Prvek hlava = null;
int cislo= 14;
while (cislo<26) {

    hlava = new Prvek(cislo,hlava);
    cislo += 4;
```

```
}
```

Tento kód odpovídá struktuře (jednosměrný seznam s odkazem pouze na hlavu):



Jednosměrný seznam lze vylepšit, že se udržuje reference i na poslední, nebo volný prvek.

```
class Prvek {

    int hodnota;
    Prvek dalsi;
    // popr. dalsi konstruktory, treba prazdny, co nastavi vse na null
    public Prvek (int h, Prvek p) {
        hodnota = h;
        dalsi = p;
    }

}
public class Seznam {

    Prvek prvni;
    Prvek volny;
    public Seznam () {
        prvni = new Prvek(0, null);
        volny = prvni;
    }

    public void vlozNaKonec(int x) {
        volny.hodn = x;
        volny.dalsi = new Prvek();
        volny = volny.dalsi;
    }
    public void vypis() {
        Prvek pom = prvni;
        while (pom!=volny) {
            System.out.print(pom.hodn+" ");
            pom = pom.dalsi;
```

```
        }
        System.out.println();
    }
    public void vlozNaZacatek(int x) {
        prvni = new Prvek(x, prvni);
    }
    public boolean jePrvkem1(int x) {
        Prvek pom = prvni;
        while (pom.hodn!=x && pom!=volny)
            pom = pom.dalsi;
        return pom!=volny; }

} // konec tridy Seznam
// volani pr.
Seznam s = new Seznam();
s.vlozNaKonec(6);
s.vlozNaKonec(7);
s.vypis();
```

3 Stromy

V informatice je strom široce využívanou datovou strukturou, která představuje stromovou strukturu s propojenými uzly.

Pojmy:

1. „Cesta“ k nějakému uzlu je definována jako posloupnost všech uzlů od kořene k uzlu.
2. „Délka cesty“ je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti – 1 (mínus jedna).
3. „Hloubka uzlu“ je definována jako délka cesty od kořene k uzlu. Prvky se stejnou hloubkou jsou na „téže úrovni“.
4. „Výška stromu“ je rovna hodnotě maximální hloubky uzlu, se označuje též za „hloubku stromu“.
5. „Šířkou stromu“ je počet uzlů na stejné úrovni.
6. Strom má „nejmenší výšku“ právě tehdy, když na všech úrovních (s možnou výjimkou té poslední) má tato struktura plný počet uzlů. Úroveň všech listů je stejná nebo se liší maximálně o 1.

3.1 Uspořádanost

„Uspořádaný“ nebo také „seřazený strom“ je takový strom, ve kterém jsou všichni přímí potomci každého uzlu seřazeni. Tudíž, pokud uzel má n dětí, lze určit prvního přímého potomka, druhého přímého potomka, až n-tého přímého potomka. U „neuspořádaného stromu“ se jedná o strom v čistě strukturálním smyslu. To znamená, že pro daný uzel nejsou uspořádáni potomci.

3.2 Vyvážený strom

„Vyvážený strom“ je takový strom, který má uzly rovnoměrně rozložené, tedy má nejmenší výšku. Ideální situace je taková, kdy má strom v každé hladině, kromě poslední, maximální počet uzlů, a v poslední hladině má uzly co nejvíce vlevo.

3.3 Procházení stromu (podrobněji pravděpodobně v ALG)

3.3.1 Do šířky

Procházením „stromu do šířky“ (anglicky „level-order“) se rozumí procházení stromem po vrstvách úrovní (tzn. po hladinách).

3.3.2 Do hloubky

Procházení začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větvi (tj. v žádném podstromu) již není následník. Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:

3.3.2.1 PREORDER

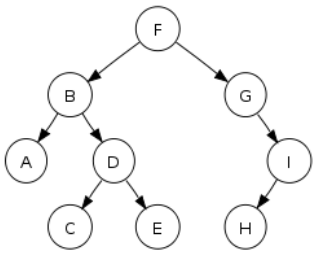
1. proveď akci
2. projdi levý podstrom
3. projdi pravý podstrom

3.3.2.2 INORDER

1. projdi levý podstrom
2. proveď akci
3. projdi pravý podstrom

3.3.2.3 POSTORDER

1. projdi levý podstrom
2. projdi pravý podstrom
3. proveď akci



Výsledky způsobů procházení v binárním vyhledávacím stromu,

N = navštívený uzel, **L** = levý, **R** = pravý

- ***Preorder*** (NLR): F, B, A, D, C, E, G, I, H
- ***Inorder*** (LNR): A, B, C, D, E, F, G, H, I
- ***Postorder*** (LRN): A, C, E, D, B, H, I, G, F
- Procházení do šířky (*po vrstvách*) ***Level-order***: F, B, G, A, D, I, C, E, H

3.4 Binární stromy

Každý prvek (uzel) má nanejvýš dva následníky (obecně lze prvkům stromu říkat také node, z ang., čteno [nouda]). Binární strom obsahuje uzly, které mají nejvíce dva syny.
Důležité pojmy:

1. kořen stromu - nejvyšší prvek, nemá rodiče
2. levý podstrom - strom levých potomků
3. pravý podstrom - pravých
4. vnitřní uzel - prvek, který má potomky (často se sem nepočítá kořen, někdy ano)
5. list - prvek, který již nemá potomky

U binarního stromu rozlišujeme další pojmy:

- Plný binární strom - všechny jeho listy jsou ve stejné hloubce.
- Úplný binární strom - každý vnitřní uzel má dva syny.
- Vyvážený binární strom - hloubka listů se od sebe liší maximálně o jedna.

3.4.1 Trochu matiky

- h – hloubka stromu,
- n – počet uzlů
- n0 – počet listů
- n2 – počet vnitřních uzlů

Úplný binární strom:
minimální počet uzlů: $n = 2^h + 1$
maximální počet uzlů: $n{=}2^{h+1} - 1$
počet listů: n/2 (zaokrouhleno nahoru)

4 Abstraktní datové typy

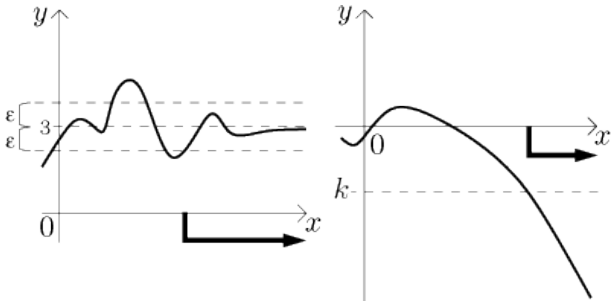
Tato část předmětu PRG2 není napsána v zadání u této otázky a je součástí ALG (je pouze obsahem slajdů k PRG2). Proto zde není popsána.

Společná část: 8 - MA2

June 6, 2012

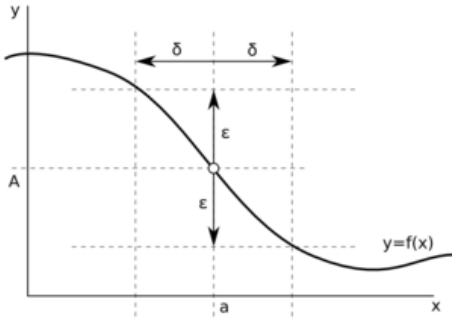
0.1 Limita funkce a posloupnosti

0.1.1 funkce



Definice. (limita pro vlastní bod a vlastní limitu)
Nechť f je funkce definovaná na nějakém prstencovém okolí bodu $a \in \mathbb{R}$, nechť $L \in \mathbb{R}$.
Řekneme, že „L je **limita** funkce f pro x jdoucí k a“, nebo že „f jde k L pro x jdoucí k a“, jestliže $\forall \varepsilon > 0\ \exists \delta > 0$ takové, že $\forall x$: $\big[0 < |x-a| < \delta \implies |f(x) - L| < \varepsilon\big]$.

Zapisujeme to „$\lim\limits_{x \to a} \big(f(x)\big) = L$,“ popřípadě „$f(x) \to L$ pro $x \to a$“.



Definice.
Nechť $a \in \mathbb{R}$, $\varepsilon > 0$. Definujeme okolí bodu a:

$U_\varepsilon(a) = \{x \in \mathbb{R};\ \|x-a\| < \varepsilon\} = (a-\varepsilon, a+\varepsilon)$	**ε-okolí bodu** a
$P_\varepsilon(a) = \{x \in \mathbb{R};\ 0 < \|x-a\| < \varepsilon\} = (a-\varepsilon, a) \cup (a, a+\varepsilon)$	**prstencové ε-okolí bodu** a
$U_\varepsilon^+(a) = \{x \in \mathbb{R};\ a \le x < a+\varepsilon\} = \langle a, a+\varepsilon)$	**pravé ε-okolí bodu** a
$P_\varepsilon^+(a) = \{x \in \mathbb{R};\ a < x < a+\varepsilon\} = (a, a+\varepsilon)$	**pravé prstencové ε-okolí bodu** a
$U_\varepsilon^-(a) = \{x \in \mathbb{R};\ a-\varepsilon < x \le a\} = (a-\varepsilon, a\rangle$	**levé ε-okolí bodu** a
$P_\varepsilon^-(a) = \{x \in \mathbb{R};\ a-\varepsilon < x < a\} = (a-\varepsilon, a)$	**levé prstencové ε-okolí bodu** a

Definice. (jednostranné limity)
Nechť f je funkce definovaná na nějakém levém prstencovém okolí bodu $a \in \mathbb{R} \cup \{\infty\}$, nechť $L \in \mathbb{R}^*$.
Řekneme, že „L je **limita** funkce f pro x jdoucí k a **zleva**“,
jestliže $\forall$ okolí $U = U(L)\ \exists$ levé prstencové okolí $P = P^-(a)$ takové, že $\forall x \in P$: $f(x) \in U$.
Říkáme také, že „f jde k L pro x jdoucí k a zleva“, nebo „f má limitu L v a zleva“, nebo že „f jde k L v a zleva“. Zapisujeme to „$\lim\limits_{x \to a^-} \big(f(x)\big) = L$,“ popřípadě zkráceně „$f(a^-) = L$“.

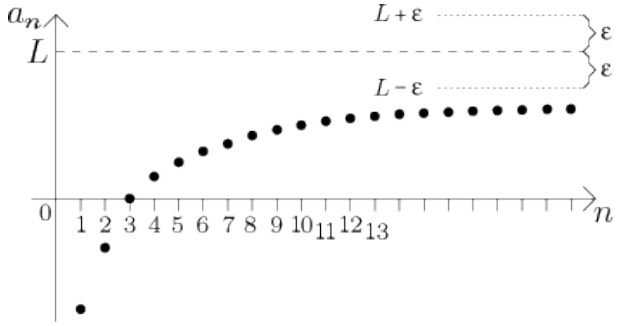
Nechť f je funkce definovaná na nějakém pravém prstencovém okolí bodu $a \in \mathbb{R} \cup \{-\infty\}$, nechť $L \in \mathbb{R}^*$.
Řekneme, že „L je **limita** funkce f pro x jdoucí k a **zprava**“,
jestliže $\forall$ okolí $U = U(L)\ \exists$ pravé prstencové okolí $P = P^+(a)$ takové, že $\forall x \in P$: $f(x) \in U$.
Říkáme také, že „f jde k L pro x jdoucí k a zprava“, nebo „f má limitu L v a zprava“, nebo že „f jde k L v a zprava“. Zapisujeme to „$\lim\limits_{x \to a^+} \big(f(x)\big) = L$“, popřípadě zkráceně „$f(a^+) = L$“.

- Pokud najdeme limitu L, která je **reálné číslo**, řekneme, že je to **vlastní limita**

a že limita konverguje. Jinak řekneme, že limita diverguje. Limita **nekonečno nebo mínus nekonečno** se nazývá **nevlastní limita**. Pokud najdeme nějakou limitu (vlastní či nevlastní), řekneme, že limita existuje. Jinak řekneme, že limita neexistuje.

Definice.
Nechť f je funkce definovaná na neprázdné množině M.
Jestliže je f omezená shora na M, definujeme její **supremum** na M, značeno $\sup_M(f)$, jako nejmenší horní mez f na M. Jinak definujeme $\sup_M(f) = \infty$.
Jestliže je f omezená zdola na M, definujeme její **infimum** na M, značeno $\inf_M(f)$, jako největší dolní mez f na M. Jinak definujeme $\inf_M(f) = -\infty$.

0.1.2 posloupnost



- **Definice:** Uvažujme posloupnost a_n. Řekneme, že nekonečno je limita této posloupnosti pro n jdoucí do nekonečna, nebo že posloupnost jde do nekonečna pro n jdoucí do nekonečna, jestliže pro každé reálné číslo K existuje přirozené číslo N takové, že pro všechna n = N, N + 1, N + 2,... máme $a_n > K$.
- Když má posloupnost limitu, která je reálné číslo, řekneme, že posloupnost konverguje. Taková limita se nazývá **vlastní limita**.
- Když má posloupnost limitu, která je plus či mínus nekonečno, říkáme této limitě **nevlastní limita**.
- Když má posloupnost limitu, vlastní či nevlastní, řekneme, že limita **existuje**.
- Pokud posloupnost nemá vůbec žádnou limitu, řekneme, že limita **neexistuje**.
- Posloupnosti s nevlastní limitou a bez limity se nazývají **divergentní**.

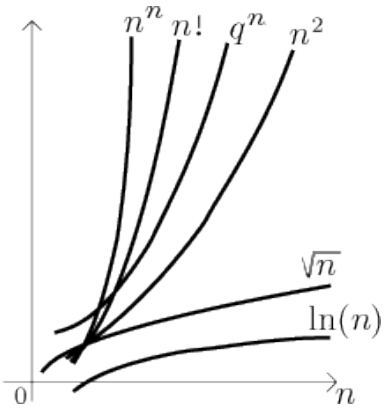
0.1.3 Rychlost růstu

škála mocnin

Fakt. (**škála mocnin v nekonečnu**)
Pro libovolná $a, b > 0$:

$$\lim_{x\to\infty}\left(\frac{e^{ax}}{x^b}\right) = \infty, \qquad \lim_{x\to\infty}\left(\frac{x^a}{\ln^b(x)}\right) = \infty.$$

Značení: $a^x \gg x^a \gg \ln^b(x)$.



0.1.4 L'Hopitalovo pravidlo

Při hledání limity podílu dvou funkcí (i posloupností) dostaneme "neurčitý podíl" →řešíme l'Hopitalovým pravidlem

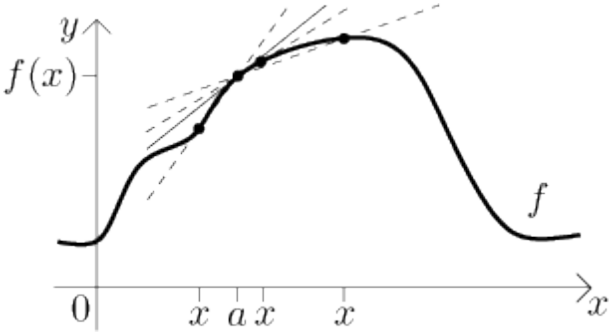
$$\lim_{n\to\infty}\left(\frac{a_n}{b_n}\right) \Rightarrow \lim_{x\to\infty}\left(\frac{f(x)}{g(x)}\right) = \left\langle\!\!\left\langle \begin{matrix} f(x)\to 0 \text{ pro } x\to\infty \\ g(x)\to 0 \text{ pro } x\to\infty \end{matrix} \right\}_{\frac{0}{0}} \implies \text{l'H} \right\rangle\!\!\right\rangle$$
$$= \lim_{x\to\infty}\left(\frac{f'(x)}{g'(x)}\right),$$

$$\lim_{n\to\infty}\left(\frac{a_n}{b_n}\right) \Rightarrow \lim_{x\to\infty}\left(\frac{f(x)}{g(x)}\right) = \left\langle\!\!\left\langle \begin{matrix} f(x)\to \pm\infty \text{ pro } x\to\infty \\ g(x)\to \pm\infty \text{ pro } x\to\infty \end{matrix} \right\}_{\frac{\infty}{\infty}} \implies \text{l'H} \right\rangle\!\!\right\rangle$$
$$= \lim_{x\to\infty}\left(\frac{f'(x)}{g'(x)}\right).$$

příklad

$$\lim_{n\to\infty}\left(\frac{\ln(n)}{n^2}\right) = \left\langle\!\!\left\langle \frac{\infty}{\infty} \right\rangle\!\!\right\rangle \Rightarrow \lim_{x\to\infty}\left(\frac{\ln(x)}{x^2}\right)$$
$$= \left\langle\!\!\left\langle \begin{matrix} \ln(x)\to\infty \text{ pro } x\to\infty \\ x^2\to\infty \text{ pro } x\to\infty \end{matrix} \right\}_{\frac{\infty}{\infty}} \implies \text{l'H} \right\rangle\!\!\right\rangle$$
$$= \lim_{x\to\infty}\left(\frac{\frac{1}{x}}{2x}\right) = \lim_{x\to\infty}\left(\frac{1}{2x^2}\right) = \left\langle\!\!\left\langle \frac{1}{\infty} \right\rangle\!\!\right\rangle = 0.$$

0.2 Derivace



Definice.
Nechť f je funkce definovaná na nějakém okolí bodu a.
Řekneme, že f je **diferencovatelná** v a, jestliže $\lim_{x\to a}\left(\frac{f(x)-f(a)}{x-a}\right)$ konverguje.
Pak definujeme **derivaci** f v a jako

$$f'(a) = \lim_{x\to a}\Big(\frac{f(x)-f(a)}{x-a}\Big).$$

Alternativní vzorec: $f'(a) = \lim_{h\to 0}\left(\frac{f(a+h)-f(a)}{h}\right)$.

Leibnizovo značení:

$$f'(a) = \frac{\mathrm{d}f}{\mathrm{d}x}(a) = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=a}.$$

Věta. (derivace v bodě a skládání funkcí)
Nechť je funkce f diferencovatelná v a a funkce g diferencovatelná v $b = f(a)$.
Pak je funkce $g \circ f = g(f)$ diferencovatelná v a a platí

$$(g\circ f)'(a) = g'(f(a))\cdot f'(a).$$

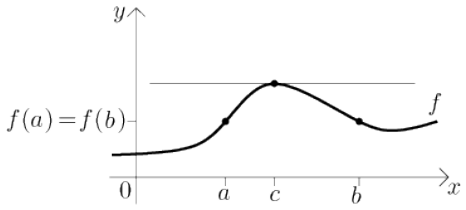
Definice.
Nechť f je funkce definovaná na nějakém pravém okolí bodu a.
Řekneme, že f je **diferencovatelná** v a **zprava**, jestliže $\lim_{x\to a^+}\left(\frac{f(x)-f(a)}{x-a}\right)$ konverguje.
Pak definujeme **derivaci** f v a **zprava** jako

$$f'_+(a) = \lim_{x\to a^+}\Big(\frac{f(x)-f(a)}{x-a}\Big).$$

Nechť f je funkce definovaná na nějakém levém okolí bodu a.
Řekneme, že f je **diferencovatelná** v a **zleva**, jestliže $\lim_{x\to a^-}\left(\frac{f(x)-f(a)}{x-a}\right)$ konverguje.
Pak definujeme **derivaci** f v a **zleva** jako

$$f'_-(a) = \lim_{x\to a^-}\Big(\frac{f(x)-f(a)}{x-a}\Big).$$

0.2.1 vlastnosti derivace

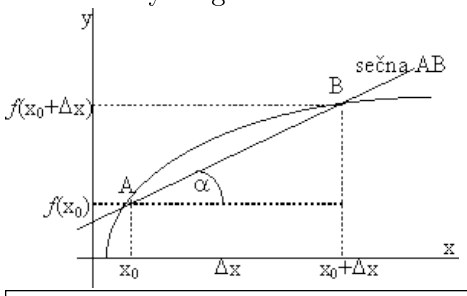
- jestliže je f diferencovatelná v a a $f'(a) \neq 0$, pak je i příslušná inverzní funkce f_{-1}diferencovatelná v b
- Jestliže je funkce f diferencovatelná v bodě a, pak je f spojitá v a.
- Nechť a < b jsou reálná čísla. Nechť f je funkce spojitá na intervalu a,b a diferencovatelná na (a,b). Jestliže f(a) = f(b), pak existuje c z (a,b) takové, že $f'(c) = 0$ (věta o střední hodnotě - **Rolleova věta**)
- 

- **Věta.** (Věta o střední hodnotě, Lagrangeova věta)
Nechť f je spojitá na intervalu $\langle a, b\rangle$ a diferencovatelná na jeho vnitřku (a, b).
Pak existuje $c \in (a, b)$: $f'(c) = \frac{f(b) - f(a)}{b - a}$.

0.2.2 význam derivace

0.2.2.1 geometrický význam

směrnice tečny ke grafu dané funkce v daném bodě



$$f'(x_0) = \lim_{x \to x_0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

0.2.2.2 fyzikální význam

- derivace podle časové proměnné, vyjadřující rychlost změny nějaké proměnné v čase (např. okamžitá rychlost: $v = \frac{ds}{dt}$)
- diferenciální rovnice

0.2.3 Monotonie

- vlastnost, označující, zda je funkce v bodě či na daném intervalu monotónní

existuje nějaké okolí U(a) bodu a takové, že pro všechna x v tomto okolí platí:
rostoucí
$x > a \Rightarrow f(x) > f(a) x < a \wedge \Rightarrow f(x) < f(a)$.
klesající
$x > a \Rightarrow f(x) < f(a) \wedge x < a \Rightarrow f(x) > f(a)$,
nerostoucí
$x > a \Rightarrow f(x) \leq f(a) \wedge x < a \Rightarrow f(x) \geq f(a)$,
neklesající
$x > a \Rightarrow f(x) \geq f(a) \wedge x < a \Rightarrow f(x) \leq f(a)$.

- typ monotonie určíme z první derivace f(x)

rostoucí pro f′(x) > 0; **klesající** pro f′(x) < 0

0.2.3.1 kritický bod

Definice: Nechť je funkce f definovaná na nějakém okolí bodu c. Řekneme, že c je **kritický bod**, jestliže f ’(c) = 0 nebo f ’(c) neexistuje.

0.2.4 Lokální extrémy

Definice.
Nechť f: $D(f) \mapsto \mathbb{R}$ je funkce, kde $D(f) \subseteq \mathbb{R}^n$. Nechť $\vec{a}$ je vnitřní bod $D(f)$.
Řekneme, že f má v $\vec{a}$ **lokální maximum** nebo že $f(\vec{a})$ je lokální maximum, jestliže existuje $U = U(\vec{a})$ takové, že $f(\vec{a}) \geq f(\vec{x})$ pro všechna $x \in U$.
Řekneme, že f má v $\vec{a}$ **lokální minimum** nebo že $f(\vec{a})$ je lokální minimum, jestliže existuje $U = U(\vec{a})$ takové, že $f(\vec{a}) \leq f(\vec{x})$ pro všechna $x \in U$.
Pokud jsou v definici maxima/minima ostré nerovnosti pro $\vec{x} \neq \vec{a}$, pak se dotyčný extrém nazývá **ostrý**.

Nechť je f spojitá v c:

- Jestliže existuje pravé okolí c, na kterém je f rostoucí, a levé okolí a, na kterém je f klesající, pak má f lokální minimum v c
- Jestliže existuje pravé okolí c, na kterém je f klesající, a levé okolí a, na kterém je f rostoucí, pak má f lokální maximum v c

Definice.
Nechť f je funkce definovaná na intervalu I.
Řekneme, že f je na intervalu I **konvexní**, jestliže

$$\forall x < y < z \in I\colon \frac{f(y)-f(x)}{y-x} \leq \frac{f(z)-f(y)}{z-y}.$$

Řekneme, že f je na intervalu I **konkávní**, jestliže

$$\forall x < y < z \in I\colon \frac{f(y)-f(x)}{y-x} \geq \frac{f(z)-f(y)}{z-y}.$$

inflexní bod - f přechází z konvexní na konkávní nebo naopak a je tam dvakrát diferencovatelná

0.2.5 Asymptoty

Definice.
Nechť f je funkce definovaná na nějakém jednostranném prstencovém okolí bodu $a \in \mathbb{R}$.
Řekneme, že přímka $x = a$ je **svislá asymptota** f, nebo že f má svislou asymptotu v a, jestliže

$$\lim_{x\to a^+}\big(f(x)\big) = \pm\infty \qquad \text{nebo} \qquad \lim_{x\to a^-}\big(f(x)\big) = \pm\infty.$$

Definice.
Nechť f je funkce definovaná na okolí ∞.
Řekneme, že přímka $y = B$ je **vodorovná asymptota** f v ∞,
jestliže $\lim_{x\to\infty}\big(f(x)\big) = B$.

Nechť f je funkce definovaná na okolí $-\infty$.
Řekneme, že přímka $y = B$ je **vodorovná asymptota** f v $-\infty$,
jestliže $\lim_{x\to-\infty}\big(f(x)\big) = B$.

Definice.
Nechť f je funkce definovaná na okolí ∞.
Řekneme, že přímka $y = Ax + B$ je **šikmá asymptota** f v ∞,
jestliže $\lim_{x\to\infty}\big(f(x) - (Ax+B)\big) = 0$.

Nechť f je funkce definovaná na okolí $-\infty$.
Řekneme, že přímka $y = Ax + B$ je **šikmá asymptota** f v $-\infty$,
jestliže $\lim_{x\to-\infty}\big(f(x) - (Ax+B)\big) = 0$.

0.2.6 Parciální derivace

Parciální derivace funkce více proměnných představuje v matematice takovou derivaci dané funkce, při které se derivuje pouze vzhledem **k jedné z proměnných**, ostatní proměnné jsou považovány za konstanty

0.2.7 Gradient

= diferenciální operátor udávající směr růstu

Definice.
Nechť f: $D(f) \mapsto \mathbb{R}$ je funkce, $D(f) \subseteq \mathbb{R}^n$. Nechť $\vec{a}$ je vnitřní bod $D(f)$.
Jestliže existují všechny parciální derivace $\frac{\partial f}{\partial x_i}(\vec{a})$ pro $= 1, \ldots, n$, pak definujeme **gradient** f v $\vec{a}$ jako vektor

$$\nabla f(\vec{a}) = \Big(\frac{\partial f}{\partial x_1}(\vec{a}), \ldots, \frac{\partial f}{\partial x_n}(\vec{a})\Big).$$

Alternativní značení: $\nabla f(\vec{a}) = \vec{\nabla} f(\vec{a}) = \mathrm{grad}(f)(\vec{a})$.

Věta. (Sylvesterovo kritérium)
Nechť G je otevřená množina v $\mathbb{R}^n$ a $f \in C^1(G)$.
Nechť $\vec{a} \in G$ je stacionární bod f a H je Hessova matice f v $\vec{a}$. Nechť Δ_i jsou levé horní subdeterminanty H.
Jestliže $\Delta_i > 0$ pro všechna i, pak je $f(\vec{a})$ (ostré) lokální minimum.
Jestliže $\Delta_1 < 0$, $\Delta_2 > 0$, $\Delta_3 < 0$ atd. až $(-1)^n \Delta_n > 0$, pak je $f(\vec{a})$ (ostré) lokální maximum.
Jestliže $\Delta_2 < 0$, pak je $f(\vec{a})$ **sedlový bod**.

Společná část: 9 - MA2

June 4, 2012

Abstract

Význam integrálu, základní metody výpočtu a nevlastní integrál; řady a jejich konvergence (význam, příklady použití, geometrická řada), Taylorův polynom a řada.

1 Integrál

1.1 Neurčitý integrál

Poznámka: Habala ho ztotožňuje s Newtonovym (sou i jiný definice, ale držel bych se jeho)

= množina primitivních funkcí integrované funkce

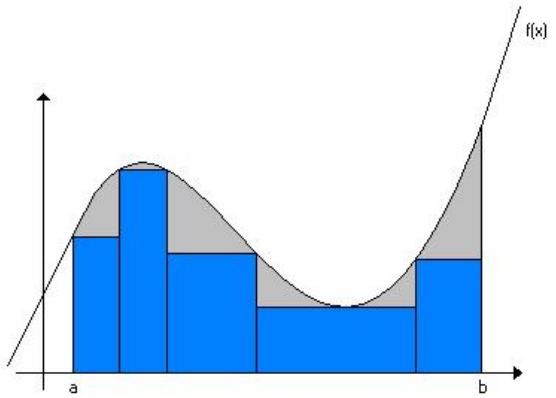
Primitivní funkce: Nechť f je funkce na intervalu I. Řekneme, že funkce F je primitivní funkce k f na I, jestliže je F spojitá na I, diferencovatelná na jeho vnitřku I^O a $F' = f$ na I^O.

Newtonův integrál: Nechť f je funkce, která má na intervalu I primitivní funkci. Definujeme neurčitý integrál f na I jako množinu všech takových primitivních funkcí. Značení: $\int f(x)dx = \{$F; F je primitivní funkce k f na I$\}$. Jestliže máme jednu takovou primitivní funkci F, pak nepřesně ale tradičně píšeme $\int f(x)dx = F(x) + C, x \in I$.

1.2 Určitý integrál

Riemannův (určitý) integrál odpovídá matematickému obsahu oblasti pod grafem f, který je roven geometrickému obsahu částí nad osou x mínus obsah částí pod osou x.

Nekonečný součet nekonečně malých (úzkých) sloupců pod křivkou



$\int_c^b f(x)dx$

1.2.1 Výpočet určitého integrálu

- Standardní způsob výpočtu určitého integrálu je založen na **základní větě integrálního počtu**

Nejprve se najde primitivní funkce F k dané funkci f na daném intervalu a,b a pak se použije Newton-Leibnizův vzorec:

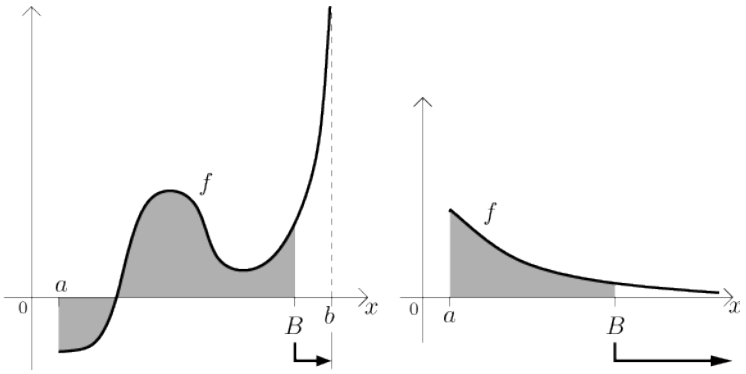
$$\int_a^b f(x)\,dx = F(b) - F(a).$$

$$F(b) - F(a) = \Big[F(x)\Big]_a^b.$$

funguje jen pro funkce f, které jsou spojité na uzavřeném intervalu <a,b>, jinak řešíme jako **nevlastní integrál**

- při hledání primitivní funkce se používá metoda **substituce** a **per-partes**

1.3 Nevlastní integrál



Definice: Nechť a je reálné číslo, nechť b > a je reálné číslo nebo b = ∞ . Nechť f je funkce Riemannovsky integrovatelná na intervalech a, B pro všechna B z (a, b). Pak definujeme nevlastní Riemannův integrál z f od a do b jako:

o nevlastní integrál se jedná pokud:

- jedna či obě integrační meze (koncové body integračního intervalu) jsou nekonečné
- integrovaná funkce není spojitá v některých bodech integračního intervalu

$$\int_a^b f(x)\,dx = \lim_{B\to b^-}\left(\int_a^B f(x)\,dx\right)$$

$$\int_a^b f(x)\,dx = \lim_{A\to a^+}\left(\int_A^b f(x)\,dx\right)$$

1.4 Metody výpočtu

1.4.1 substituce

obecně

$$\int f(g(x))g'(x)\,dx = \left|\begin{array}{c} y = g(x) \\ dy = g'(x)\,dx \end{array}\right| = \int f(y)\,dy$$
$$= F(y) + C = F(g(x)) + C.$$

příklad

$$\int \operatorname{cotg}(x)\,dx = \int \frac{\cos(x)}{\sin(x)}\,dx = \int \frac{1}{\sin(x)}\,\cos(x)\,dx = \left|\begin{array}{c} y = \sin(x) \\ dy = \cos(x)\,dx \end{array}\right|$$
$$= \int \frac{1}{y}\,dy = \ln|y| + C = \ln|\sin(x)| + C,\ x \neq k\pi.$$

1.4.2 per-partes

obecně

$$\int_a^b f(x)g'(x)\,dx = \Big[f(x)g(x)\Big]_a^b - \int_a^b f'(x)g(x)\,dx.$$

příklad

$$\int_0^\pi x\cos(x)\,dx = \left|\begin{array}{ll} f = x & g' = \cos(x) \\ f' = 1 & g = \sin(x) \end{array}\right| = \Big[x\,\sin(x)\Big]_0^\pi - \int_0^\pi \sin(x)\,dx$$
$$= \pi\sin(\pi) - 0 - \int_0^\pi \sin(x)\,dx = -\int_0^\pi \sin(x)\,dx.$$

1.4.3 parciální zlomky

$$\frac{p(x)}{q(x)} = \sum_{n=1}^{N}\sum_{i=1}^{r_n}\frac{A_{n,i}}{(x-a_n)^i} + \sum_{m=1}^{M}\sum_{j=1}^{t_m}\frac{B_{m,j}x + C_{m,j}}{(x^2+b_mx+c_m)^j}$$
$$= \frac{A_{1,1}}{(x-a_1)} + \frac{A_{1,2}}{(x-a_1)^2} + \ldots + \frac{A_{1,r_1}}{(x-a_1)^{r_1}}$$
$$+ \frac{A_{2,1}}{(x-a_2)} + \ldots + \frac{A_{2,r_2}}{(x-a_2)^{r_2}} + \ldots + \frac{A_{N,1}}{(x-a_N)} + \ldots + \frac{A_{N,r_N}}{(x-a_2)^{r_N}}$$
$$+ \frac{B_{1,1}x + C_{1,1}}{(x^2+b_1x+c_1)} + \frac{B_{1,2}x + C_{1,2}}{(x^2+b_1x+c_1)^2} + \ldots \frac{B_{1,t_1}x + C_{1,t_1}}{(x^2+b_1x+c_1)^{t_1}} + \ldots$$
$$+ \frac{B_{M,1}x + C_{M,1}}{(x^2+b_Mx+c_M)} + \ldots + \frac{B_{M,t_M}x + C_{M,t_M}}{(x^2+b_Mx+c_M)^{t_M}}.$$

1.4.4 tabulkové integrály

$$\int x^\alpha\,dx = \frac{x^{\alpha+1}}{\alpha+1} + C,\ x > 0; \quad \text{pro } \alpha \neq -1$$

$$\int \frac{1}{x}\,dx = \ln|x| + C,\ x \neq 0 \qquad \int e^x\,dx = e^x + C$$

$$\int \sin(x)\,dx = -\cos(x) + C \qquad \int \frac{1}{\cos^2(x)}\,dx = \mathrm{tg}(x) + C,\ x \neq \frac{\pi}{2} + k\pi$$

$$\int \cos(x)\,dx = \sin(x) + C \qquad \int \frac{1}{\sin^2(x)}\,dx = -\,\mathrm{cotg}(x) + C,\ x \neq k\pi$$

$$\int \sinh(x)\,dx = \cosh(x) + C \qquad \int \frac{1}{\cosh^2(x)}\,dx = \mathrm{tgh}(x) + C$$

$$\int \cosh(x)\,dx = \sinh(x) + C \qquad \int \frac{1}{\sinh^2(x)}\,dx = -\,\mathrm{cotgh}(x) + C,\ x \neq 0$$

$$\int \frac{1}{1+x^2}\,dx = \mathrm{arctg}(x) + C \qquad \int \frac{1}{\sqrt{1-x^2}}\,dx = \arcsin(x) + C,\ x \in (-1,1)$$

2 Řady

Definice: Nechť $a_{k\,k\geqq n_0}$ je posloupnost (reálných čísel). Pojmem řada (reálných čísel) rozumíme abstraktní symbol

$\sum_{k=n_0}^{\infty} a_k$
Pro všechna celá čísla $N > n_0$ definujeme její částečné součty řady vzorcem:
$s_n = \sum_{k=n_0}^{N} a_k$

- řada konverguje k A, jestliže posloupnost s_n konverguje (k A).

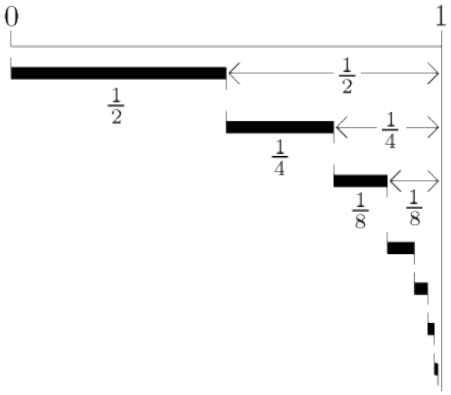
$\sum_{k=n_0}^{\infty} a_k = a$

- řada diverguje, jestliže posloupnost s_n diverguje.

$\sum_{k=n_0}^{\infty} a_k = \pm\infty$

příklad:

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots$$



2.0.5 Testování konvergence řad

Absolutní konvergence: Řada $\sum_{k=n_0}^{\infty} a_k$ absolutně konverguje, pokud konverguje $\sum_{k=n_0}^{\infty} |a_k|$

metody testování (všechny na http://math.feld.cvut.cz/mt/txte/2/txc3eb2.htm)

Věta. (**integrální kritérium**)
Nechť $f \geq 0$ je nerostoucí na $\langle n_0, \infty)$ pro $n_0 \in \mathbb{Z}$.
Řada $\sum_{k=n_0}^{\infty} f(k)$ konverguje právě tehdy, když $\int_{n_0}^{\infty} f(x)\,dx$ konverguje.

Navíc pak platí

$$\int_{n_0}^{\infty} f(x)\,dx \leq \sum_{k=n_0}^{\infty} f(k) \leq f(n_0) + \int_{n_0}^{\infty} f(x)\,dx.$$

Důsledek. (p-**test**)
$\sum \frac{1}{k^p}$ konverguje tehdy a jen tehdy, když $p > 1$.

Věta. (**srovnávací kritérium**)
Uvažujme řady $\sum a_k$, $\sum b_k$.
Nechť existuje n_0 tak, aby $0 \leq a_k \leq b_k$ pro všechna $k \geq n_0$.

(i) Jestliže $\sum b_k$ konverguje, pak také $\sum a_k$ konverguje.

(ii) Jestliže $\sum a_k$ diverguje, pak také $\sum b_k$ diverguje

Symbolicky: $a_k \leq b_k \implies \sum a_k \leq \sum b_k$.

Věta. (**limitní srovnávací kritérium**)
Uvažujme řady $\sum a_k$, $\sum b_k$.
Nechť existuje $n_0 \in \mathbb{Z}$ tak, aby $a_k, b_k > 0$ pro všechna $k \geq n_0$.

Předpokládejme, že $\lim_{k\to\infty} \left(\frac{a_k}{b_k}\right) = A > 0$. Pak

$\sum a_k$ konverguje tehdy a jen tehdy, když konverguje $\sum b_k$.

Symbolicky: $a_k \sim b_k \implies \sum a_k \sim \sum b_k$.

Věta.
Uvažujme řadu $\sum a_k$, nechť $a_k \geq 0$ pro všechna k.

(i) **(limitní) odmocninové kritérium**:
Předpokládejme, že $\varrho = \lim_{k\to\infty} \left(\sqrt[k]{a_k}\right)$ konverguje.
1) Jestliže $\varrho < 1$, pak $\sum a_k$ konverguje.
2) Jestliže $\varrho > 1$, pak $\sum a_k$ diverguje $(= \infty)$.

(ii) **(limitní) podílové kritérium**:
Předpokládejme, že $\lambda = \lim_{k\to\infty} \left(\frac{a_{k+1}}{a_k}\right)$ konverguje.
1) Jestliže $\lambda < 1$, pak $\sum a_k$ konverguje.
2) Jestliže $\lambda > 1$, pak $\sum a_k$ diverguje $(= \infty)$.

- pro alternující řady ("střídající $+, -+, ...$")

Věta. (**Leibnizovo kritérium**)
Uvažujme řadu $\sum a_k$, nechť $a_k = (-1)^k b_k$.
Předpokládejme, že $b_k \geq 0$ pro všechna k a $\{b_k\}$ je nerostoucí.

Řada $\sum (-1)^k b_k$ konverguje právě tehdy, když $\lim\limits_{k\to\infty} (b_k) = 0$.

2.1 Geometrická řada

=součet členů geometrické posloupnosti ($a_{n+1} = a_n \cdot q$)

definice: Nechť $a, q \in R$. Řada $\sum_{k=n_0}^{\infty} a \cdot q^k$ se nazývá **geometrická řada**.

Součet geometrické řady je dán jako limita posloupnosti n-tých částečných součtů:

$$\lim_{n\to\infty} s_n = \lim_{n\to\infty} \frac{a_1}{1-q} + \lim_{n\to\infty} \frac{a_1 \cdot q^n}{q-1}$$

$$\lim_{n\to\infty} s_n = \begin{cases} \frac{a_1}{1-q} & \text{pro } |q| < 1 \\ \pm\infty, & \text{pro } q \geq 1 \\ \text{nekonverguje (osciluje)} & \text{pro } q \leq -1 \end{cases}$$

2.2 Aplikace řad

2.2.1 použití Fourierových řad pro frekvenční analýzu

Slouží k zápisu jakéhokoliv periodického průběhu pomocí goniometrických funkcí sinus a kosinus.

Základní myšlenka Fourierových řad je, že danou funkci vyjádříme jako kombinaci oscilací, počínaje tou, jejíž frekvence je dána zadanou funkcí (buď její periodicitou nebo délkou omezeného intervalu, na kterém je zadána), a pak se berou násobky této frekvence čili používáme dělených period.

- **zvuková komprese:** Když dostaneme zvukový vzorek, Fourierova transformace nám umožňuje jej rozložit na základní vlny a uchovat v tomto tvaru.
- **uchovávání obrazové informace** (např. databáze otisků)

2.2.2 Mocninná řada ve výpočtech

- Před rozmachem kalkulačkek se při výpočtech všechny funkce nahrazovaly Taylorovými řadami, popřípadě jejich konečnými částmi - polynomy.
- **vyčíslení Pí:** Pí je transcendentní číslo, což znamená, že jej nemůžeme vyjádřit pomocí algebraických operací. Jeden způsob jeho vyčíslení nabízí řady.
- **výpočet složitých integrálů,** které nelze vyjádřit pomocí elementárních funkcí a obvyklých operací (včetně skládání)

- **Taylorův polynom**

2.3 Taylorův polynom a řada

Taylorův polynom aproximuje hodnoty funkce, která má v daném bodě derivaci, pomocí polynomu, jehož koeficienty závisí na derivacích funkce v tomto bodě. Čím vyšší stupeň tím vyšší přesnost pro vzdálenější body.

$$\begin{aligned} T_n(x) &= f(a) + f'(a)(x-a) + \tfrac{1}{2!}f''(a)(x-a)^2 + \ldots + \tfrac{1}{n!}f^{(n)}(a)(x-a)^n \\ &= \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k. \end{aligned}$$

Taylorova řada se liší od polynomu tím, že se jedná o **nekonečný** součet

$$T(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

2.3.1 důležité řady

$$e^x = \sum_{k=0}^{\infty} \frac{1}{k!}x^k = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots, \qquad x \in I\!R;$$

$$\sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!}x^{2k+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, \qquad x \in I\!R;$$

$$\cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!}x^{2k} = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, \qquad x \in I\!R;$$

$$\begin{aligned} \ln(x) &= \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k}(x-1)^k \\ &= (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \ldots, \qquad x \in (0,2\rangle; \end{aligned}$$

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + x^3 + \ldots, \qquad x \in (-1,1);$$

$$\begin{aligned} (c+x)^A &= \sum_{k=0}^{\infty} \binom{A}{k} c^{A-k}x^k \\ &= \sum_{k=0}^{\infty} \frac{A(A-1)\cdot\ldots\cdot(A-k+1)}{k!}c^{A-k}x^k, \qquad x \in (-c,c). \end{aligned}$$

10 Syntaxe a sémantika výrokové a predikátové logiky. Sémantický důsledek a tautologická ekvivalence. Booleovský kalkul. Rezoluční metoda (A0B01LGR)

10.1 Výroková logika

10.1.1 Výroky

Máme danou neprázdnou množinu A tzv. *atomických výroků* (též jim říkáme *logické proměnné*). Konečnou posloupnost prvků z množiny A, logických spojek a závorek nazýváme *výroková formule* (zkráceně jen *formule*), jestliže vznikla podle následujících pravidel:

1. Každá logická proměnná (atomický výrok) $a \in A$ je výroková formule.
2. Jsou-li α, β výrokové formule, pak $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$ a $(\alpha \Leftrightarrow \beta)$ jsou také výrokové formule.
3. Nic jiného než to, co vzniklo pomocí konečně mnoha použití bodů 1 a 2, není výrokové formule.

Všechny formule, které vznikly z logických proměnných množiny A, značíme $P(A)$.

Poznámka: Spojka $\neg$ se nazývá *unární*, protože vytváří novou formuli z jedné formule. Ostatní zde zavedené spojky se nazývají *binární*, protože vytvářejí novou formuli ze dvou formulí.

V dalším textu logické proměnné označujeme malými písmeny např. $a, b, c, \ldots$ nebo $x, y, z, \ldots$, výrokové formule označujeme malými řeckými písmeny např. $\alpha, \beta, \gamma, \ldots$ nebo $\varphi, \psi, \ldots$. Také většinou nebudeme ve formulích psát ty nejvíc vnější závorky - tj. píšeme $a \vee (b \Rightarrow c)$ místo $(a \vee (b \Rightarrow c))$.

10.1.2 Syntaktický strom formule

To jak formule vznikla podle bodů 1 a 2, si můžeme znázornit na *syntaktickém stromu*, též *derivačním stromu* dané formule. Jedná se o kořenový strom, kde každý vrchol, který není listem je ohodnocen logickou spojkou a jedná-li se o binární spojku, má vrchol dva následníky, jedná-li se o unární spojku, má vrchol pouze jednoho následníka. Přitom pro

formule $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \Rightarrow \beta)$ odpovídá levý následník formuli α, pravý následník formuli β. Listy stromu jsou ohodnoceny logickými proměnnými.

10.1.3 Podformule

Ze syntaktického stromu formule α jednoduše poznáme všechny její podformule: *Podformule* formule α jsou všechny formule odpovídající podstromům syntaktického stromu formule α.

10.1.4 Pravdivostní ohodnocení

Pravdivostní ohodnocení, též pouze *ohodnocení formulí*, je zobrazení $u : P(A) \to \{0,1\}$, které splňuje pravidla

1. $\neg\alpha$ je pravdivá právě tehdy, když α je nepravdivá, tj. $u(\neg\alpha) = 1$ právě tehdy, když $u(\alpha) = 0$;
2. $\alpha \wedge \beta$ je pravdivá právě tehdy, když α a β jsou obě pravdivá, tj. $u(\alpha \wedge \beta) = 1$ právě tehdy, když $u(\alpha) = u(\beta) = 1$;
3. $\alpha \vee \beta$ je nepravdivá právě tehdy, když α a β jsou obě nepravdivé, tj. $u(\alpha \vee \beta) = 0$ právě tehdy, když $u(\alpha) = u(\beta) = 0$;
4. $\alpha \Rightarrow \beta$ je nepravdivá právě tehdy, když α je pravdivá a β nepravdivá, tj. $u(\alpha \Rightarrow \beta) = 1$ právě tehdy, když $u(\alpha) = 1$ a $u(\beta) = 0$;
5. $\alpha \Leftrightarrow \beta$ je pravdivá právě tehdy, když buď obě formule α a β jsou pravdivé nebo obě jsou nepravdivé, tj. $u(\alpha \Leftrightarrow \beta) = 1$ právě tehdy, když $u(\alpha) = u(\beta)$.

10.1.5 Pravdivostní tabulky

Vlastnosti, které ohodnocení formulí musí mít, znázorňujeme též pomocí tzv. pravdivostních tabulek logických spojek. Jsou to:

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \Rightarrow \beta$	$\alpha \Leftrightarrow \beta$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

10.1.6 Věta

Každé zobrazení $u_0 : A \to \{0,1\}$ jednoznačně určuje ohodnocení $u : P(A) \to \{0,1\}$ takové, že $u_0(a) = u(a)$ pro všechna $a \in A$.

10.1.7 Důsledek

Dvě ohodnocení $u, v : P(A) \to \{0,1\}$ jsou shodná právě tehdy, když pro všechny logické proměnné $x \in A$ platí $u(x) = v(x)$.

10.1.8 Tautologie, kontradikce, splnitelné formule

Formule se nazývá *tautologie*, jestliže je pravdivá ve všech ohodnoceních formulí; nazývá se *kontradikce*, jestliže je nepravdivá ve všech ohodnoceních formulí. Formule je *splnitelná*, jestliže existuje aspoň jedno ohodnocení formulí, ve kterém je pravdivá.

Příklady:

1. Formule $\alpha \vee \neg\alpha$, $\alpha \Rightarrow \alpha$, $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ jsou tautologie.
2. Formule $a \vee b$, $(a \Rightarrow b) \Rightarrow a$ jsou splnitelné, ale ne tautologie.
3. Formule $\alpha \wedge \neg\alpha$ je kontradikce. Kontradikce je také každá negace tautologie.

10.1.9 Tautologická ekvivalence formulí

Řekneme, že formule φ a ψ jsou *tautologicky ekvivalentní* (také *sémanticky ekvivalentní*), jestliže pro každé ohodnocení u platí $u(\varphi) = u(\psi)$.

Tvrzení: Pro každé formule α, β, γ platí:

- $\alpha \models \alpha$,
- je-li $\alpha \models \beta$, pak i $\beta \models \alpha$,
- je-li $\alpha \models \beta$ a $\beta \models \gamma$, pak i $\alpha \models \gamma$.

Jsou-li $\alpha, \beta, \gamma, \delta$ formule splňující $\alpha \models \beta$ a $\gamma \models \delta$, pak platí

- $\neg\alpha \models \neg\beta$;
- $(\alpha \wedge \gamma) \models (\beta \wedge \delta)$, $(\alpha \vee \gamma) \models (\beta \vee \delta)$, $(\alpha \Rightarrow \gamma) \models (\beta \Rightarrow \delta)$, $(\alpha \Leftrightarrow \gamma) \models (\beta \Leftrightarrow \delta)$.

Příklad: Pro každou formuli α je formule $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ tautologie.

Ano, máme

$$\alpha \Rightarrow (\beta \Rightarrow \alpha) \;\models\; \neg\alpha \vee (\neg\beta \vee \alpha) \;\models\; (\neg\alpha \vee \alpha) \vee \neg\beta,$$

kde poslední formule je tautologie.

10.1.10 Tvrzení

Pro každé formule α, β, γ platí

- $\alpha \wedge \alpha \models \alpha$, $\alpha \vee \alpha \models \alpha$ (idempotence $\wedge$ a $\vee$);
- $\alpha \wedge \beta \models \beta \wedge \alpha$, $\alpha \vee \beta \models \beta \vee \alpha$ (komutativita $\wedge$ a $\vee$);

- $\alpha \wedge (\beta \wedge \gamma) \models (\alpha \wedge \beta) \wedge \gamma$, $\alpha \vee (\beta \vee \gamma) \models (\alpha \vee \beta) \vee \gamma$ (asociativita $\wedge$ a $\vee$);
- $\alpha \wedge (\beta \vee \alpha) \models \alpha$, $\alpha \vee (\beta \wedge \alpha) \models \alpha$ (absorpce $\wedge$ a $\vee$);
- $\neg\neg\alpha \models \alpha$;
- $(\alpha \Rightarrow \beta) \models (\neg\alpha \vee \beta)$.

Poznámka: Platí $\alpha \models \beta$ právě tehdy, když $\alpha \Leftrightarrow \beta$ je tautologie.

10.1.11 Další spojky

Každá formule s jednou (nebo žádnou) logickou proměnnou představuje zobrazení z množiny $\{0, 1\}$ do množiny $\{0, 1\}$. Existují čtyři taková zobrazení:

x	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1

Funkce f_1 je konstantní 0, jedná se o kontradikci a budeme ji značit **F**. Podobně funkce f_4 je tautologie (konstantní 1), značíme je **T**. Funkce f_2 je vlastně logická proměnná x a funkce f_3 je $\neg x$. Tedy nemáme další unární spojky.

10.1.12 NAND

Logická spojka $|$, nazývaná NAND (také Sheffův operátor), je definována

$$x \mid y \models \neg (x \wedge y) .$$

10.1.13 NOR

Logická spojka $\downarrow$, nazývaná NOR (také Peiceova šipka), je definována

$$x \downarrow y \models \neg (x \vee y) .$$

10.1.14 XOR

Lofická spojka $\bigoplus$, nazývaná XOR (také vylučovací nebo), je definována

$$x \bigoplus y \models \neg (x \Leftrightarrow y) .$$

10.1.15 CNF a DNF

Každé formuli o n logických proměnných odpovídá pravdivostní tabulka. Na tuto tabulku se můžeme dívat jako na zobrazení, které každé n-tici 0 a 1 přiřazuje 0 nebo 1. Ano, řádek pravdivostní tabulky je popsán n-ticí 0 a 1, hodnota je pak pravdivostní hodnota formule pro toto dosazení za logické proměnné. Zobrazení z množiny všech n-tic 0 a 1 do množiny $\{0,1\}$ se nazývá *Booleova funkce*. Naopak platí, že pro každou Booleovu funkci existuje formule, která této funkci odpovídá. Ukážeme v dalším, že dokonce můžeme volit formu ve speciálním tvaru, v tzv. *konjunktivním normálním tvaru* a *disjunktivním normálním tvaru*.

10.1.16 Booleova funkce

Booleovou funkcí n proměnných, kde n je přirozené číslo, rozumíme každé zobrazení $f : \{0,1\}^n \to \{0,1\}$, tj. zobrazení, které každé n-tici $(x_1, x_2, \ldots, x_n)$ nul a jedniček přiřazuje nulu nebo jedničku (označenou $f(x_1, x_2, \ldots, x_n)$).

10.1.17 Disjunktivní normální tvar

Literál je logická proměnná nebo negace logické proměnné. Řekneme, že formule je v *disjunktivním normálním tvaru*, zkráceně v *DNF*, jestliže je disjunkcí jedné nebo několika formulí, z nichž každá je literálem nebo konjunkcí literálů.

Poznamenejme, že literálu nebo konjunkci literálů se také říká *minterm*. Jestliže každý minterm obsahuje všechny proměnné, říkáme, že se jedná o *úplnou DNF*.

Věta: Ke každé Booleově funkci f existuje formule v *DNF* odpovídající *f*.

Důsledek: Ke každé formuli α existuje formule β, která je v DNF a navíc $\alpha \mid\!=\mid \beta$.

10.1.18 Konjunktivní normální tvar

Řekneme, že formule je v *konjunktivním normálním tvaru*, zkráceně v *CNF*, jestliže je konjunkcí jedné nebo několika formulí, z nichž každá je literálem nebo disjunkcí literálů.

Poznamenejme, že literálu nebo disjunkci literálů se také říká *maxterm* nebo *klausule*. Jestliže každá klausule obsahuje všechny proměnné, říkáme, že se jedná o *úplnou CNF*.

Věta: Ke každé Booleově funkci f existuje formule v *CNF* odpovídající *f*.

Důsledek: Ke každé formuli α existuje formule β, která je v CNF a navíc $\alpha \mid\!=\mid \beta$.

10.1.19 Booleovský kalkul

Víme, že pro pravdivostní ohodnocení formulí platí:

$$u(a \vee b) = \max\{u(a).u(b)\} = \max\{x, y\},$$

$$u(a \wedge b) = \min\{u(a), u(b)\} = \min\{x, y\},$$

$$u(\neg a) = 1 - u(a) = 1 - x.$$

kde $x = u(a)$, $y = u(b)$.

10.1.20 Booleovské operace

To motivuje zavedení booleovských operací (pro hodnoty 0,1):

$$\text{součin} \qquad x \cdot y = \min\{x, y\},$$

$$\text{logický součet} \quad x + y = \max\{x, y\},$$

$$\text{doplněk} \qquad \bar{x} = 1 - x.$$

Pro tyto operace platí řada rovností, tak, jak je známe z výrokové logiky:
Tvrzení: Pro všechna $x, y, z \in \{0, 1\}$ platí:

1. $x \cdot x = x$, $x + x = x$;
2. $x \cdot y = y \cdot x$, $x + y = y + x$;
3. $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, $x + (y + z) = (x + y) + z$;
4. $x \cdot (y + x) = x$, $x + (y \cdot x) = x$;
5. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$, $x + (y \cdot z) = (x + y) \cdot (x + z)$;
6. $\bar{\bar{x}} = x$;
7. $\overline{x + y} = \bar{x} \cdot \bar{y}$, $\overline{x \cdot y} = \bar{x} + \bar{y}$;
8. $x + \bar{x} = 1$, $x \cdot \bar{x} = 0$;
9. $x \cdot 0 = 0$, $x \cdot 1 = x$;
10. $x + 1 = 1$, $x + 0 = x$.

10.1.21 Booleovy funkce v DNF a CNF

Nyní můžeme pro Booleovu funkci psát pomocí výše uvedených operací, např.

$$f(x, y, z) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}z$$

a říkat, že jsme Booleovu funkci napsali v *disjunktivní normální formě*. Rovnost opravdu platí; dosadíme-li za logické proměnné jakékoli hodnoty, pak pravá strana rovnosti určuje hodnotu Booleovy funkce *f*. Obdobně jako jsme Booleovu funkci f napsali v disjunktivní normální formě, můžeme ji také napsat v *konjunktivní normální formě* a to takto:

$$f(x, y, z) = (\bar{x} + y + z)(\bar{x} + \bar{y} + z)(\bar{x} + \bar{y} + \bar{z}).$$

Věta: Každou Booleovu funkci lze napsat v disjunktivní normální formě i v konjunktivní normální formě.

10.1.22 Sémantický důsledek

10.1.22.1 Množina formulí pravdivá v ohodnocení

Řekneme, že množina formulí S je *pravdivá* v ohodnocení u, jestliže každá formule z S je pravdivá v u, tj. je-li $u(\varphi) = 1$ pro všechna $\varphi \in S$. Jinými slovy, množina formulí S je nepravdivá v ohodnocení u, jestliže existuje formule $\varphi \in S$, která je nepravdivá v ohodnocení u.

Fakt, že množina formulí S je pravdivá v ohodnocení u zapisujeme též $u(S) = 1$, fakt, že S je nepravdivá v u, zapisujeme také $u(S) = 0$.

Poznámka: Prázdná množina formulí je pravdivá v každém ohodnocení.

10.1.22.2 Splnitelná množina formulí

Řekneme, že množina formulí S je *splnitelná*, jestliže existuje pravdivostní ohodnocení u, v němž je S pravdivá. V opačném případě se množina S nazývá *nesplnitelná*.

Poznamenejme, že prázdná množina formulí je splnitelná.

10.1.22.3 Sémantický důsledek

Řekneme, že formule φ je *konsekventem*, též *sémantickým* nebo *tautologickým důsledkem* množiny formulí S, jestliže φ je pravdivá v každém ohodnocení u, v němž je pravdivá S.

Fakt, že formule φ je konsekventem množiny S, označujeme $S \models \varphi$. Je-li množina S prázdná, píšeme $\models \varphi$ místo $\emptyset \models \varphi$. Je-li množina S jednoprvková, tj. $S = \{\alpha\}$, píšeme $\alpha \models \varphi$ místo $\{\alpha\} \models \varphi$.

10.1.22.4 Příklady

Pro každé tři formule α, β, γ platí:

1. $\{\alpha, \alpha \Rightarrow \beta\} \models \beta$.
2. $\{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma\} \models (\alpha \Rightarrow \gamma)$.
3. $\{\alpha \Rightarrow \beta, \neg\beta\} \models \neg\alpha$.
4. $\{\alpha \vee \beta, \alpha \Rightarrow \gamma, \beta \Rightarrow \gamma\} \models \gamma$.

10.1.22.5 Tvrzení

1. Je-li S množina formulí a $\varphi \in S$, pak φ je konsekventem S, tj. $S \models \varphi$ pro každou $\varphi \in S$.
2. Tautologie je konsekventem každé množiny formulí S.
3. Formule φ je tautologie právě tehdy, když $\models \varphi$.
4. Každá formule je konsekventem nesplnitelné množiny formulí.

10.1.22.6 Poznámka

Uvědomme si, že $\alpha =| \beta$ právě tehdy, když platí současně $\alpha \models \beta$ a také $\beta \models \alpha$.

10.1.22.7 Tvrzení

Pro každé dvě formule α a β platí:
$\alpha \models \beta$ právě tehdy, když $\alpha \Rightarrow \beta$ je tautologie.

10.1.22.8 Věta

Pro množinu formulí S a formuli φ platí:
$S \models \varphi$ právě tehdy, když $S \cup \{\neg\varphi\}$ je nesplnitelná.

10.1.22.9 Věta o dedukci

Pro množinu formulí S a formule φ a ψ platí
$S \cup \{\varphi\} \models \psi$ právě tehdy, když $S \models (\varphi \Rightarrow \psi)$.

10.1.23 Rezoluční metoda ve výrokové logice

Rezoluční metoda rozhoduje, zda daná množina klausulí je splnitelná nebo je nesplnitelná. Tím je také "universální metodou" pro řešení základních problémů ve výrokové logice, neboť:

1. Daná formule φ je sémantickým důsledkem množiny formulí S právě tehdy, když množina $S \cup \{\neg\varphi\}$ je nesplnitelná.

2. Ke každé formuli α existuje množina klausulí S_α taková, že α je pravdivá v ohodnocení u právě tehdy, když v tomto ohodnocení je pravdivá množina S_α.

10.1.23.1 Klausule

Množinu všech logických proměnných označíme A. Připoměňme, že *literál* je buď logická proměnná (tzv. *positivní literál*) nebo negace logické proměnné (tzv. *negativní literál*). *Komplementární literály* jsou literály p a $\neg p$. *Klausule* je literál nebo disjunkce konečně mnoha literálů (tedy i žádného). Zvláštní místo mezi klausulemi zaujímá *prázdná klausule*, tj. klausule, která neobsahuje žádný literál a tudíž se jedná o kontradikci. Proto ji budeme označovat F.

Pro jednoduchost zavedeme následující konvenci: Máme dánu klausuli C a literál p, který se v C vyskytuje. Pak symbolem $C \setminus p$ označujeme klausuli, která obsahuje všechny literály jako C kromě p. Tedy např. je-li $C = \neg x \vee y \vee \neg z$, pak

$$C \setminus \neg z = \neg x \vee y.$$

10.1.23.2 Rezoloventa

Řekneme, že klausule D je rezolventou klausulí C_1 a C_2 právě tehdy, když existuje literál p takový, že p se vyskytuje v klausuli C_1, $\neg p$ se vyskytuje v klausuli C_2 a

$$D = (C_1 \setminus p) \vee (C_2 \setminus \neg p) .$$

Také říkáme, že klausule D je rezolventou C_1 a C_2 podle literálu p a značíme $D = res_p (C_1, C_2)$.

10.1.23.3 Tvrzení

Máme dány dvě klausule C_1, C_2 a označme D jejich rezolventu. Pak D je sémantický důsledek množiny $\{C_1, C_2\}$.

10.1.23.4 Tvrzení

Máme dánu množinu klausulí S a označme D rezolventu některých dvou klausulí z množiny S. Pak množiny S a $S \cup \{D\}$ jsou pravdivé ve stejných ohodnoceních.

10.1.23.5 Rezoluční princip

Označme
$R(S) = S \cup \{D | D$ je rezoloventa některých klausulí z $S\}$
$R^0 (S) = S$
$R^{i+1} (S) = R\left(R^i (S)\right)$ pro $i \in N$
$R^\star (S) = \bigcup \left\{R^i (S) \,|i \geq 0\right\}$.
Protože pro konečnou množinu logických proměnných existuje jen konečně mnoho klausulí, musí existovat n takové, že $R^n (S) = R^{n+1} (S)$. Pro toto n platí $R^n (S) = R^\star (S)$.

10.1.23.6 Věta (Rezoluční princip)

Množina klausulí S je splnitelná právě tehdy, když $R^\star (S)$ neobsahuje prázdnou klausuli F.

10.1.23.7 Základní postup

Předchozí věta dává návod, jak zjistit, zda daná množina klausulí je spnitelná nebo je nesplnitelná:

1. Formule množiny M převedeme do CNF a M pak nahradíme množinou S všech klausulí vyskytujících se v některé formuli v CNF. Klausule, které jsou tautologiemi, vynecháme. Jestliže nám nezbyde žádná klausule, množina M se skládala z tautologií a je pravdivá v každém pravdivostním ohodnocení.

2. Vytvoříme $R^\star (S)$.

3. Obsahuje-li $R^\star(S)$ prázdnou klausuli, je množina S (a tedy i množina M) nesplnitelná, v opačném případě je M splnitelná.

Je zřejmé, že konstrukce celé množiny $R^\star(S)$ může být zbytečná — stačí pouze zjistit, zda $R^\star(S)$ obsahuje F.

10.1.23.8 Výhodnější postup

Existuje ještě jeden postup, který usnadní práci s použitím rezoluční metody. Ten nejenom že nám odpoví na otázku, zda konečná množina klausulí S je splnitelná nebo nesplnitelná, ale dokonce nám umožní v případě splnitelnosti sestrojit aspoň jedno pravdivostní ohodnocení, v němž je množina S pravdivá.

Máme konečnou množinu klausulí S, kde žádná klausule není tautologií. Zvolíme jednu logickou proměnnou (označme ji x), která se v některé z klausulí z S vyskytuje. Najdeme množinu klausulí S_1 s těmito vlastnostmi:

1. Žádná klausule v S_1 neobsahuje logickou proměnnou x.
2. Množina S_1 je splnitelná právě tehdy, když je splnitelná původní množina S.

Množinu S_1 vytvoříme takto: Rozdělíme klausule množiny S do tří skupin: M_0 se skládá ze všech klausulí množiny S, které neobsahují logickou proměnnou x.

M_x se skládá ze všech klausulí množiny S, které obsahují positivní literál x.

$M_{\neg x}$ se skládá ze všech klausulí množiny S, které obsahují negativní literál $\neg x$.

Označme N množinu všech rezolvent klausulí množiny S podle literálu x; tj. rezolvent vždy jedné klausule z množiny M_x s jednou klausulí z množiny $M_{\neg x}$. Všechny tautologie vyřadíme.

Položíme $S_1 = M_0 \cup N$.

Tvrzení: Množina klausulí S_1 zkonstruovaná výše je splnitelná právě tehdy, když je splnitelná množina S.

Dostali jsme tedy množinu klausulí S_1, která již neobsahuje logickou proměnnou x a je splnitelná právě tehdy, když je splnitelná množina S. Navíc, množina S_1 má o jednu logickou proměnnou méně než množina S.

Nyní opakujeme postup pro množinu S_1. Postup skončí jedním ze dvou možných způsobů:

1. Při vytváření rezolvent dostaneme prázdnou kalusuli F. Tedy S je nesplnitelná.
2. Dostaneme prázdnou množinu klausulí. V tomto případě je množina S splnitelná.

10.2 Predikátová logika

10.2.1 Syntaxe predikátové logiky

Nejprve zavedeme syntaxi predikátové logiky, tj. uvedeme pravidla, podle nichž se tvoří syntakticky správné formule predikátové logiky. Význam a pravdivostní hodnota nás bude zajímat až dále.

Správně utvořené formule budou řetězce (posloupnosti) symbolů tzv. *jazyka predikátové logiky*.

10.2.1.1 Jazyk predikátové logiky $\mathcal{L}$

Jazyk predikátové logiky se skládá z

1. *logických symbolů*, tj.:
   a) spočetné množiny individuálních proměnných: $Var = \{x, y, \ldots, x_1, x_2, \ldots\}$
   b) výrokových logických spojek: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
   c) obecného kvantifikátoru $\forall$ a existenčního kvantifikátoru $\exists$
2. *speciálních symbolů*, tj.:
   a) množiny $Pred$ predikátových symbolů (nesmí být prázdná)
   b) množiny $Kons$ konstantních symbolů (může být prázdná)
   c) množiny $Func$ funkčních symbolů (může být prázdná)
3. pomocných symbolů, jako jsou závorky "$(, [,),]$" a čárka ",".

Pro každý predikátový i funkční symbol máme dáno přirozené číslo n kolika objektů se daný predikát týká, nebo kolika proměnných je daný funkční symbol. Tomuto číslu říkáme *arita* nebo též *četnost* predikátového symbolu nebo funkčního symbolu. Funkční symboly mají aritu větší nebo rovnu 1, predikátové symboly připouštíme i arity 0.

10.2.1.2 Poznámka

Predikátové symboly budeme většinou značit velkými písmeny, tj. např. $P, Q, R, \ldots, P_1, P_2, \ldots$; konstantní symboly malými písmeny ze začátku abecedy, tj. $a, b, c, \ldots, a_1, a_2, \ldots$, a funkční symboly většinou $f, g, h, \ldots, f_1, f_2, \ldots$. Formule predikátové logiky budeme označovat malými řeckými písmeny (obdobně, jako jsme to dělali pro výrokové formule). Kdykoli se od těchto konvencí odchýlíme, tak v textu na to upozorníme.

Poznamejme, že přestože často budeme mluvit o n-árních predikátových symbolech a n-árních funkčních symbolech, v běžné praxi se setkáme jak s predikáty, tak funkcemi arity nejvýše tři. Nejběžnější jsou predikáty a funkční symboly arity 1, těm říkáme též *unární*, nebo arity 2, těm říkáme též *binární*.

Predikátové symboly arity 0 představují nestrukturované výroky (netýkají se žádného objektu). Tímto způsobem se v predikátové logice dá popsat i výrok: „Prší".

10.2.1.3 Termy

Množina *termů* je definována těmito pravidly:

1. Každá proměnná a každý konstantní symbol je term.

2. Jestliže f je funkční symbol arity n a $t_1, t_2, \ldots, t_n$ jsou termy, pak $f(t_1, t_2, \ldots, t_n)$ je také term.

3. Nic, co nevzniklo konečným použitím pravidel 1 a 2, není term.

Poznámka: Term je zhruba řečeno objekt, pouze může být složitěji popsán než jen proměnnou nebo konstantou. V jazyce predikátové logiky termy vystupují jako „podstatná jména".

10.2.1.4 Atomické formule

Atomická formule je predikátový symbol P aplikovaný na tolik termů, kolik je jeho arita. Jinými slovy, pro každý predikátový symbol $P \in Pred$ arity n a pro každou n-tici termů $t_1, t_2, \ldots, t_n$ je $P(t_1, t_2, \ldots, t_n)$ atomická formule.

10.2.1.5 Formule

Množina *formulí* je definována těmito pravidly:

1. Každá atomická formule je formule.

2. Jsou-li φ a ψ dvě formule, pak $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$ jsou opět formule.

3. Je-li φ formule a x proměnná, pak $(\forall x\varphi)$ a $(\exists x\varphi)$ jsou opět formule.

4. Nic, co nevzniklo pomocí konečně mnoha použití bodů 1 až 3, není formule.

10.2.1.6 Konvence

1. Úplně vnější závorky nepíšeme. Píšeme tedy např. $(\exists x P(x)) \vee R(a, b)$ místo $((\exists x P(x)) \vee R(a, b))$.

2. Spojka „negace" má vždy přednost před výrokovými logickými spojkami a proto píšeme např. $\forall x(\neg P(x) \Rightarrow Q(x))$ místo $\forall x((\neg P(x)) \Rightarrow Q(x))$.

10.2.1.7 Syntaktický strom formule

Ke každé formuli predikátové logiky můžeme přiřadit její *syntaktický strom* (též *derivační strom*) podobným způsobem jako jsme to udělali v případě výrokových formulí. Rozdíl je v tom, že kvantifikátory považujeme za unární (tj. mají pouze jednoho následníka) a také pro termy vytváříme jejich syntaktický strom. Listy syntaktického stromu jsou vždy ohodnoceny buď proměnnou nebo konstantou.

10.2.1.8 Podformule

Podformule formule φ je libovolný podřetězec φ, který je sám formulí. Jinými slovy: Podformule formule φ je každý řetězec odpovídající podstromu syntaktického stromu formule φ, určenému vrcholem ohodnoceným predikátovým symbolem, logickou spojkou nebo kvantifikátorem.

10.2.1.9 Volný a vázaný výskyt proměnné

Máme formuli φ a její syntaktický strom. List syntaktického stromu obsazený proměnnou x je výskyt proměnné x ve formuli φ. Výskyt proměnné x je *vázaný* ve formuli φ, jestliže při postupu od listu ohodnoceného tímto x ve směru ke kořeni syntaktického stromu narazíme na kvantifikátor s touto proměnnou. V opačném případě mluvíme o *volném* výskytu proměnné x.

10.2.1.10 Sentence

Formule, která má pouze vázané výskyty proměnné, se nazývá *sentence*, též *uzavřená formule*. Formuli, která má pouze volné výskyty proměnné, se říká *otevřená formule*.

10.2.1.11 Legální přejmenování proměnné

Přejmenování výskytů proměnné x ve formuli φ je legálním přejmenováním proměnné, jestliže

- jedná se o výskyt vázané proměnné ve φ;
- přejmenováváme všechny výskyty x vázané daným kvantifikátorem;
- po přejmenování se žádný dříve volný výskyt proměnné nesmí stát vázaným výskytem.

10.2.1.12 Rovnost formulí

Dvě formule považujeme za *stejné*, jestliže se liší pouze legálním přejmenováním vázaných proměnných.

Každou formuli φ lze napsat tak, že každá proměnná má ve formuli buď jen volné výskyty nebo jen vázané výskyty.

10.2.2 Sémantika predikátové logiky

Nyní se budeme zabývat sémantikou formulí, tj. jejich významem a pravdivostí.

10.2.2.1 Intepretace jazyka predikátové logiky

Interpretace predikátové logiky s predikátovými symboly $Pred$, konstantními symboly $Kons$ a funkčními symboly $Func$ je dvojice $\langle U, [-] \rangle$, kde

- U je neprázdná množina nazývaná *universum*;

- $\langle [-] \rangle$ je přiřazení, které
  1. každému predikátovému symbolu $P \in Pred$ arity n přiřazuje podmnožinu $[P]$ množiny U^n, tj. n-ární relaci na množině U.
  2. každému konstantnímu symbolu $a \in Kons$ přiřazuje prvek z U, značíme jej $[a]$,
  3. každému funkčnímu symbolu $f \in Func$ arity n přiřazuje zobrazení množiny U^n do U, značíme je $[f]$,

Množina U se někdy nazývá *domain* a označuje D.

10.2.2.2 Kontext proměnných

Je dána interpretace $\langle U, [-] \rangle$. Kontext proměnných je zobrazení ρ, které každé proměnné $x \in Var$ přiřadí prvek $\rho(x) \in U$. Je-li ρ kontext proměnných, $x \in Var$ a $d \in U$, pak

$$p[x := d]$$

označuje kontext proměnných, který má stejné hodnoty jako ρ, a liší se pouze v proměnné x, kde má hodnotu d. Kontextu proměnných $p[x := d]$ též říkáme *update* kontextu ρ o hodnotu d v x.

10.2.2.3 Interpretace termů při daném kontextu proměnných

Je dána interpretace $\langle U, [-] \rangle$ a kontext proměnných ρ. Pak termy interpretujeme následujícím způsobem.

1. Je-li term konstatní symbol $a \in Kons$, pak jeho hodnota je prvek $[a]_\rho = [a]$. Je-li term proměnná x, pak jeho hodnota je $[x]_\rho = \rho(x)$.
2. Je-li $f(t_1, t_2, \ldots, t_n)$ term. pak jeho hodnota je

$$[f(t_1, t_2, \ldots, t_n)]_\rho = [f]\left([t_1]_\rho, \ldots, [t_n]_\rho\right).$$

   Jinými slovy, hodnota termu $f(t_1, t_2, \ldots, t_n)$ je funkční hodnota funkce $[f]$ provedené na n-tici prvků $[t_1]_\rho, \ldots, [t_n]_\rho$ z U.

10.2.2.4 Pravdivostní hodnota formule v dané interpretaci a daném kontextu

Nejprve definujeme *pravdivost* formulí v dané interpretaci $\langle U, [-] \rangle$ při daném kontextu proměnných ρ:

1. Nechť φ je atomická formule. Tj. $\varphi = P(t_1, t_2, \ldots, t_n)$, kde P je predikátový symbol arity n a $t_1, t_2, \ldots, t_n$ jsou termy. Pak φ je pravdivá v interpretaci $\langle U, [-] \rangle$ a kontextu ρ právě tehdy, když

$$\left(\left[t_1\right]_\rho, \ldots, \left[t_n\right]_\rho\right) \in [P]\,.$$

Jinými slovy: φ je v naší interpretaci pravdivá právě tehdy, když n-tice hodnot termů $\left(\left[t_1\right]_\rho, \ldots, \left[t_n\right]_\rho\right)$ má vlastnost $[P\,]$.

2. Jsou-li φ a ψ formule, jejichž pravdivost v interpretaci $\langle U, [-]\rangle$ a kontextu ρ již známe, pak
   - $\neg\varphi$ je pravdivá právě tehdy, když φ není pravdivá.
   - $\varphi \wedge \psi$ je pravdivá právě tehdy, když φ a ψ jsou pravdivé.
   - $\varphi \vee \psi$ je nepravdivá právě tehdy, když φ i ψ jsou nepravdivé.
   - $\varphi \Rightarrow \psi$ je nepravdivá právě tehdy, když φ je pravdivá a ψ je nepravdivá.
   - $\varphi \Leftrightarrow \psi$ je pravdivá právě tehdy, když buď obě formule φ a ψ jsou pravdivé, nebo obě formule φ a ψ jsou nepravdivé.
3. Je-li φ formule a x proměnná, pak
   - $\forall x \varphi\,(x)$ je pravdivá právě tehdy, když fromule φ je pravdivá *v každém* kontextu $p\,[x := d]$, kde d je prvek U.
   - $\exists x \varphi\,(x)$ je pravdivá právě tehdy, když fromule φ je pravdivá *v aspoň jednom* kontextu $p\,[x := d]$, kde d je prvek U.

10.2.2.5 Pravdivostní hodnota sentence

Sentence φ je *pravdivá v interpretaci* $\langle U, [-]\rangle$ právě tehdy, když je pravdivá v každém kontextu proměnných ρ.

Poznamenejme, že pro sentence v předchozí definici jsme mohli požadovat pravdivost v alespoň jednom kontextu.

10.2.2.6 Model sentence

Interpretace $\langle U, [-]\rangle$, ve které je sentence φ pravdivá, se nazývá *model sentence* φ.

10.2.2.7 Tautologie, kontradikce, splnitelná sentence

Sentence φ se nazývá *tautologie*, jestliže je pravdivá v každé interpretaci. Sentence se nazývá *kontradikce*, jestliže je nepravdivá v každé interpretaci. Nazývá se *splnitelná*, jestliže je pravdivá v aspoň jedné interpretaci.

Také jsme mohli formulovat předchozí definice pomocí pojmu „model". Tautologie je sentence, pro kterou je každá interpretace jejím modelem; sentence je splnitelná, má-li model; sentence je kontradikce, nemá-li model.

Následující sentence jsou tautologie. (P je unární predikátový symbol, Q je binární predikátový symbol a a je konstantní symbol.)

1. $(\forall x P(x)) \Rightarrow P(a)$;
2. $P(a) \Rightarrow (\exists x P(x))$;

Následující sentence jsou splnitelné formule:

1. $\forall x \exists y Q(x, y)$,
2. $\forall x \forall y (x + y = y + x)$,

kde Q = jsou binární predikátové symboly, $+$ je binární funkční symbol.

Zvláštní příklady kontradikcí neuvádíme. Kontradikce jsou přesně ty formule, jejichž negace je tautologie. Tak např. formule $(\forall x P(x) \wedge \neg(\forall x P(x)))$ je kontradikce. Je dobré si uvědomit, že jde o „dosazení" formule $\forall x P(x)$ do výrokové kontradikce $p \wedge \neg p$.

10.2.2.8 Splnitelné množiny sentencí

Množina sentencí M je *splnitelná* právě tehdy, když existuje interpretace $\langle U, [-] \rangle$, v níž jsou všechny sentence z M pravdivé. Takové interpretaci pak říkáme *model* množiny sentencí M.

Množina sentencí M je *nesplnitelná*, jestliže ke každé interpretaci $\langle U, [-] \rangle$ existuje formule z M, která je v $\langle U, [-] \rangle$ nepravdivá.

Z poslední definice vyplývá, že prázdná množina sentencí je splnitelná.

10.2.3 Tautologická ekvivalence

10.2.3.1 Tautologická ekvivalence sentencí

ˇekneme, že dvě sentence φ a ψ jsou *tautologicky ekvivalentní* právě tehdy, když mají stejné modely, tj. jsou pravdivé ve stejných interpretacích. Jinými slovy, mají stejnou pravdivostní hodnotu ve všech interpretacích.

Někdy se říká, že sentence jsou *sémanticky* ekvivalentní místo, že jsou tautologicky ekvivalentní.

Poznámka: Dá se jednoduše dokázat, že tautologická ekvivalence je relace ekvivalence na množině všech sentencí daného jazyka $\mathcal{L}$ a že má podobné vlastnosti jako tautologická ekvivalence formulí výrokové logiky.

10.2.3.2 Tvrzení

Nechť φ a ψ jsou sentence. Pak platí:

$\varphi \mid\!=\mid \psi$ právě tehdy, když $\varphi \Leftrightarrow \psi$ je tautologie.

Tautologické ekvivalence: (P a Q jsou unární predikátové symboly.)

1. $\neg(\forall x P(x)) \mid\!=\mid (\exists x \neg P(x))$,
2. $\neg(\exists x P(x)) \mid\!=\mid (\forall x \neg P(x))$.
3. $(\forall x P(x)) \wedge (\forall x Q(x)) \mid\!=\mid \forall x (P(x) \wedge Q(x))$;

10.2.4 Sémantický důsledek

10.2.4.1 Sémantický důsledek

Řekneme, že sentence φ je *sémantickým důsledkem*, též *konsekventem* množiny sentencí S právě tehdy, když každý model množiny S je také modelem sentence φ. Tento fakt značíme

$$S \models \varphi.$$

Můžeme též říci, že sentence φ *není* konsekventem množiny sentencí S, jestliže existuje model množiny S, který není modelem sentence φ. To znamená, že existuje interpretace $\langle U, [-]\rangle$, v níž je pravdivá každá sentence z množiny S a není pravdivá formule φ. Jedná se tedy o obdobný pojem jako ve výrokové logice, pouze místo o pravdivostním ohodnocení mluvíme o interpretaci.

10.2.4.2 Konvence

Jestliže množina sentencí S je jednoprvková, tj. $S = \{\psi\}$, pak píšeme $\psi \models \varphi$ místo $\{\psi\} \models \varphi$. Je-li množina S prázdná, píšeme $\models \varphi$ místo $\emptyset \models \varphi$.

Obdobně jako pro výrokovou logiku, dostáváme řadu jednoduchých pozorování. Pro množiny sentencí M, N a sentenci φ platí:

1. Je-li $\varphi \in M$, je $M \models \varphi$.
2. Je-li $N \subseteq M$ a $N \models \varphi$, je i $M \models \varphi$.
3. Je-li φ tautologie, pak $M \models \varphi$ pro každou množinu sentencí M.
4. Je-li $\models \varphi$, pak φ je tautologie.
5. Je-li M nesplnitelná množina, pak $M \models \varphi$ pro každou sentenci φ.

10.2.4.3 Tvrzení

Nechť φ a ψ jsou sentence. Pak platí:
$\varphi \models\models \psi$ právě tehdy, když $\varphi \models \psi$ a $\psi \models \varphi$.

10.2.4.4 Tvrzení

Nechť φ a ψ jsou sentence. Pak platí:
$\varphi \models \psi$ právě tehdy, když $\varphi \Rightarrow \psi$ je tautologie.

10.2.4.5 Věta

Pro každou množinu sentencí S a každou sentenci φ platí:
$S \models \varphi$ právě tehdy, když $S \cup \{\neg\varphi\}$ je nesplnitelná množina.

10.2.5 Rezoluční metoda v predikátové logice

Rezoluční metoda v predikátové logice je obdobná stejnojmenné metodě ve výrokové logice. Ovšem vzhledem k bohatší vnitřní struktuře formulí predikátové logiky je složitější. Používá se v logickém programování a je základem programovacího jazyka Prolog.

Nejprve zavedeme literály a klausule v predikátové logice.

10.2.5.1 Literál

Literál je atomická formule (tzv. *pozitivní literál*), nebo negace atomické formule (tzv. *negativní literál*). *Komplementární literály* jsou dva literály, z nichž jeden je negací druhého.

10.2.5.2 Klausule

Klausule je sentence taková, že všechny kvantifikátory jsou obecné a stojí na začátku sentence (na jejich pořadí nezáleží) a za nimi následují literál nebo disjunkce literálů.

Ve výrokové logice jsme pro každou formuli α našli k ní tautologicky ekvivalentní množinu klausulí S_α a to tak, že α i S_α byly pravdivé ve stejných pravdivostních ohodnocení. Takto jednoduchá situace v predikátové logice není. Ukážeme si, jak k dané sentenci φ najít množinu klausulí S_φ a to tak, že φ je splnitelná právě tehdy, když množina S_φ je splnitelná.

10.2.5.3 Rezoloventy klausulí

Ve výrokové logice jsme rezolventy vytvářeli tak, že jsme si vždy vzali dvě klausule, které obsahovaly dvojici komplementárních literálů, a výsledná rezolventa byla disjunkcí všech ostatních literálů z obou klausulí. Situace v predikátové logice je složitější. Postup, jak vytváříme rezolventy v predikátové logice, si ukážeme na příkladech.

Poznámka: Ne vždy rezolventa existuje.

10.2.5.4 Příklad

Najděme rezoloventu klausulí $K_1 = \forall x \forall y \left(P\left(x\right) \vee \neg Q\left(x, y\right)\right)$ a $K_2 = \forall x \forall y \left(Q\left(x, y\right) \vee R\left(y\right)\right)$, kde P a R jsou unární predikátové symboly a Q je binární predikátový symbol, x, y jsou proměnné.

Klausule K_1 a K_2 obsahují dvojici komplementárních literálů, totiž $\neg Q\left(x, y\right)$ je literál K_1 a $Q\left(x, y\right)$ je literál K_2. Rezoloventou klausulí K_1 a K_2 je tedy $K = \forall x \forall y \left(P\left(x\right) \vee R\left(y\right)\right)$.

10.2.5.5 Unifikační algoritmus

Vstup: Dva positivní literály L_1, L_2, které nemají společné proměnné.

Výstup: Hlášení neexistuje v případě, že hledaná substituce neexistuje, v opačném případě substituce ve tvaru množiny prvků tvaru x/t, kde x je proměnná, za kterou se dosazuje, a t je term, který se za proměnnou x dosazuje.

1. Položme $E_1 := L_1, E_2 := L_2, \theta := \emptyset$

2. Jsou-li E_1, E_2 prázdné řetězce, stop. Množina θ určuje hledanou substituci. V opačném případě položíme $E_1 := E_1\theta$, $E_2 := E_2\theta$ (tj. na E_1, E_2 provedeme substituci θ).

3. Označíme X první symbol řetězce E_1, Y první symbol řetězce E_2.

4. Je-li $X = Y$, odstraníme X a Y z počátku E_1 a E_2. Jsou-li X a Y predikátové nebo funkční symboly, odstraníme i jim příslušné závorky a jdeme na krok 2.

5. Je-li X proměnná, neděláme nic.

   Je-li Y proměnná (a X nikoli), přehodíme E_1, E_2 a X, Y .

   Není-li ani X ani Y proměnná, stop. Výstup neexistuje.

6. Je-li Y proměnná nebo konstanta, položíme $\theta := \theta \cup \{X/Y\}$. Odstraníme X a Y ze začátků řetězců E_1 a E_2 (spolu s čárkami, je-li třeba) a jdeme na krok 2.

7. Je-li Y funkční symbol, označíme Z výraz skládající se z Y a všech jeho argumentů (včetně závorek a čárek). Jestliže Z obsahuje X, stop, výstup neexistuje.

   V opačném případě položíme $\theta := \theta \cup \{X/Z\}$, odstraníme X a Z ze začátků E_1 a E_2 (odstraníme čárky, je-li třeba) a jdeme na krok 2.

10.2.5.6 Rezoluční princip

Je obdobný jako rezoluční princip ve výrokové logice:
Je dána množina klausulí S. Označme
$R(S) = S \cup \{K | K$ je nejobecnější rezoloventa některých klausulí z $S\}$
$R^0(S) = S$
$R^{i+1}(S) = R\left(R^i(S)\right)$ pro $i \in \mathbb{N}$
$R^\star(S) = \bigcup \left\{R^i(S) \,|i \geq 0\right\}$.
Množina klausulí S je splnitelná právě tehdy, když $R^\star(S)$ neobsahuje prázdnou klausuli F.

Jestliže je množina S konečná, existuje přirozené číslo n_0 takové, že $R^{n_0}(S) = R^{n_0 1}$. Pak $R^\star(S) = R^{n_0}(S)$.

1 Orientované a neorientované grafy, souvislost, silná souvislost, stromy a kostry, Eulerovy grafy, Hamiltonovy grafy, nezávislé množiny, barvení grafu. (A0B01LGR)

1.1 Definice orientovaného grafu

Orientovaný graf je trojice G=(V,E,ϵ), kde V je konečná množina vrcholů (též zvaných uzlů), E je konečná množina jmen hran (též nazývaných orientovaných hran) a \epsilon) je přiřazení, které každé hraně e \epsilon E přiřazuje uspořádanou dvojici vrcholů a nazývá se vztah incidence.

Jestliže ε(e)=(u,v) pro $u, v \in V$, říkáme, že vrchol u je počáteční vrchol hrany e a vrchol v je koncový vrchol hrany e; značíme PV(e)=u a KV(e)=v. O vrcholech u,v říkáme, že jsou krajní vrcholy hrany e, též že jsou incidentní s hranou e. Jestliže počátční vrchol a koncový vrchol jsou stejné, říkáme, že hrana e je orientovaná smyčka.

1.2 Definice neorientovaného grafu

Nerientovaný graf je trojice G=(V,E,ε), kde V je konečná množina vrcholů (též zvaných uzlů), E je konečná množina jmen hran a ε je přiřazení, které každé hraně $e \in E$ přiřazuje množinu {u,v} pro vrcholy $u, v \in V$ a nazývá se vztah incidence. Jestliže ε(e)={u,v} pro u,v ϵ V, říkáme, že u,v jsou krajní vrcholy hrany e, též že jsou incidentní s hranou e. Je-li u=v, říkáme že e je (neorientovaná) smyčka.

1.3 Stromy

Orientovaný nebo neorientovaný graf se nazývá strom, je-li souvislý a neobsahuje-li kružnici

V každém stromu s alespoň dvěma vrcholy existuje vrchol stupně 1.

Každý strom o n vrcholech má n-1 hran.

Poznámka Mějme souvislý graf G. Přidáme-li k němu hranu (aniž bychom zvětšili množinu vrcholů), zůstane graf souvislý.

Mějme graf G bez kružnic. Odebereme-li v grafu G hranu, vzniklý graf opět nebude obsahovat kružnici.

Strom je graf, který má nejmenší počet hran aby mohl být souvislý a současně největší počet hran aby v něm neexistovala kružnice.

Tvrzení Je dán graf G, pak následující je ekvivalentní

1. G je strom
2. Graf G nemá kružnice a přidáme-li ke grafu libovolnou hranu uzavřeme přesně jednu kružnici.
3. Graf G je souvislý a odebráním libovolné hrany přestane být souvislý.

Poznamenejme, že přidáním hrany zde rozumíme přidání hrany mezi již existující vrcholy (další vrcholy nepřidáváme)

1

1.4 Souvislost

1.4.1 Souvislé grafy

Řekneme, že graf je souvislý, jestliže pro každé dva vrcholy u, v grafu existuje neorientovaná cesta z u do v. Poznamenejme, že vždy existuje cesta z vrcholu u do sebe, totiž triviální cesta. Také platí, že neorientovaná cesta z vrcholu u do vrcholu v je také neorientovanou cestou z v do u.

1.4.2 Komponenty souvislosti

Máme dán graf G. Komponenta souvislosti (někdy též komponenta slabé souvislosti) je maximální množina vrcholů A taková, že indukovaný podgraf určený A je souvislý.

Maximální množinou zde rozumíme takovou množinu A, pro kterou platí, že přidáme-li k množině A libovolný vrchol, podgraf indukovaný touto větší množinou už souvislý nebude.

2

1.5 Silná souvislost

1.5.1 Silně souvislé grafy

Řekneme, že orientovaný graf G je silně souvislý, jestliže pro každé dva vrcholy u, v existuje orientovaná cesta z vrcholu u do vrcholu v a orientovaná cesta z vrcholu v do vrcholu u.

[1] //see also: podgrafy
[2] //see also: paralelní hrany, prostý graf, stupně vrcholů, matice incidence, sled, tah a cesta

Poznámka V definici silně souvislého grafu jsme mohli požadovat pouze existenci orientované cesty z vrcholu u do vrcholu v. Je to proto, že existenci takové cesty vyžadujeme pro všechny dvojice vrcholů, tedy i pro dvojici v, u. Dále si uvědomte, že vždy existuje orientovaná cesta z vrcholu do sebe – je to triviální cesta.

Souvislý graf je silně souvislý právě tehdy, když každá hrana leží v nějakém cyklu.
[3]

1.6 Minimální kostra

1.6.1 Kostra grafu

Je dán souvislý graf G. Faktor grafu G, který je stromem, se nazývá kostra grafu G. Připomeňme, že faktor grafu G je podgraf grafu G, který má stejnou množinu vrcholů jako G.

1.6.2 Minimální kostra

Je dán souvislý graf G spolu s ohodnocením hran c, tj. pro každou hranu $e \in E(G)$ je dáno číslo c(e) (číslo c(e) nazýváme cenou hrany e).

Minimální kostra grafu G=(V,E) je taková kostra grafu K=(V,L), že $\sum e \in Lc(e)$ je nejmenší (mezi všemi kostrami grafu G).

1.6.2.1 Tvrzení

V každém souvislém ohodnoceném grafu existuje minimální kostra. Nemusí však být jediná. Obecný postup pro hledání minimální kostry

1.6.3 Obecný postup pro hledání minimální kostry

Je dán souvislý graf G=(V,E) a ohodnocení hran c.

1. Na začátku máme L=0. Označíme S množinu všech komponent souvislosti grafu K=(V,L); tj. na začátku je $s = v; v \in V$.
2. Dokud není graf K=(V,L) souvislý (tj. dokud S se neskládá z jediné množiny), vybereme hranu e podle následujících pravidel:
   a) E spojuje dvě různé komponenty souvislosti S,S‘ grafu K (tj. dvě množiny z S)
   b) A pro S nebo S‘ je nejlevnější hranou která vede z komponenty ven

   Hranu e přidáme do množiny L a množiny S a S‘ nahradíme jejich sjednocením.
3. Postup ukončíme, jestliže jsme přidali n-1 hran (tj. jestliže se S skládá z jediné množiny).

[3] //See also: Silně souvislé komponenty, kondenzace grafu, hledání silně souvislých komponent, Tarjanův algoritmus pro nalezení silně souvislých komponent

1.6.4 Kruskalův algoritmus

Jedná se o modifikaci obecného postupu pro hledání minimální kostry:

1. Setřídíme hrany podle ceny do neklesající posloupnosti, tj. $c(e1) \leq c(e2) \leq \ldots . \leq c(em)$. Položíme L=0, $S = v; v \in V$.
2. Probíráme hrany v daném pořadí. Hranu ei přidáme do L, jestliže má oba krajní vrcholy v různých množinách $S, S' \in S$. V S množiny S a S‘ nahradíme jejich sjednocením. V opačném případě hranu přeskočíme.
3. Algoritmus končí, jestliže jsme přidali n-1 hran (tj. S se skládá z jediné množiny).

1.6.5 Primův algoritmus

Jedná se o modifikaci obecného postupu pro hledání minimální kostry:

1. Vybereme libovolný vrchol v. Položíme L=0, S={v}.
2. Vybereme nejlevnější hranu e, která spojuje některý vrchol x z množiny S s vrcholem y, který v S neleží. Vrchol y přidáme do množiny S a hranu e přidáme do L.
3. Opakujeme krok 2 dokud nejsou všechny vrcholy v množině S.

1.6.6 Jádro grafu

Podmnožina vrcholů K orientovaného grafu G se nazývá jádro grafu, jestliže splňuje následující podmínky:

1. Pro každou hranu e s počátečním vrcholem PV(e) є K platí KV(e) (NENÍ)є K. (Neexistuje hrana, která by vedla z množiny K do sebe)
2. Pro každý vrchol v, který neleží v K, existuje hrana e s PV(e)=v a KV(e) є K. (z každého vrcholu, který leží mimo K, se můžeme dostat po hraně zpět do K)

[4]

1.7 Eulerovy grafy

Tah je sled, ve kterém se neopakují hrany. Jinými slovy, tah obsahuje hrany grafu vždy nejvýše jedenkrát.

1.7.1 Eulerovské tahy

Tah v grafu se nazývá eulerovský, jestliže prochází každou hranou; jinými slovy, obsahuje-li každou hranu přesně jedenkrát. Eulerovské tahy se dělí na uzavřené a otevřené, orientované a neorientované.

[4] //See also: Kořenové stromy, kořen, následník, předchůdce a list, výška kořenového stromu, binární kořenové stromy, halda

1.7.2 Eulerův graf

Graf G se nazývá eulerovský graf, jestliže v něm existuje uzavřený eulerovský tah. V případě, že graf G je orientovaný, požadujeme existenci orientovaného uzavřeného eulerovského tahu.

Aplikace eulerovských tahů

- Kreslení s co nejmenším počtem tahů
- Úloha čínského pošťáka
- De Bruijnova posloupnost

V silně souvisém orientovaném grafu existuje uzavřený orientovaný eulerovský tah právě tehdy, když pro každý vrchol v grafu platí $d-(v) = d+(v)$ (Tj. v každém vrcholu končí stejný počet hran jako v něm začíná).

V souvislém grafu existuje uzavřený neorientovaný eulerovský tah právě tehdy, když každý vrchol má sudý stupeň.

1.7.3 Postup na hledání uzavřeného orientovaného eulerovského tahu

Vybereme libovolný vrchol v grafu. Protože graf je souvislý, v každém vrcholu začíná i končí alespoň jedna hrana. Z vrcholu v vytváříme náhodně orientovaný tah; tj. procházíme hrany tak, abychom žádnou hranou neprošli dvakrát. Takto pokračujeme, dokud je to možné, tj. dokud se nevrátíme do výchozího vrcholu v a ve vrcholu v již nezačíná žádná dosud nepoužitá hrana. Tím jsme dostali uzavřený tah. Jestliže tento tah obsahuje všechny hrany, je to hledaný uzavřený eulerovský tah. Neobsahuje-li takto zkonstruovaný tah všechny hrany, pak na tahu existuje vrchol w takový, že v něm začíná nepoužitý hrana. (To vyplývá ze souvislosti grafu.) Získaný tah ve vrcholu w rozpojíme a náhodně konstruujeme uzavřený tah (z dosud nepoužitých hran) začínající a končící ve vrcholu w. Tento postup opakujeme, dokud nedostaneme tah obsahující všechny hrany.

Tvrzení V souvislém orientovaném grafu existuje otevřený orientovaný eulerovský tah právě tehdy, když existují vrcholy u1, u2 takové, že $d-(u1) = d+(u1)+1, d-(u2) = d+(u2)-1$, a pro každý jiný vrchol v grafu platí $d-(v) = d+(v)$.

V souvislém grafu existuje otevřený neorientovaný eulerovský tah právě tehdy, když v grafu existují přesně dva vrcholy lichého stupně.

Tvrzení Je dán souvislý neorientovaný graf G s 2k vrcholy lichého stupně. Pak existuje k hranově disjunktních otevřených tahů takových, že každá hrana grafu G leží v právě jednom z těchto tahů. Ke grafu G přidáme k hran a to tak, že každá nově přidaná hrana spojuje vždy dva vrcholy lichého stupně. Tím dostaneme eulerovský graf G" (ano, každý vrchol má již sudý stupeň). V grafu G‘‘ najdeme eulerovský uzavřený tah. Jestliže z něj odstraníme všechny přidané vrcholy, rozpadne se na k hranově disjunktních tahů. Tyto tahy splňují podmínky tvrzení.

1.8 Hamiltonovské grafy

Připomeňme, že cesta je tah, ve kterém se neopakují vrcholy (s výjimkou uzavřené cesty, kdy se první vrchol rovná poslednímu).

1.8.1 Hamiltonovské cesty, kružnice, cykly

Je dán graf G. Otevřená cesta se nazývá hamiltonovská cesta, obsahuje-li všechny vrcholy (a tudíž všechny vrcholy přesně jedenkrát). Obdobně hamiltonovská kružnice je kružnice, která obsahuje každý vrchol grafu; hamiltonovský cyklus je cyklus, který obsahuje každý vrchol v grafu.

Úlohy dělíme na existenční a optimalizační. V existenční úloze jde o to, zjistit zda v daném grafu existuje hamiltonovská cesta, kružnice nebo cyklus. V optimalizačních úlohách máme hrany grafu navíc ohodnoceny délkami a požaduje se nalezení hamiltonovské cesty, kružnice nebo cyklu s co nejmenším součtem délek jednotlivých hran tvořících cestu, kružnici nebo cyklus.

Na rozdíl od hledání eulerovských tahů, je hledání hamiltonovských cest, kružnic nebo cyklů velmi obtížná úloha. Přesněji, zjištění, zda v daném grafu existuje hamiltonovská cesta, kružnice nebo cyklus je tzv. NP-úplná úloha. Přesto, nebo právě proto, jsou úlohy tohoto typu v praxi rozšířené.

Aplikace

- Problém obchodného cestujícího
- Dopravní úlohy
- Plánování procesů

Jednoduché nutné podmínky pro existenci hamiltonovské cesty, kružnice nebo cyklu

- Existuje-li v grafu hamiltonovská cesta, musí být graf souvislý
- Existuje-li v grafu hamiltonovská kružnice, musí mít každý vrchol stupeň alespoň 2
- Existuje-li v grafu G hamiltonovský cyklus, musí být graf silně souvislý

Netriviální nutná a postačující podmínka pro zjištění, zda daný graf obsahuje hamiltonovskou cestu, kružnici nebo cyklus, není známa.
[5]

1.9 Nezávislé množiny

Je dán neorientovaný (orientovaný) graf G. Množina vrcholů A se nazývá nezávislá množina vrcholů, jestliže žádná hrana grafu G nemá oba krajní vrcholy v množině A. Jinými slovy, podgraf indukovaný množinou A je diskrétní.

[5] //See also: Metoda větví a mezí

1.9.1 Maximální nezávislá množina

Je dán graf G. Nezávislá množina N se nazývá maximální nezávislá množina, jestliže jakákoli její nadmnožina už není nezávislá. Jinými slovy, N je maximální nezávislá množina, jestliže pro každý vrchol v, který neleží v N, existuje vrchol w ϵ N takový, že v G existuje hrana mezi v a w.

1.9.2 Nezávislost grafu

Je dán neorientovaný nebo orientovaný graf G. Počet vrcholů v nejpočetnější nezávislé množině grafu G se nazývá nezávislost grafu G a značíme jej α(G).

Nejpočetnější nezávislá množina je jistě také maximální, ale ne každá maximální nezávislá množina je současně nejpočetnější.

Poznámka Jádro orientovaného grafu G je nezávislá množina grafu G; to vyplývá z první podmínky, kterou jádro musí splňovat. Ovšem ne každá nezávislá množina orientovaného grafu G je současně jádrem grafu G; jádro musí splňovat obě podmínky (viz výše).

Úplný neorientovaný graf G nazýváme úplným grafem, jestliže je prostý, nemá smyčky a každé dva různé vrcholy jsou spojené hranou. Úplný neorientovaný graf G s n vrcholy má (n(n-1))/2 hran.

1.10 Obarvení grafu

Je dán neorientovaný graf G bez smyček. Barevnost grafu G (též chromatické číslo grafu G) je nejmenší k takové, že G je k-barevný. Barevnost grafu G značíme χ(G).

Množina vrcholů obarvená stejnou barvou tvoří nezávislou množinu grafu. Graf je jednobarevný právě tehdy, když nemá žádnou hranu.

Graf G je dvoubarevný právě tehdy, když neobsahuje kružnici liché délky.

1.10.1 Dvoubarevné grafy

Zjistit, zda je daný graf dvoubarevný, se dá jednoduchou modifikací prohledávání do šířky: Provedeme prohledání grafu do šířky. Vrcholům, které ležely v sudých hladinách, přiřadíme barvu 1; vrcholům, které ležely v lichých hladinách, přiřadíme barvu 2.

Jestliže graf neobsahoval kružnici liché délky, jedná se o obarvení grafu a graf je tedy dvoubarevný. Vede-li hrana mezi dvěma vrcholy v hladinách stejné parity, obsahuje graf kružnici liché délky a není proto dvoubarevný.

Poznámka Zjistit, zda daný graf je tříbarevný, je těžký problém (obecně NP-úplný problém). Pro každý graf G, který mý m hran platí

$\chi(G) \leq \frac{1}{2} + \sqrt{2m + \frac{1}{4}}$

Tvrzení označíme Δ největší stupeň vrcholu grafu G. Pak χ(G)<=Δ+1 Sekvenční barvení

Následující postup obarví graf Δ+1 barvami. Označíme množinu barev B=$\{1,\ldots, \Delta+1\}$.

1. Seřadíme vrcholy do posloupnosti (libovolně). Např. v1,v2,. . . .,vn
2. Probíráme vrcholy v tomto pořadí a vrcholu vi přiřadíme vždy tu nejmenší barvu, kterou nemá žádný jeho soused vrcholu.

Tento algoritmus dává horní odhad pro barevnost grafu. Jedná se ovšem o odhad, který může být velmi vzdálen od barevnosti grafu. Přesněji, existují dvoubarevné grafy, které při nevhodném uspořádání vrcholů v kroku 1, algoritmus obarví n/2 barvami (kde n je počet vrcholů grafu)

Tvrzení Pro každý neorientovaný graf G bez smyček platí: $\alpha(G) + \chi(G) \leq n + 1$. Kde n je počet vrcholů grafu G. [6]

[6] //See also: Biparitní grafy, klika v grafu, doplňkový graf

Společná část - otázka č. 12

Programování v jazyce JAVA: struktura tříd a programu. Události, zdroj a posluchač události, šíření událostí, vlastní události, více zdrojů a posluchačů, rozlišení zdrojů. Výjimky a jejich zpracování, propagace výjimek, hierarchie výjimek, kontrolované a nekontrolované výjimky. (A0B36PR2)

June 13, 2012

1 JAVA - struktura jazyka

Většina této části PRG2 byla zpracována pro otázky pro PRG1 (5, 6, 7 -> struktura tříd a programu). Proto zde vynecháno, nebo stručně doplněno co chybělo.

1.1 Typy programovacích jazyků

- deklarativní - nepopisují jak se co má dělat, ale co má být výsledkem (př. HTML, Prolog: nepopisuje kroky výpočtu, ale fakta o problému a požadovaný výsledek)
- imperativní - zbytek (JAVA, C...), viz. imperativní programování otázka č. 5

1.2 Programovací styly

- Naivní
- Procedurální
- Objektově orientovaný
- Návrhové vzory
- SOA - service-oriented architecture

2 JAVA GUI

(Tato kapitola = pouze stručný přehled <= není to přímo vypsáno jako otázka). Vizuální a interaktivní komunikaci počítač-člověk podporují balíčky:

2.1 Knihovny pro GUI

Základní knihovny:

2.1.1 AWT - Abstract Window ToolKit

- první, těžké(heavyweight)
- vykreslení zajišťuje platforma – rychlejší, ne vždy vše funguje vše stejně

2.1.2 SWING

- doporučené
- nové komponenty (tree-view, list box,...),
- robustní
- Look and Feel - na platformě nezávislý a vypadá stejně na všech platformách a přitom respektuje i18n
- důsledné oddělení modelu od pohledu a řadiče

2.1.3 (SWT-Standard Widget Toolkit, Eclipse IBM)

- podobné AWT (platformově závislé vykreslení)
- mnoho rozšiřujících vlastností

2.2 Základní součásti GUI

Velmi stručně, jen pro úvod do problematiky, v zadání otázky není vypsáno.

2.2.1 Komponenty (dialogové prvky) - v knihovně javax.swing

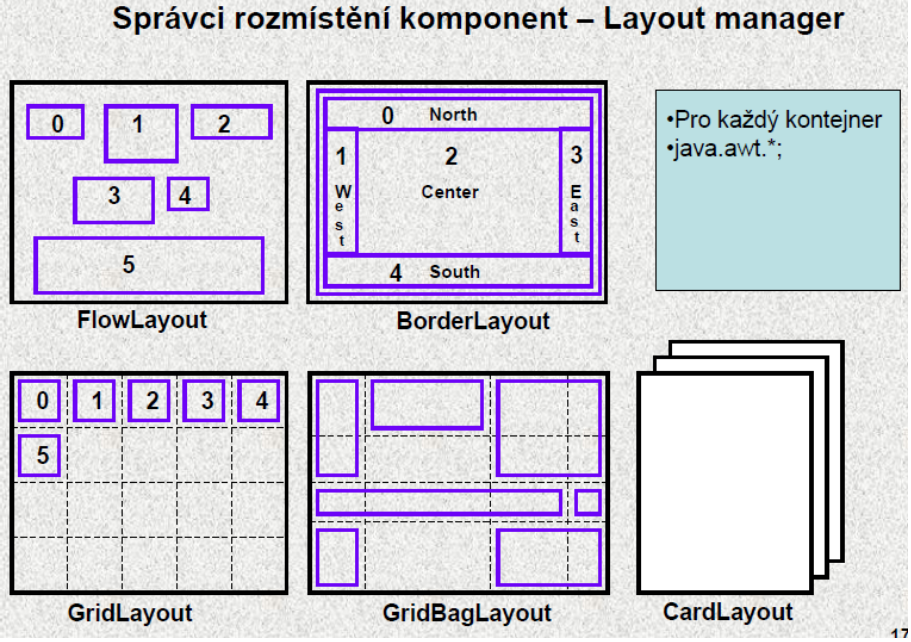
- tlačítka, seznamy, jezdci, textová pole, zatrhávací tlačítka, rádio tlačítka, ...
- společné metody pro velikost, barvu, umístění textu, ...

2.2.2 Kontejnery (v oknech) - v knihovně javax.swing

- Kontejnery se vkládají do oken
- komponenty musí být umístěny v kontejnerech
- dva základní typy kontejnerů:
  - **JPanel** – nejjednodušší, přidělí se komponenty (také JApplet),
  - **JFrame** – složitější, ale více možností

2.2.3 Správce rozmístění (Layout Manager) - v knihovně javax.swing a java.awt

- definuje pozici komponent v kontejneru
- postupně, pevná pozice, podle mřížky, sdružování, ..
- vzhled a chování celé aplikace




```
// pr. border layout
class Okno4_1 extends JFrame{

    public Okno4_1 () {

        ...
        Container kon = getContentPane();
        kon.setBackground(Color.green);
        BorderLayout srb = new BorderLayout();
```

```
        kon.setLayout(srb);
        JButton tl1 = new JButton("Test1");
        kon.add(tl1,srb.WEST);
        JButton tl2 = new JButton("Test2");
        kon.add(tl2,srb.EAST);
        JButton tl3 = new JButton("Test3");
        kon.add(tl3,srb.NORTH);
        setContentPane(kon);
    }
}
```

3 Obsluha událostí

Mechanismus, který umožní zpracovávat vstupní informace programu, predávané nejčastěji přes GUI. Mechanismus reakce na akci uživatele: stisk tlačítka, zadání textu, stisk tlačítka myši, ...

1. Zpracování vstupní informace v GUI je realizováno:
   a) vysláním „události“ na jedné straně „producentem“ (producent = třída)
   b) zachycením této „události“ (událost = objekt této třídy) „posluchačem“ (posluchač = naše třída)
2. Pro každou komponentu je třeba:
   a) deklarovat typ zachycované události, kterou je zájem zpracovat
   b) určit „posluchače“, který má událost obsloužit
3. Akcí uživatele vznikne událost
   a) událost je objektem Javy!
4. Události jsou zachyceny
   a) události jsou zpracovány (obslouženy) „posluchači“ (listener) – třídami s uživatelskými metodami pro reakci na událost
   b) „posluchači“ jsou třídy, které implementují rozhraní naslouchání – musejí mít schopnost „naslouchání“

3.1 Událost

Událost je objekt, který vznikne změnou stavu zdroje - důsledek interakce uživatele střídícími grafickými elementy GUI

1. Událost vznikne:
   a) kliknutím na tlačítko
   b) stiskem klávesy
   c) posunem kurzoru, atd.
2. Události jsou produkovány tzv. producenty což jsou:
   a) tlačítka

b) rámy

c) ostatními grafické prvky

3.2 Zpracování události

Informace o jedné události (zdroj události, poloha kurzoru, atd.) jsou shromážděny v objektu, jehož třída určuje charakter události, napr.:

- ActionEvent ~ událost generovaná tlacítkem
- WindowEvent ~ událost generovaná oknem
- MouseEvent ~ událost generovaná myší

Všechny třídy událostí jsou **následníky třídy event** a jsou umísteny v **java.awt.event.***

Základní princip zpracování událostí:

1. Události jsou generovány zdroji událostí (jsou to objekty, které nesou informaci o události)
2. Události jsou přijímány ke zpracování posluchači událostí (to jsou opět objekty tříd s metodami schopnými událost zpracovat)
3. Zdroj události rozhoduje o tom, který posluchač má reagovat (registruje si svého posluchače)

```
class Okno extends JFrame{

    public Okno (){
        ...
        FlowLayout srb = new FlowLayout();
        kon.setLayout(srb);
        JButton aTlac = new JButton("Pred stiskem");
        kon.add(aTlac);
        // registrace posluchace
        aTlac.addActionListener(new Posluchac0());
        setContentPane(kon);
    }

}
// posluchac
class Posluchac0 implements ActionListener {

    public void actionPerformed(ActionEvent e) {
```

```
        //urcen objekt udalosti
        JButton o=(JButton)e.getSource();
        //reakce na udalost
        o.setLabel(‘‘Po stisku");
    }
}
```

Pozn. Jedna třída může být producentem i posluchačem (addActionListener(**this**), třída zároveň i implemetuje příslušné rozhranní a potřebnou metodu). Pro zpracování události se často používá i vnitřní třída (přehlednost, efektivita kódu, zapouzdření).

```
class Okno6 extends Okno3 {

    JLabel lab1;
    // vnitrni trida, obsluha udalosti
    class Udalost implements ActionListener {
        String vypis;
        public Udalost(String vypis) {
            this.vypis = vypis;
        }
        // oblsuha udalosti
        public void actionPerformed(ActionEvent e) {
            lab1.setText(vypis);
        }
    }
    Udalost aUD, nUD;
    // konstruktor
    Okno6() {
        lab1 = new JLabel("Nazdar6");
        kon.add(lab1);
        aUD = new Udalost("AHOJ");
        // registrace 1. posluchace
        aTlac.addActionListener(aUD);
        nUD = new Udalost("NAZDAR");
        // registrace 2. posluchace
        bbTlac.addActionListener(nUD);
    }
}
```

Anonymní třída jako posluchač:

```
aTlac.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
            lab.setText("AHOJ");
        }
    });
```

3.3 Model šíření události

1. Události jsou předávány posluchačům, které si nejprve musí producent zaregistrovat – např. metodami:
   a) addActionListener()
   b) addWindowListener()
   c) addMouseListener()
   d) ...
2. Producent vysílá jen těm posluchačům, které si sám zaregistroval.
3. Posluchač musí implementovat některé z posluchačských rozhraní (!) (schopnost naslouchat):
   a) ActionListener
   b) WindowListener
   c) MouseListener
   d) ...

3.4 Implementace modelu události

Posluchac události musí implementovat príslušné rozhraní (interface), tj. implementovat príslušné abstraktní metody rozhraní. Pro každý druh události je definována **abstraktní metoda (handler)**, která tuto událost ošetruje:

- actionPerformed
- mouseClicked
- windowClosing, atd.

Handlery jsou deklarovány v rozhraních - posluchaci:

- ActionListener
- MouseListener
- WindowListener, atd.

Předání události posluchači ve skutečnosti znamená vyvolání činnosti handleru. Objekt události je předán jako skutečný parametr handleru. Producent registruje posluchače zavoláním registrační metody:

- addActionListener
- addMouseListener
- addWindowListener, atd.

Vazba mezi producentem a posluchacem je vztah N:M. Jeden posluchac muže být registrován u více producentů, u jednoho producenta muže být registrováno více posluchačů. Událost se předá všem posluchačům, avšak pořadí zpracování není zaručeno.

3.5 Více zdrojů události - jeden posluchač

Následuje ukázka zpracování události a rozlišení producenta:

```
// rozliseni popisem zdroje
public void actionPerformed(ActionEvent e) {

    String s = e.getActionCommand();
    String napis = (s.equals(aTlac.getLabel()))?"PRVNI":"DRUHE";
    lab.setText(napis);

}
// rozliseni objektem zdroje
public void actionPerformed(ActionEvent e) {

    Object o = e.getSource();
    String napis = (o.equals(aTlac)) ? "PRVNI!":"DRUHE!";
    lab.setText(napis);

}
```

3.6 Jeden zdroj událost - více posluchačů

Zajistí se zkrátka vícenásobným voláním metody addActionListener(new Posluchac()) u příslušného elementu. Každá třída=posluchač má svojí metodu, která se po události provede. Pořadí provedení není určeno.

3.7 Zrušení posluchače

removeActionListener(ActionListener posluchac)

4 Výjimky

Vyjímka je „nestandardní situace“:

1. Situace, které jsou nestandardní, či které my považujeme za nestandardní, měli bychom reagovat a mužeme a dokážeme reagovat (RuntimeException)
   a) Pokus o čtení z prázdného zásobníku EmptyStackException
   b) Dělení nulou, indexování mimo rozsah pole, špatný formát císel AritmeticException, NumberFormatException
2. Situace, na které musíme reagovat, Java nás přinutí (Exception, IOException)
   a) Odkaz na chybějící soubor FileNotFoundException
3. Chyba v hardware, závažné chyby, nemůžeme reagovat (Error), (OutOfMemoryError,UnknownError)
   a) Chyba v JVM
   b) HW chyba

Obecne chyba vzniká pri porušení sémantických omezení jazyka Java. Bezpečnostní prvek Javy: zpracování chyb a nestandardních stavů není ponecháno jen na vůli programátora! Reakce na očekávané chyby se vynucuje na úrovni překladu, při nerespektování se překlad nepodaří.

4.1 Reakce na výjimky

Chyba při provádení programu v jazyku Java nemusí znamenat ukončení programu – chybu lze ošetrit a pokracovat dál. Při vzniku výjimky je automaticky vytvořen objekt, který nese informace o vzniklé výjimce. Mechanismus výjimek umožní přenést řízení z místa, kde výjimka vznikla do místa, kde bude zpracována. Oddělení "výkonné" části (try) od části "chybové" - catch.

4.1.1 Úplné neošetření výjimky

Žádné použití throws nebo try-catch způsobí ukončení programu - CHYBA. Java sama ohlásí při překladu, které části jsou kritické a je je třeba ošetrit. Nejsou-li, pak minimálně "vyhozením" na vyšší úroven pomocí klauzule throws, jinak nedojde k překladu.

4.1.2 Neošetření výjimky, ale předání výše

public static void main(String[] args) **throws IOException** { ... }

Závisí-li chod dalšího programu na korektní funkci metody, nemá cenu ji ošetrovat a přitom by činnost programu stejně nemohla pokračovat (proste tu hodnotu musíme mít a ne jen "nespadnutý" program!)

4.1.3 Ošetření výjimky a předání výš - throw

```
public static int XctiInt() throws Exception {

    try {
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        return i;
    } catch (Exception e) {
        System.out.println("Chyba v udaji");
        throw e; // predani vyse
    }

}
```

Volající metoda musí opět použít try-catch, nebo výjimku throws výše.

4.1.4 Kompletní Ošetření výjimky - try-catch

Kompletní ošetření výjimky se provádí konstrukcí try - catch:

try { ... } catch (Exception e) { ... }

Klauzulí catch muže být i více pro ruzné typy výjimek. Pokud výjimka vyhovuje jedné větvi catch, tato se provede a ostatní už ne. Existuje také blok finally, ten se provede vždy, ať už výjimka nastane nebo ne.

4.2 Mechanismus šíření výjimek

Jestliže vznikne výjimka, potom JVM hledá odpovídající klauzuli, která je schopná výjimku ošetrit (tj. převzít rízení):

1. pokud výjimka vznikla v bloku príkazu try, hledá se odpovídající klauzule catch v tomto príkazu, další príkazy bloku try se neprovedou a rízení se predá konstrukci ošetrující výjimku daného typu do místa ošetrení výjimky (tzv. handler = catch blok)

2. pokud výjimka vznikne mimo príkaz try, predá se rízení do místa volání metody a pokracuje se podle predchozího bodu

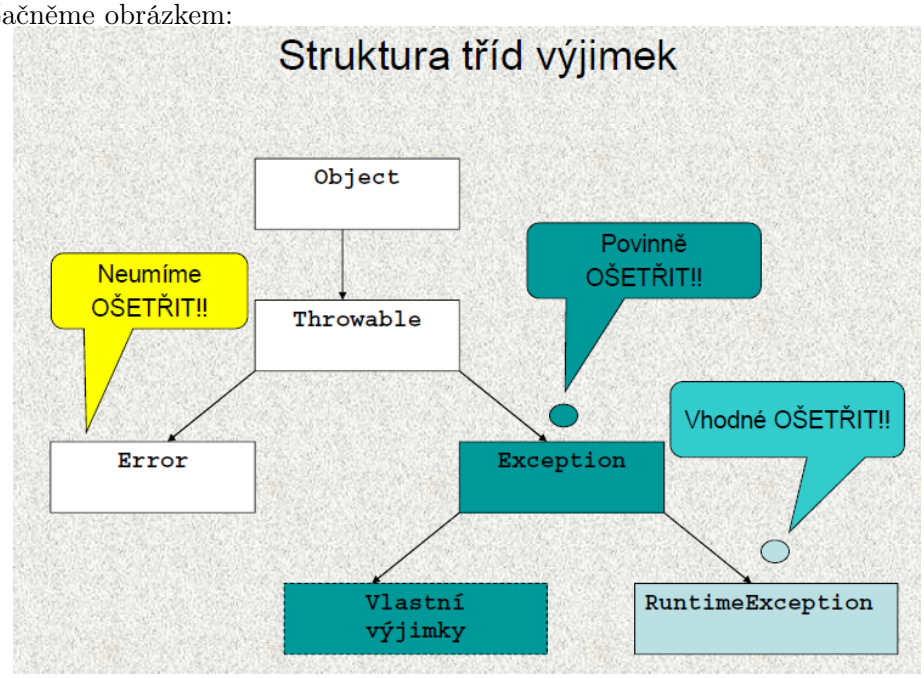
3. pokud taková konstrukce v těle funkce (metody, konstruktoru) není, skončí funkce nestandardně a výjimka se šíří na dynamicky nadřazenou úroven

4. není-li výjimka ošetrena ani ve funkci main, vypíše se (na výstup) a program skončí

Pro rozlišení různých typů výjimek je v jazyku Java zavedena řada knihovních tříd, výjimky jsou instancemi těchto tříd.

4.3 Typy výjimek

Začněme obrázkem:



4.3.1 Kontrolované

Kontrolované výjimky musí být na rozdíl od nekontrolovaných explicitne deklarovány v hlavicce metody, ze které se mohou šírit, jedná se o výjimky trídy Exception, je nutné je povinne obsloužit. Oznacují se též jako výjimky synchronní:

```
void m() throws Exception {
    if (...) throw new Exception();
}
```

Potomek třídy Exception muže ošetrovat i "naše" výjimky, jednoznačne synchronní, vznikají na námi specifikovaném míste. Vyžadují povinné ošetrení, jinak se ohlásí!. Třída Exception je nadtřída výjimek, které prevážne vznikají vlastní chybou aplikace a má smysl je ošetrovat (typicky ošetrení chyb vstupu/výstupu (IOException)).

4.3.2 Nekontrolované

Nekontrolované výjimky jsou takové, které se mohou šírit z vetšiny metod a proto by jejich deklarování obtežovalo, tzv. asynchronní výjimky.

- bežný uživatel není schopen výjimku ošetrit – ze třídy Error. Třída Error je nadtřída všech výjimek, které převážne vznikají v důsledku softwarových či hardwarových chyb výpočetního systému a které vetšinou nelze v aplikaci smysluplně ošetrit.
  - MemoryOverflowError - přetecení paměti
  - ClassFormatError - chybný formát byte-kódu
- chyby, které ošetrujeme podle potřeby, prekladac nekontroluje, zda tyto výjimky jsou ošetreny - podtřídy třídy RuntimeException. Nemusíme na ně reagovat, ale mužeme je predat "výše", překladač nás k reakci nenutí.
  - ArithmeticException - delení 0
  - IndexOutOfBoundsException - indexace mimo meze
  - NumberFormatException - nedovolený prevod znaku na císlo, prectení nenumerické hodnoty
  - NegativeArraySizeException - vytvárení pole se zápornou délkou
  - NullPointerException - dereference odkazu null

4.4 Vlastní výjimky

Ve vlastních třídách muže nastat stav, který chceme ošetrit standardně výjimecným stavem. Vlastní výjimka je potomkem třídy Exception. Jedná se o tzv.synchronní výjimku, vzniká na přesne definovaném míste. Vetšinou se jedná o výjimku, na kterou chceme reakci uživatele. Reakci na vlastní výjimky systém vyžaduje.

```
throw new Exception(); // generujeme vyjímku
```

4.4.1 Vytvoření vlastní výjimky

```
class MojeVyjimka extends Exception {

    int n;
    int d;
    MojeVyjimka(int i, int j) {
        n = i;
        d = j;
    }
    public String toString() {
```

```
        return "Hodnota " + n + "/" + d +" není integer.";
    }
}
//priklad pouziti
throw new MojeVyjimka(numer[i], denom[i]);
```

13 Základy programovaní v C, charakteristika jazyka, model kompilace, struktura programu, makra, podmíněný překlad, syntaxe jazyka, struktury, uniony, výčtové typy, preprocesor, základní knihovny, základní vstup a výstup, pointery, dynamická správa paměti, pole a ukazatelé, funkce a pointery. (A0B36PR2)

13.1 C - Obecná charakteristika

C je nízkoúrovňový, kompilovaný, relativně minimalistický programovací jazyk.

- Univerzální programovací jazyk nižší až střední úrovně
  - Strukturovaný (funkce + data)
  - Zdrojový kód přenositelný (portable), nutno ctít podmínky přenositelnosti, překladač ne (je závislý na platformě)
  - Rychlý, efektivní, kompaktní kód, mnohdy nepřehledný
- Pružný a výkonný, ale nestabilní
- Podpora
  - konstrukcí jazyka vysoké úrovně (funkce, datové struktury)
  - operací blízkých assembleru (ukazatele, bitové operace,...)
    - Slabá typová kontrola
    - Málo odolný programátorovým chybám
- Dává velkou volnost programátorovi v zápisu programu

  - Výhoda: dobrý programátor vytvoří efektivní, rychlý a kompaktní kód
  - Nevýhoda: špatný nebo unavený programátor, pak nepřehledný program náchylný k chybám
- Nutná vlastní správa paměti
- silná vazba na HW, využití jeho možností (inline assembler), špatná nebo žádná podpora národních zvyklostí
- Použití:
  - operační systémy,
  - řídicí systémy
  - grafika
  - databáze
  - programování ovladačů grafických, zvukových a dalších karet
  - programování vestavěných - embeded systémů
  - číslicové zpracování signálů (DSP),
  - ...

13.2 Podobnost C a JAVA

- Program začíná funkcí main(), je základní funkcí programu, každý program musí obsahovat právě jednu tuto funkci
- Stavba funkcí / metod, – Jméno funkce, formální parametry, návratová hodnota, vymezení těla funkce, vymezení bloku, vlastnosti lokálních proměnných (jsou v zásobníku), předávání primitivních typů parametrů hodnotou, return.
- Množina znaků pro konstrukci identifikátorů
- Primitivní typy proměnných se znaménkem (Java nezná proměnné bez znam.)
  - char, short, int, long, float, double
- Aritmetické, logické, relační, bitové operátory
- Podmíněný příkaz if() / if() else
- Příkazy cyklů while() , do while(), for(;;), break, continue
- Programový přepínač switch(), case, default, break

```
int main (int argc, char** argv) {
```

```
    printf("I-copy-and-paste-all-the-time Policy \n");
    return (0);

}
//
//
// delsi ukazka vseho mozneho, je syntakticky spravne, nektere veci pridany jen pro u
// cyklus for, stdin, osetreni vstupu, konstanty...
#include <stdio.h>  /* hlavičkový soubor, přípona h */
#include <stdlib.h>
#include ‘‘konstanty.h’’ // uživatelské soubory se vkládají takto
#define NASOBITEL 5 /* symbolicka konstanta */
#include <math.h> // obsahuje funkce jako sqrt()...
// Zpracovani posloupnosti
// argc     počet parametrů na příkazovém řádku
// *argv[]  pole ukazatelů na příkazy příkazového řádku
int main(int argc, char** argv) {

    int i, suma, dalsi;
    const double PI = 3.14; // unused, jen ukazka -> v math.h neni PI
    printf("Zadejte 5 cisel \n");
    suma = 0;
    for(i = 1; i <= 5; i++){
        scanf("%d", &dalsi);
        // osetreni vstupu
        if(dalsi < 1){

            printf("\n n = %d neni prirozene cislo \n\n", n);
            exit(EXIT_FAILURE);
        }
        suma = suma + dalsi;
    }
    printf("suma = %d \n\n", suma);
    return (EXIT_SUCCESS);

}
```

13.3 Co C nemá

- Interpret kódu (JVM) (C je kompilovaný)
- Objektovou podporu
  - Třídy, objekty, zapouzdření, dědičnost, polymorfismus

- Jednotnou metodiku vytváření a použití strukturovaných proměnných
  - referenční proměnná, new()
- Automatickou správu paměti
  - Garbage collector
- Velikost proměnných nezávislou na platformě
- Způsob uložení proměnných nezávislý na OS (Little-endian, Big endian,...)
- Ošetření výjimek metodikou chráněných bloků
  - try, catch, finally
- Standardní podporu grafického uživatelského rozhraní GUI
- Standardní podporu řízení událostí (events, listeners)
- Standardní podporu webovských aplikací (aplety, síťové připojení)
- Standardní podporu (semi)paralelního zpracování úloh
  - threads, multitasking, multithreading

13.4 Co C nemá, nebo je jinak

- C je kompilovaný jazyk
  - Zdrojový kód je nezávislý (portable) na platformě (málo závislý)
  - Spustitelný kód je závislý na platformě
- Členění programu na moduly, určení jejich vazeb (interface)
- Import knihoven (systémových i uživatelských)
- Určení doby života proměnných, vlastní správa paměti
- Definování konstant a maker
- Vytváření strukturovaných proměnných (mohou být i statické)
- Určení viditelnosti proměnných, modifikátory přístupu
- Ošetření běhových chyb (run-time errors)
  - odpovědný programátor, překladač nevynucuje
- Správa paměti (heap management)
  - odpovědný programátor, malloc / free
- Práce s booleovskými proměnnými a řetězci (boolean a String není)

13.5 Co je v C navíc

- Preprocessor
  - Vkládání hlavičkových souborů (header file) do zdrojového kódu
  - Podmíněný překlad
  - Makra
  - #pragma – doplňující příkazy závislé na platformě
- Linker – spojování přeložených modulů a knihoven do spustitelného kódu
- x (už NEPLATÍ) enum – výčtové typy (množina číslovaných pojmenovaných konstant)
- **Ukazatele (pointer) jako prostředek nepřímého adresování proměnných**
- struct a bitová pole (strukturované proměnné z různých prvků)
- union – překrytí proměnných různého typu (sdílení společné paměti)
- typedef – zavedení nových typů pomocí již známých typů
- sizeof – určení velikosti proměnné (i strukturovaného typu)
- Jiné názvy i parametry funkcí ze standardních knihoven (práce se soubory, znaky, řetězci, matematické funkce, ...)
- příkaz goto navesti;

```
int hledejMax(int *p) {
    for(. . .) {
        for(. . .) {
            if(. . .)
              goto error; // Ven z vnitrniho bloku
        }
    return(. . .);
    error: // Cil skoku uvnitr funkce
        return(. . .);
    }
```

13.6 Program C obsahuje

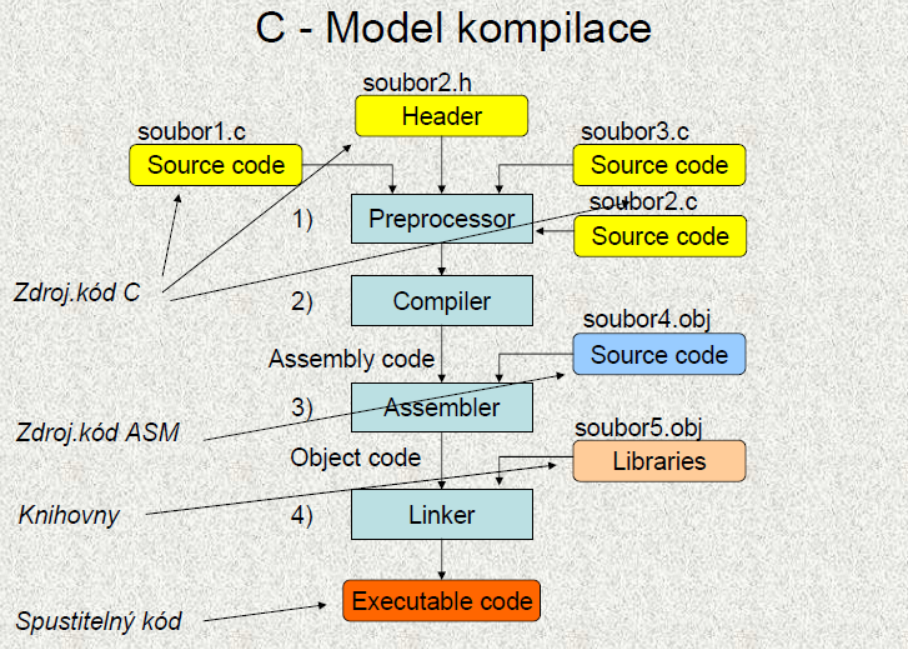
- Příkazy preprocesoru (Preprocessor commands)
- Definice typů (type definitions)
- Prototypy funkcí (function prototypes) kde je uvedena deklarace:
  - Jména funkce
  - Vstupních parametrů
  - Návratové hodnoty funkce
- Proměnné (variables)
- Funkce (functions) (procedura v C je funkce bez návratové hodnoty nebo void)
- Komentáře: //, /* */, nesmí být vnořené

Uvnitř funkce nelze definovat lokální funkce (definice funkci nesmí být vnořené).

13.7 C kompilace

1. Preprocesor:
   a) Čte zdrojový kód v C
   b) Odstraní komentáře (nahradí každý komentář jednou mezerou, pozor: některé překladače nepodporují // komentáře, pouze /* */)
   c) Upraví zdrojový text podle direktiv preprocesoru (řádky začínající #)
      i. Vloží do textu obsah jiného souboru #include ....
      ii. Odebere text vymezený direktivami podmíněného překladu
      iii. Expanduje makra
2. Překladač C:
   a) Čte výstup z preprocesoru
   b) Kontroluje syntaktickou správnost textu
   c) Hlásí chyby a varování
   d) Generuje text v assembleru (když nejsou chyby)
3. Assembler:
   a) Čte výstup z překladače C
   b) Generuje relokovatelný object kód (kód s nevyřešenými odkazy mezi moduly)
   c) Přeloží případné moduly zapsané přímo v assembleru (mix programovacích jazyků)

4. Linker (spojovací program):
   a) Čte object kód všech zúčastněných modulů programu
   b) Připojí knihovní object moduly (přeložené dříve nebo dodané)
   c) Vyřeší odkazy mezi moduly
   d) Generuje spustitelný kód (zjednodušené)



13.8 Celočíselné typy

- Rozsahy celočíselných typů v C nejsou dány normou, ale implementací (pro 16ti a 64 bitové prostředí jsou jiné než je uvedeno v následující tabulce - ta je pro 32 bitové prostředí)
- limits.h, float.h
- Norma pouze garantuje
  - short <= int <= long unsigned short <= unsigned <= unsigned long
- Celočíselné literály (zápisy čísel):
  - dekadický 123 456789
  - hexadecimální 0x12 0xFFFF (začíná 0x nebo 0X)
  - oktalový 0123 0567 (začíná 0)
  - unsigned 123456U (přípona U nebo u)

  - long 123456L (přípona L nebo l)
  - unsigned long 123456UL (přípona UL nebu ul)
- Není-li uvedena přípona, jde o literál typu int

C – Celočíselné datové typy

Typ	Velikost [byte]	Rozsah	Použití
char	1	-128 až +127 nebo 0 až 255	Znaky
unsigned char	1	0 až 255	Malá čísla
signed char	1	-128 až +127	Malá čísla
int	2 nebo 4	-32.768 až +32.767 nebo -2.147.483.648 až +2.147.483.647	Celá čísla
unsigned int	2 nebo 4	0 až 65.535 nebo 0 až 4.294.967.295	Kladná celá čísla
short	2	-32.768 až +32.767	Celá čísla
unsigned short	2	0 až 65.535	Kladná celá čísla
long	4	-2.147.483.648 až +2.147.483.647	Velká celá čísla
unsigned long	4	0 až 4.294.967.295	Kladná celá čísla

•Typ boolean není (až ANSI C99), =0 ->false, !=0 -> true

- C - racionální čísla (neceločíselné datové typy):
  - Velikost reálných čísel určená implementací
  - Většina překladačů se řídí standardem IEEE-754-1985, potom jsou rozsahy reálných čísel dány následující tabulkou:

Typ	Velikost [byte]	Rozsah (uveden pro kladná č.)	Přesnost
float	4	1.2E-38 až 3.4E+38	6 desítkových číslic
double	8	2.3E-308 až 1.7E+308	15 desítkových číslic
long double	10	3.4E-4932 až 1.1E+4932	19 desítkových číslic

- C - typ void: void značí prázdnou hodnotu nebo proměnnou bez typu (jen ukazatelé)
  - void funkce1 (....) - fukce bez návratové hodnoty (procedura)
  - int funkce2 (void) - funkce bez vstupních parametrů
  - void *ptr; - ukazatel bez určeného typu (viz dále)
- #define konstanta
  - #define CERVENA 0 /* Zde muze byt komentar */

- je to makro bez parametrů, každé #define musí být na samostatné řádku)
- Preprocesor provede textovou náhradu všech výskytů slova CERVENA znakem 0
- může být i vnořená: #define MAX_2 MAX_1+30

13.9 Funkce

C je modulární jazyk = funkce je jeho základním stavebním blokem.

- Každý program v C obsahuje minimálně funkci main()

  ```
  int main(int argc, char** argv) { ... }
  ```

- Běh programu začíná na začátku funkce main()
- Definice funkce obsahuje hlavicku funkce a její tělo
- C používá prototypu funkce k deklaraci informací nutných pro překladač, aby mohl správně přeložit volání funkcí i v případě, že definice funkce je umístěna dále v kódu modulu nebo je jiném modulu

  ```
  int max(int a,int b);
  ```

- Deklarace se skládá pouze z hlavičky funkce, (odpovídá interface v Javě)
- Parametry se do funkce předávají hodnotou (call by value), parametrem může být i ukazatel (pointer). Ten pak dovolí předávat parametry i odkazem.
- C nepovoluje funkce vnořené do jiných funkcí (lokální funkce ve funkci)
- Jména funkcí jsou implicitně extern, a mohou se exportovat do ostatních modulů (samostatně překládaných souborů)
- Specifikátor static před jménem funkce omezí viditelnost jejího jména pouze na daný modul (lokální funkce modulu)
- Formální parametry funkce jsou lokální proměnné inicializované skutečnými parametry při volání funkce
- C dovoluje rekurzi, lokální proměnné jsou pro každé jednotlivé volání zakládány znovu (v zásobníku). Kód funkce v C reentrantní (reentrant = Reentrantní provádění bloků znamená, že je možné provádět několikanásobně volaný blok paralelně.).
- Funkce nemusí mít žádné vstupní parametry, zapisuje se funkceX(void)
- Funkce nemusí vracet žádnou funkční hodnotu, pak je návratový typ void (je to procedura)

- Pokud v definici parametrů funkce je klíčové slovo const - tento parametr (předaný odkazem, např. pole) nelze uvnitř funkce měnit
  - př. int fce (const char *src) { ... } , z pole lze pouze číst, jeho prvky nelze uvnitř funkce měnit

13.9.1 Specifikátory paměťové třídy

C - Specifikátory paměťové_třídy (Storage Class Specifiers - SCS):

SCS	Význam
auto (lokální)	Definuje proměnnou jako dočasnou (automatickou). Lze použít jen pro lokální proměnné deklarované uvnitř funkce. Implicitní nastavení je auto, **její platnost je omezena na život bloku, je v zásobníku**
register	Doporučuje překladači umístit proměnnou do registru procesoru (rychlost přístupu). Ten nemusí vyhovět (nemá-li volné registry). Jinak jako proměnné auto.
static	Deklaruje proměnnou jako statickou uvnitř bloku {..}. Vně bloku (kde je proměnná implicitně statická) omezuje její viditelnost na modul. **Ponechává si hodnotu při opuštění bloku**, existuje po celou dobu chodu programu, **v datové oblasti**
extern	Rozšiřuje viditelnost statických proměnných z modulu na celý program, globální proměnné, extern tam, kde se použije, definice bez **v datové oblasti**

- Pozn: v deklaraci proměnné lze uvést vždy jen jeden SCS

x JAVA: globální proměnné nejsou, obchází se také deklarací static (ale je třeba si uvědomit odlišný význam, static v JAVE způsobí, že proměnná existuje pouze jednou pro celou třídu, všechny instance ji sdílí (konstanty v JAVE - final))

13.9.2 Stdin

Načtení hodnoty ze stdin: scanf("%d", &x);

- do funkce scanf vstupuje jako parametr adresa (resp. reference) paměťového místa, kam se má načtená hodnota uložit
- jde o předání parametru odkazem, jde tedy o parametr, který může být využit pro vstup i výstup hodnoty (x JAVA - parametry pouze hodnotou (primitivni typy))
- funkce v C může takto „vracet “ více hodnot

13.9.2.1 Bezpečné načtení double v C

```
#include <stdio.h>
#include <stdlib.h>
int nextDouble(double *cislo){

    // === Bezpecne pro libovolny zadany pocet znaku ===
    // Navratova hodnota:
```

```
    // TRUE - zadano realne cislo
    // FALSE - neplatny vstup
    enum boolean {FALSE,TRUE}; // v ANSI C99 uz existuje true a false, zde výčty
    const int BUF_SIZE = 80;
    char vstup[BUF_SIZE],smeti[BUF_SIZE];
    // fgets precte az sizeof(vstup)-1 znaků ze stdin (standardní vstup) a pushne je
    fgets(vstup,sizeof(vstup),stdin);
    // sscanf je jako scanf, ale cteno z bufferu
    if(sscanf(vstup,"%lf%[^\n]",cislo,smeti) != 1)
        return(FALSE); // Input error
    return(TRUE);
}
```

13.10 Dynamická správa paměti - Alokace paměti

Pole lze alokovat normálně staticky, musí se ale předem zadat jeho velikost.
Způsob dynamické alokace:

```
(int*)malloc(count*sizeof(int)) //zde je typ proměnné int*, ne int !
```

Komplexnější příklad použití:

```
int* ctiPole1 (int *delka, int max_delka){

    // Navratovou hodnotou funkce je ukazatel na pridelene pole
    int i, *p;
    printf(" Zadejte pocet cisel = ");
    // funkce nextInt je temer totozna, jako funkce nextDouble predstavena vyse
    if (!nextInt(delka)) {

        printf("\n Chyba - Zadany udaj neni cele cislo\n\n");
        exit(EXIT_FAILURE);
    }
    if(*delka < 1 || *delka > max_delka){

        printf("\n Chyba - pocet cisel = <1,%d> \n\n",MAX_DELKA);
        exit(EXIT_FAILURE);
    }
    // Alokace pameti (prideleni pameti z "heapu")
    if((p=(int*)malloc((*delka)*sizeof(int))) == NULL){
```

```
            printf("\n Chyba - neni dostatek volne pameti \n\n");
            exit(EXIT_FAILURE);
        }
        printf("\n Zadejte cela cisla (kazde ukoncit ENTER)\n\n");
        for (i = 0; i < *delka; i++) {
            if (!nextInt(p+i)) {

                printf("\n Chyba - Zadany udaj neni cele cislo\n\n");
                exit(EXIT_FAILURE);
            }
        }
        return(p); // p - ukazatel na pridelene a naplnene pole

    }
```

Dealokace probíhá pomocí funkce free, předává se jí ukazatel na začátek alokované paměti:

```
void free(void *ptr)
```

13.11 Operátory

Jako JAVA. Výraz může být operandem, výraz má typ a hodnotu (x void hodnotu nemá). Priority operátorů:

C - Priorita a asociativita operátorů

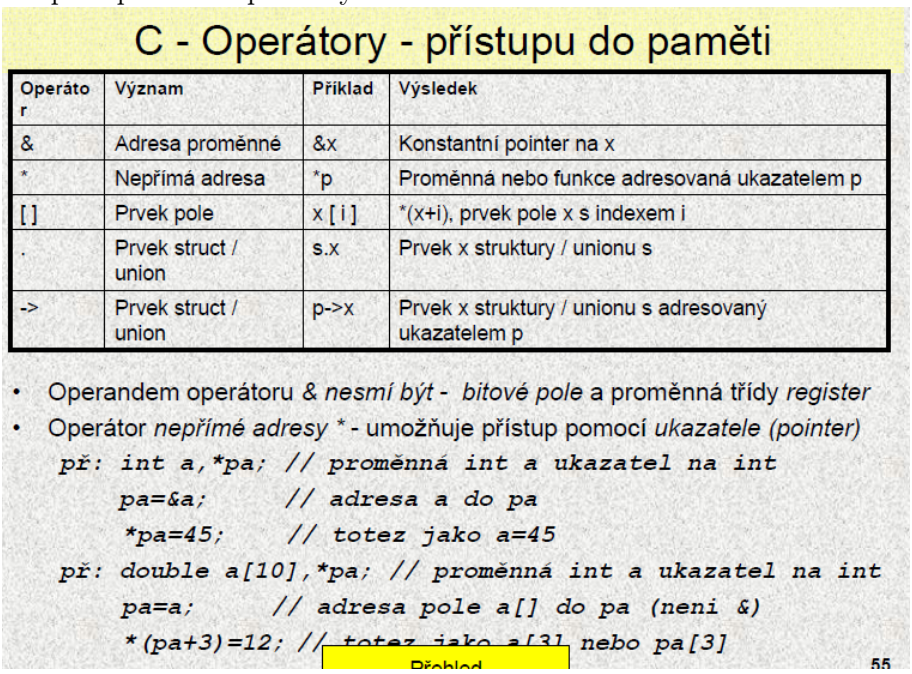
- Pořadí vyhodnocování výrazu se řídí prioritou a asociativitou
- 18/2*3+11 je 38 .... (18/2)*3 +11

Priorita	Operátor	Asociativita		Priorita	Operátor	Asociativita
1	() [] ->	L->R		9	^	L->R
2	! ~ ++ -- + - (type) * & sizeof	R->L		10	|	L->R
3	* / %	L->R		11	&&	L->R
4	+ -	L->R		12	||	L->R
5	<< >>	L->R		13	?:	R->L
6	< <= > >=	L->R		14	= += -= *= /= %= &= ^= |= <<= >>=	R->L
7	== !=	L->R		15	,	L->R
8	&	L->R				

- Ve výrazu: funkce1() + funkce2(), není definováno, která funkce se provede jako první.

- Máme (2 příkazy): int a = 3; a+=a++ + ++a * a++;
  - v JAVE: 31
  - v C: 26, 31, ... není definováno pořadí, programátor není upozorněn, že jde o nejednoznačný zápis
- Zkrácené vyhodnocování logických operátorů (viz JAVA)
- Operandy musí být stejného aritmetického typu, nebo oba struct nebo union stejného typu, nebo oba pointery stejného typu (pravý může být NULL)

Přístup do paměti - operátory:



C - Operátory - přístupu do paměti

Operátor	Význam	Příklad	Výsledek
&	Adresa proměnné	&x	Konstantní pointer na x
*	Nepřímá adresa	*p	Proměnná nebo funkce adresovaná ukazatelem p
[ ]	Prvek pole	x [i]	*(x+i), prvek pole x s indexem i
.	Prvek struct / union	s.x	Prvek x struktury / unionu s
->	Prvek struct / union	p->x	Prvek x struktury / unionu s adresovaný ukazatelem p

- Operandem operátoru & *nesmí být* - *bitové pole* a proměnná třídy *register*
- Operátor *nepřímé adresy* * - umožňuje přístup pomocí *ukazatele (pointer)*

```
př: int a,*pa; // proměnná int a ukazatel na int
    pa=&a;     // adresa a do pa
    *pa=45;    // totez jako a=45
př: double a[10],*pa; // proměnná int a ukazatel na int
    pa=a;      // adresa pole a[] do pa (neni &)
    *(pa+3)=12; // totez jako a[3] nebo pa[3]
```

Přehled 55

13.12 Ukazatelé - Pointery

- Motivace
  - Predávání parametru odkazem
  - Práce s poli, rízení pruchodem polem
  - Pointer na funkci
  - Využívání pole funkcí
  - Vytvárení seznamových struktur
  - Hešování
- Na rozdíl od Javy je možné s ukazatelem provádet aritmetické operace

- Ukazatel v C je prímo implementován pametí procesoru a je možné prímo adresovat, vcetne požadavku na registry
  - Mocný nástroj pro implementaci strojove orientovaných aplikací
- Ukazatel (pointer) je promenná jejíž hodnotou je „ukazatel“na jinou promennou (analogie neprímé adresy ve strojovém kódu ci v assembleru)
- Ukazatel má též typ promenné na kterou muže ukazovat
  - ukazatel na char, int,..
  - „ukazatel na pole“
  - ukazatel na funkci
  - ukazatel na ukazatel
  - atd.
- Ukazatel muže být též bez typu (void), pak muže obsahovat adresu libovolné promenné. Její velikost pak nelze z vlastností ukazatele urcit
- Neplatná adresa, ale definovaná v ukazateli má hodnotu konstanty NULL (kterémukoliv pointeru lze přiřadit hodnotu NULL)
- C za behu programu nekontroluje zda adresa v ukazateli je platná •
- Specialitou C je pointer na funkci!
  - ukazatel umožní, aby funkce byla parametrem funkce
  - ukázka:

```
void funkce1(int x); // Prototyp funkce
void (*pFnc)(int x); // Ukazatel na funkci s parametrem int
int max=200;
pFnc=funkce1; // Adresa funkce1 do ukazatele pFnc
(*pFnc)(max); // volani funkce ‘‘funkce1’’ s parametrem max
```

- Pomocí ukazatele lze predávat parametry funkci odkazem (call by reference) – základní využití
- Adresa promenné se zjistí adresovým operátorem & (ampersand), tzv. referencní operátor (promenná >>> adresa této promenné) int x;
- K promenné na kterou ukazatel ukazuje se pristoupí operátorem neprímé adresy * (hvezdicka), tzv. dereferencní operátor (promenná ukazatel >> hodnota z adresy, kam ukazuje) int *px; px=&x;

13.12.1 Ukázka pro lepší pochopení

```
int x=30; // promenná typu int,  &x - adresa promenné x
int *px; // *px promenná typu int, px promenná typu pointer na int
px=&x; // do promenné typu pointer na int se uloží adresa promenné x
printf(" %d " " %d \n", x, px); // 30 2280564
printf(" %d " " %d \n", &x, *px); // 2280564 30
printf(" %d " " %d \n", *(&x), &(*px)); // 30 2280564
```

13.12.2 Další ukázka

```
int x;
int *px;
int **ppx;
x=1;
px=NULL;
ppx=NULL;
px=&x;
ppx=&px;
**ppx=6;
*px=10;
x = 55;
// *ppx = 20 // CHYBA, konverze int na int* nelze
// px = 20 // CHYBA, konverze int na int* nelze

printf(" %d " " %d " " %d" " \n", x, *px, **ppx); // 55  55  55
printf(" %d %d %d %d " , &x, px, ppx, *ppx); // 2293527  2293527  2293527  (pozn. poi
```

13.12.3 Operace s pointery

- Povolené aritmetické operace s ukazateli:
  - –pointer + integer
  - pointer - integer
  - pointer1 - pointer2 (musí být stejného typu)
- Povolené operandy relace:
  - dva ukazatele (pointers) shodného typu nebo jeden z nich NULL nebo typu void
- Jednoduché prirazení - povolené operandy
  - dva operandy typu - pointer (stejného typu) nebo pravý operand=NULL nebo jeden pointer typu void
- Aritmetické operace jsou užitecné když ukazatel ukazuje na pole

13.13 Pole

- Pole je množina prvku (promenných) stejného typu
- K prvkum pole se pristupuje pomocí poradového císla prvku (indexu) •
- Index musí být celé císlo (konstanta, promenná, výraz)
- Index prvního prvku je vždy roven 0
- Pole se funkcím předává odkazem: void funkcePole (int p[])
- Prvky pole mohou být promenné libovolného typu (i strukturované)
- Pole muže být jednorozmerné i vícerozmerné (prvky pole jsou opet pole)
- Definice pole urcuje:
  - jméno pole
  - typ prvku pole
  - pocet prvku pole
- Prvky pole je možné inicializovat
- Pocet prvku statického pole musí být znám v dobe prekladu
- Prvky pole zabírají v pameti souvislou oblast!
- Velikost pole (byte) = pocet prvku pole * sizeof (prvek pole)
- C - nemá promennou typu String, nahrazuje se jednorozmerným polem z prvku typu char. Poslední prvek takového pole je vždy '\0' (null char)
  - lze také zapsat char *c; c = "ja jsem string"; // c je ukazatel na prvni prvek pole
- C - nekontroluje za behu programu, zda vypocítaný index je platný!

C - Deklarace pole (array):

```
char poleA [9]; // jednorozmerne pole z prvku char
int poleB[3][3]; // dvourozmerne pole z prvku int
Uložení v paměti
```

poleB [0][0], [0][1], [0][2], [1][0], [1][1], [1][2], [2][0], [2][1], [2][2],

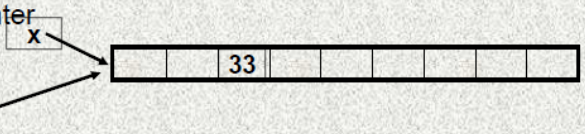
C - Inicializace pole:

```
double x[]={0.1, 0.4, 0.5};
char s[]="abc";
char s1[]={'a','b','c','\0'}; // totez jako "abc"
int ai[3][3]={{1,2,3}, {4,5,6}, {7,8,9}};
char cmd[][10]={"Load","Save","Exit"};
                                // druhy rozmer nutny
```

C - Přístup k prvkům pole:

```
ai[1][3]=15*2;
```

Identifikátor pole je pointer




```
int x[9];
x[2]=33;
x[i] ~~ obsah prvku pole i
// x ~~ pointer na počátek pole
//&x[i] ~~ adresa prvku pole - "adresa x" + i * sizeof(int)
//x[i] ~~ obsah prvku pole - *(x + i)
int *p_x;

p_x = x; // ~~ p_x = &x[0];

for (i=0;i<9;i++)x[i]=0;
for (i=0;i<9;i++)(p_x + i)=0;
```

- **Jméno pole je konstantní ukazatel na počátek pole (na prvek x[0])**

13.14 Hlavičkové soubory

- Důvody:
  - Deklarace funkčního prototypy před použitím
  - Prostředek pro zpřehlednění struktury programu
  - Ukrytí definice funkce, možnost vytváření knihoven
    - Předání souboru .h + .obj
    - Vlastní definice funkce v souborech s relativním kódem .obj
- Obsahují

  - Hlavičky funkcí (funkční prototypy)
  - deklarace funkce
  - Deklarace globálních proměnných
  - Definice datových typů
  - Definice symbolických konstant
  - Definice make

- „obdoba interface“

- Příklad soubor xxx.h:

```
/* podmineny preklad proti opakovanemu vkládání ,,include‘‘ */
#ifndef XXX  // čti: if not defined = proti duplicitě = podmíněný překlad
#define XXX
/* definice symb. konstant vyuzivanych i v jinych modulech */
#define CHYBA -1.0
/* definice maker s parametry */
#define je_velke(c) ((c) >= ‘A’ && (c) <= ‘Z’)
/* definice globalnich typu */
typedef struct{

    int vyska;
    int vaha;

} MIRY;
/* deklarace globalnich promennych modulu xxx.c */
extern MIRY m; // v jiném modulu bude definice MIRY m;
/* uplne funkcni prototypy globalnich funkci modulu xxx.c */
extern double vstup_dat(void);
extern void vystup_dat(double obsah);
#endif
```

- Příklad soubor xxx.c (includuje xxx.h)

```
#include <stdio.h> /* standardni vklad*/
#include ,,xxx.h" /* natazeni konstant,prototypu funkci a globalnich typu vlastniho mo
/* deklarace globalnich promennych */
extern int z; /*ktere nebyly definovány v hlavičkovém soubor */
/* definice globalnich promennych */
int y;/* které nejsou definovány v hlavičkovém soubor */
/* lokalni definice symbolickych konstant a maker */
#define kontrola(x) ( ((x) >= 0.0) ? (x) : CHYBA_DAT )
/* lokalni definice novych typu */
typedef struct{} OSOBA;
```

```
/* definice statickych globalnich promennych */
static MIRY m;
/* uplne funkcni prototypy lokalnich funkci */
int nextDouble(double *cislo);
/* funkce main() */
int main(int argc, char** argv){}
/* definice globalnich funkci - to, ze je glob., bylo definovano v xxx.h */
double vstup_dat(void){ ...return ();}
/*funkční prototypy v.h souboru*/
void vystup_dat(double obsah){ ... }
/* definice lokalnich funkci */
int nextDouble(double *cislo){... }
```

13.15 Typedef

Umožňuje vytvářet nové datové typy:

```
typedef double *PF;
typedef int CELE;
PF x,y;
CELE i,j;
```

13.16 Struktury

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu
  - Obdoba třídy bez metod v Javě
  - Record v jiných jazycích
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům struktury se přistupuje tečkovou notací
- K prvkům struktury je možné přistupovat i pomocí ukazatele na strukturu operátorem ->
- Struktury mohou být vnořené (jak se to řeší v Javě? odpověď: Patrně kompozicí.)
- Pro struktury stejného typu je definována operace přiřazení struct1=struct2
  - (pro proměnné typu pole přímé přiřazení není definováno, jen po prvcích) – co z toho vyplývá???
- Struktury (jako celek) nelze porovnávat relačním operátorem ==
  - co z toho vyplývá???

- Struktura může být do funkce předávána hodnotou i odkazem
- Struktura může být návratovou hodnotou funkce

```
typedef struct {   // <==== Pomoci Typedef
char jmeno[20];    // Prvky struktury, pole
char adresa[50];   // - ‘‘ - pole
int telefon;       // - ‘‘ -
int }Tid,*Tidp;
Tid sk1,skAvt[20]; // struktura, pole struktur
Tidp pid; // ukazatel na strukturu
sk1.jmeno=‘‘Jan Novak’’; // teckova notace
sk1.telefon=123456;
skAvt[0].jmeno=’’Jan Novak’’; // prvek pole
skAvt[3].telefon=123456;
pid=&sk1; // do pid adresa struktury
pid->jmeno=‘‘Jan Novak’’; // odkaz pomoci ->
pid->telefon=123456;
(*pid).jmeno=‘‘Jan Novak’’; // odkaz pomoci *
(*pid).telefon=123456;
```

13.17 Union

- Union je množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionu sdílejí společně stejná paměťová místa (překrývají se) •
- Velikost unionu je dána velikostí největšího z jeho prvků
- Skladba unionu je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům unionu se přistupuje tečkovou notací
- př:

```
union Tnum{ // <==== Tnum=jmeno sablony (tag)
    long n;
    double x;
};
union Tnum nx; // nx - promenna typu union
nx.n=123456789L; // do n hodnota long
nx.x=2.1456; // do x hodnota double (prekryva n)
```

13.18 Podmíněný překlad

```
#if VERSE_CITACE == 1
    do {
        ...
    } while(TRUE);
#endif
```

13.19 Definice vs. deklarace

1. Deklarace určuje interpretaci a vlastnosti identifikátoru(ů)
2. Definice je deklarace včetně přidělení paměti (memory allocation) proměnným, konstantám nebo funkcím

13.20 Standardní knihovny

Vlastní jazyk C neobsahuje žádné prostředky pro vstup a výstup dat, složitější matematické operace, práci s řetězci, třídění, blokové přesuny dat v paměti, práci s datem a časem, komunikaci s operačním systémem,správu paměti pro dynamické přidělování, vyhodnocení běhových chyb (run-time errors) apod..Tyto a další funkce jsou však obsaženy ve standardních knihovnách (ANSI C Library) dodávaných s překladači jazyka C. Uživatel dostává k dispozici přeložený kód knihoven (který se připojuje – linkuje k uživatelovu kódu) a hlavičkové soubory (headers) s prototypy funkcí, novými typy, makry a konstantami. Hlavičkové soubory (obdoba interface v Javě) se připojují k uživatelovu kódu direktivou preprocesoru #include <...>. Je zvykem, že hlavičkové soubory mají rozšíření *,h, např. stdio.h.

Příklad:

- Vstup a výstup (formátovaný i neformátovaný) - stdin.h
- Rozsahy čísel jednotlivých typů - limits.h
- Matematické funkce - stdlib.h, math.h
- Zpracování běhových chyb (run-time errors) - errno.h, assert.h
- Klasifikace znaků (typ char) - ctype.h
- Práce s řetězci (string handling) - string.h
- Internacionalizace (adaptace pro různé jazykové mutace) - locale.h
- Vyhledávání a třídění - stdlib.h

- Blokové přenosy dat v paměti - string.h
- Správa paměti (Dynamic Memory Management) - stdlib.h
- Datum a čas - time.h
- Komunikace s operačním systémem - stdlib.h, signal.h
- Nelokální skok (lokální je součástí jazyka, viz goto) - setjump.h

14 Asymptotická časová a paměťová složitost algoritmů, řád růstu funkcí. (A4B36ALG)

14.1 Asymptotická časová a paměťová složitost algoritmů

Asymptotická složitost je způsob klasifikace počítačových algoritmů. Určuje operační náročnost algoritmu tak, že zjišťuje jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat.

14.1.1 Třída složitosti

Funkce vyjadřující počet operací potřebných ke zpracování dat.

$$1 \ll \log(n) \ll n \ll n * \log(n) \ll n^k \ll k^n \ll n! \ll n^n$$

Pokud spadají dva algoritmy do různých tříd asymptotické složitosti, pak vždy existuje takové množství dat, od kterého je asymptoticky lepší algoritmus vždy rychlejší, bez ohledu na to, kolikrát je některý z počítačů výkonnější.

14.2 Řád růstu funkcí

U většiny algoritmů nelze říci, že jejich složitost odpovídá přesně jedné třídě, protože rychlost algoritmu závisí také na povaze dat. Z tohoto důvodu se používáme řád růstu funkcí, který zohledňuje nejhorší i nejlepší možný běh algoritmu.

- O($f(x)$) - Omicron(f(x)) – algoritmus probíhá asymptoticky stejně rychle nebo rychleji než $f(x)$
- $\Omega(f(x))$ - Gamma(f(x)) – algoritmus probíhá asymptoticky stejně rychle nebo pomaleji než $f(x)$
- $\Theta(f(x))$ - Theta(f(x)) – algoritmus probíhá asymptoticky stejně rychle jako $f(x)$, zároveň platí O($f(x)$) a $\Omega(f(x))$

Funkce $f(x)$ ohraničuje funkci našeho algoritmu $g(x)$, $\mathrm{O}(f(x))$ vyjadřuje ohraničení shora, zatímco $\Omega(f(x))$ zdola. $\Theta(f(x))$ pak značí ohraničení z obou stran. Zapsáno jinak

$$\mathrm{O}(f(x)) \to (\exists c > 0)(\exists n_0)(\forall n > n_0) : g(n) \leq c * f(n)$$

$$\Omega(f(x)) \to (\exists c > 0)(\exists n_0)(\forall n > n_0) : c * f(n) \leq g(n)$$

$$\Theta(f(x)) \to (\exists c_1, c_2 > 0)(\exists n_0)(\forall n > n_0) : c_1 * f(n) \leq g(n) \leq c_2 * f(n)$$

kde $c, c_1, c_2 \in \mathbb{R}^{>0}$ $n_0, n \in \mathbb{N}$ $f, g \in \mathbb{N} \to \mathbb{R}^{\geq 0}$. Vzorce říkají, že existuje taková konstanta c a takové n větší než určité n_0, od kterého platí odpovídající nerovnost.

15 Základní algoritmy řazení (mergesort, quicksort, heapsort, radixsort) a vyhledávání půlením intervalu, jejich složitost. (A4B36ALG)

15.1 Základní algoritmy řazení

V popiscích jednotlivých algoritmů se vyskytují následující hesla a charakteristiky

Rozděl a panuj
Tato metoda označuje ty algoritmy pro práci s daty, které řeší problém rozdělením řešené úlohy na **dílčí části (podproblémy)**, nad kterými se provádí algoritmická operace. Často se tato metoda implementuje rekurzivně nebo iterativně a původní úloha se dělí na stále menší části.

Stabilní/nestabilní algoritmus
Stabilním algoritmus je ten, který zachovává pořadí stejných prvků, tak jak byly na vstupu v seznamu. Seřadí je na odpovídající místo, ale v rámci stejných prvků zůstanou ve stejném pořadí. U nestabilního algoritmu toto nezaručíme.

15.1.1 Mergesort

Grafické zpracovaní mergesortu z předmětu A4B36ALG spolecna/15/mergesort.pdf

Mergesort je **stabilní** řadicí algoritmus typu **rozděl a panuj** s asymptotickou složitostí $O(n * \log(n))$. Merge sort pracuje na bázi **slévání** již seřazených částí pole.

15.1.1.1 Merge

Proces merge nebo-li slévání probíhá následovně. Máme dva seznamy, u kterých víme, že jsou seřazeny ve stejném pořadí. Postupně je procházíme a díváme se, který z právě iterovaných členů je větší (případně menší, záleží, jak srovnávám). Ten větší (resp. menší) z dvojice vložíme do pomocného seznamu. Takto se dostanem do fáze, kdy jeden ze seznamů, kterými iterujem, je už prázdný. Můžeme tedy zbytek druhého vzít a celý ho zkopírovat na konec do pomocného seznamu.

Ve vypsaném kódu je první `while` cyklus na řádku 13 porovnávání. Z následujících dvou cyklů (21 a 25) se provede jen jeden a to ten, který zkopíruje do pomocného seznamu zbývající členy jednoho ze dvou iterovaných seznamů.

```
1  /**
2   * Slevani pro Merge sort
3   * @param array pole k serazeni
4   * @param aux pomocne pole (stejne velikosti jako razene)
5   * @param left prvni index, na ktery smim sahnout
6   * @param right posledni index, na ktery smim sahnout
7   */
8  private static void merge(int[] array, int[] aux, int left, int
       right) {
9      int middleIndex = (left + right)/2;
10     int leftIndex = left;
11     int rightIndex = middleIndex + 1;
12     int auxIndex = left;
13     while(leftIndex <= middleIndex && rightIndex <= right) {
14         if (array[leftIndex] >= array[rightIndex]) {
15             aux[auxIndex] = array[leftIndex++];
16         } else {
17             aux[auxIndex] = array[rightIndex++];
18         }
19         auxIndex++;
20     }
21     while (leftIndex <= middleIndex) {
22         aux[auxIndex] = array[leftIndex++];
23         auxIndex++;
24     }
25     while (rightIndex <= right) {
26         aux[auxIndex] = array[rightIndex++];
27         auxIndex++;
28     }
29 }
```

15.1.1.2 Průběh algoritmu

Algoritmu dodáme array, který postupným půlením rekurzivně rozdělíme do nejmenších skupin, tedy dvojic a případně lichého členu. Na řádku 14 aplikujeme proces merge na tyto malé jednotky, a ve stacku, který se nám během rekurze vytvořil, skočíme o úroveň výš, kde sléváme podseznamy délky 4, případně 3. Takto skáčeme až do chvíle, kdy se ocitneme v hlavní smyčce programu a sléváme levou a pravou polovinu seznamu, který nám byl vložen na vstupu.

```java
/**
 * Razeni slevanim (od nejvyssiho)
 * @param array pole k serazeni
 * @param aux pomocne pole stejne delky jako array
 * @param left prvni index na ktery se smi sahnout
 * @param right posledni index, na ktery se smi sahnout
 */
public static void mergeSort(int[] array, int[] aux, int left,
    int right) {
    if(left == right) return;
    int middleIndex = (left + right)/2;
    mergeSort(array, aux, left, middleIndex);
    mergeSort(array, aux, middleIndex + 1, right);

    merge(array, aux, left, right);
}
```

15.1.2 Quicksort

Grafické zpracovaní quicksortu z předmětu A4B36ALG spolecna/15/quicksort.pdf

Quicksort je velmi rychlý **nestabilní** řadící algoritmus na principu **rozděl a panuj** s asymptotickou složitostí O(n^2) a s očekávanou složitostíO($n * \log(n)$).

15.1.2.1 Průběh algoritmu

1. Levý index se nastaví na začátek zpracovávaného úseku pole, pravý na jeho konec, zvolí se pivot, nejjednodušeji tak, že se vybere prvek na začátku zpracovávaného úseku. Levý index se pohybuje doprava a zastaví se na prvku vetším nebo rovném pivotovi. Pravý index se pohybuje doleva a zastaví se na prvku menším nebo rovném pivotovi.

2. Pokud je levý index ještě před pravým, príslušné prvky se prohodí, a oba indexy se posunou o 1 ve svém směru. Jinak pokud se indexy rovnají, jen se oba posunou o 1 ve svém směru.

3. Cyklus se opakuje, dokud se indexy neprekříží, tj. pravý se dostane pred levého.

4. Následuje rekurzivní volání (zpracování „malých" a „velkých" zvlášť) na úsek od začátku do pravého indexu včetně a na úsek od levého indexu včetně až do konce, má-li príslušný úsek délku větší než 1.

```
1  private static void qSort(int a[], int low, int high) {
2    int iL = low, iR = high;
3    int pivot = a[low];
4    do {
5      while (a[iL] < pivot) iL++;
6      while (a[iR] > pivot) iR--;
7      if (iL < iR) {
8        swap(a,iL, iR);
9        iL++; iR--; }
10     else
11       if (iL == iR) { iL++; iR--;}
12   } while( iL <= iR);
13   if (low < iR)  qSort(a, low, iR);
14   if (iL < high) qSort(a, iL, high);
15 }
```

15.1.3 Heapsort

Grafické zpracovaní heapsortu z předmětu A4B36ALG spolecna/15/heapsort.pdf

Heapsort (řazení haldou) je jedním z nejefektivnějších řadících algoritmů založených na porovnávání prvků s asymptotickou složitostí $O(n * \log(n))$. Heapsort je **nestabilní**.

15.1.3.1 Vlastnosti haldy

Halda je binární strom s rekurzivní vlastností býti haldou. Rekurzivita vlastnosti značí, že i každý podstrom haldy je taktéž haldou. V každé haldě také platí, že otec má vždy vyšší hodnotu než jeho potomci. Při její implementaci polem pak také platí, že pokud je otec na indexu i, tak jsou jeho potomci na indexech $2i + 1$ a $2i + 2$ (indexováno od 0). Z tohoto uspořádání plyne, že tvar haldy je pyramida s částečně useknutou pravou stranou základny.

15.1.3.2 Průběh algoritmu

Budu se odkazovat na zdrojový kód na následující stránce.

1. Postavme haldu nad zadaným polem. To se děje ve `for-cyklu` na řádku 7. Je důležité povšimnout si, že cyklus probíhá od poloviny a jde pozpátku. Proč? Protože seznam znázorňuje haldu, to znamená, že rodičem posledního členu bude prostřední člen. Tohle je potřeba pochopit, mrkněte na spolecna/15/heapsort.pdf.

2. Z našeho neuspořádaného seznamu jsme udělali haldu. Je to pořád seznam, array, ale splňuje charakteristiku haldy. Na řádku 10 začíná řazení.

3. Začneme tím, že prvek na vrcholu prohodíme s i-tým prvkem seznamu (řádek 11). Tím dosáváme na konci seznamu již seřazené prvky.

4. Prohozením se ale na vrcholu haldy může ocitnout prvek, který tam nemá co dělat. Proto na řádku 12 dojde k opravení haldy.

Hlavní část algoritmu je v metodě repairTop. Ta zařadí vrchol haldy na správné místo (řádek 38, proměnná topIndex se nastaví ve `while-cyklu`) a nahradí ho prvkem, který má být na vrcholu (řádek 31).

```java
/**
 * Heapsort - razeni haldou
 * @param array pole k serazeni
 * @param descending true, pokud ma byt pole serazeno sestupne, false pokud vzestupne
 */
public static void heapSort(Comparable[] array, boolean descending) {
    for (int i = array.length / 2 - 1; i >= 0; i--) {
        repairTop(array, array.length - 1, i, descending ? 1 : -1);
    }
    for (int i = array.length - 1; i > 0; i--) {
        swap(array, 0, i);
        repairTop(array, i - 1, 0, descending ? 1 : -1);
    }
}

/**
 * Umisti vrchol haldy na korektni misto v halde (opravi haldu)
 * @param array pole k setrizeni
 * @param bottom posledni index pole, na ktery se jeste smi sahnout
 * @param topIndex index vrsku haldy
 * @param order smer razeni 1 == sestupne, -1 == vzestupne
 */
private static void repairTop(Comparable[] array, int bottom, int topIndex, int order) {
    Comparable tmp = array[topIndex];
    int succ = topIndex * 2 + 1;
    if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order) {
        succ++;
    }

    while (succ <= bottom && tmp.compareTo(array[succ]) == order) {
        array[topIndex] = array[succ];
        topIndex = succ;
        succ = succ * 2 + 1;
        if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order) {
            succ++;
        }
    }
    array[topIndex] = tmp;
}
```

15.1.4 Radixsort

Grafické zpracovaní radixsortu z předmětu A4B36ALG spolecna/15/radixsort.pdf

Princip radix sortu vychází přímo z definice stabilního řazení – řadicí algoritmus je stabilní, pokud zachovává pořadí klíčů, které mají stejnou hodnotu (tj. pokud stuktury seřadíme napřed dle klíče **A**, poté podle klíče **B**, tak jsou seřazeny podle **B**, a kde jsou si hodnoty **B** rovny, tam jsou struktury v pořadí daném klíčem **A**).

Popis algoritmu vychází z následujícího kusu kódu, který je ale zkrácen. Celá kopie je v `spolecna/15/radixsort.java`. Názvy proměnných vychází z označení v `spolecna/15/radixsort.pdf`, je proto dobré procházet si kód zároveň s tímto pdf.

1. Vytvoříme pomocné tabulky

   - **z** - první výskyt znaků
   - **k** - poslední výskyt znaků
   - **d** - tabulka odkazů na další výskyt znaku

   a to z posledního znaku slova (poslední klíč při řazení, řádek 4). Funkce `initStep` projde pole všech slov na vstupu (15), zjistí poslední písmeno (16), pokud ještě není v tabulce prvních výskytů znaků **z**, přidá ho do ní a zároveň do tabulky posledního výskytu znaku **k** (18). Pokud už nějaký záznam s klíčem **c** existuje, algoritmus uloží na jeho pozici v tabulce **d** odkaz na právě iterovaný prvek (20), tento prvek se stane i posledním výskytem daného znaku (21).

2. Poté začneme iterovat přes délku slov na vstupu (5) - to je důvod, proč musí být všechny řetězce různé délky uvedeny na stejnou délku připojením "nevýznamných znaků", např. mezer.

3. Funkce `radixStep` se podívá na aktuální pomocné tabulky, které určují pořadí prvků podle aktuálního klíče. Při prvním průchodu funkcí to tedy bude podle posledního znaku.

   Vlastní funkce `radixStep` začíná iterací přes pole prvních výskytů znaků (28). V ní je vnořen další cyklus určen pomocnou tabulkou **d** a **k**. Pokud se aktuální prvek rovná koncovému výskytu znaku (39), tento vnitřní cyklus skončí. Během tohoto cyklu se testuje to samé jako ve funkci `initStep` s tím rozdílem, že tentokrát ukládáme informace do pomocných tabulek **z1**, **k1**, **d1** (34, 36, 37), takže si nepřepíšeme aktuálně iterované **z**, **k**, **d**. Přepsání proběhne až v hlavním cyklus programu (7, 8, 9).

```
1  void radix_sort (String [] a) {
2    int len   = ...; // number of chars used (2^16?)
3    int [] z  = new int [len]; int [] k  = new int [len]; int []
        z1 = new int [len]; int [] k1 = new int [len]; int [] d =
         new int [a.length]; int [] d1 = new int [a.length]; int
        [] aux;
4    initStep(a, z, k, d);
5    for (int p = a[0].length()-2; p >= 0; p-- ) {
6      radixStep(a, p, z, k, d, z1, k1, d1);
7      aux = z; z = z1; z1 = aux;
8      aux = k; k = k1; k1 = aux;
9      aux = d; d = d1; d1 = aux;
10   }
11 }
12
13 void initStep (String[] a, int[] z, int [] k, int[] d){
14   int pos = a[0].length()-1;
15   for (int i = 0; i < a.length; i++) {
16     c = (int) a[i].charAt(pos);
17     if (z[c] == -1)
18       k[c] = z[c] = i;
19     else {
20       d[k[c]] = i;
21       k[c]    = i;
22   }
23   }
24 }
25
26 void radixStep(String[]a, int pos, int[] z, int[] k,
27   int[] d, int[] z1, int[] k1, int[] d1){
28   for (int i = 0; i < z.length; i++)
29     if (z[i] != -1) {
30       j = z[i];
31       while (true) {
32         c = (int) a[j].charAt(pos);
33         if (z1[c] == -1)
34           k1[c]= z1[c] = j;
35         else {
36           d1[k1[c]] = j;
37           k1[c2] = j;
38         }
39       if (j == k[i]) break;
40       j = d[j];
41     }
42   }
43 }
```

15.1.5 Vyhledávání půlením intervalu

Metoda půlení intervalu (nebo také binární vyhledávání) je vyhledávací algoritmus typu **rozděl a panuj** na nalezení zadané hodnoty v **uspořádaném seznamu** pomocí zkracování seznamu o polovinu v každém kroku.

Binární vyhledávání najde medián, porovná s hledanou hodnotou a na základě výsledku porovnání se rozhodne o pokračování v horní nebo dolní části seznamu a rekurzivně pokračuje od začátku. Binární vyhledávání je algoritmus s logaritmickou časovou složitostí O(log(n)). Přesněji, je potřeba iterací na získání výsledku. Je značně rychlejší než lineární vyhledávání, které má časovou složitost O(n). Nicméně vyžaduje, aby data byla setříděna, je tudíž vhodný jen pro určitou množinu problémů.

15.1.6 Srovnání složitostí

Grafické zpracovaní složitostí vypracované z předmětu A4B36ALG spolecna/15/benchmark.pdf

	Nejhorší případ	Nejlepší případ	Průměrný případ	Stabilní
Quick sort	$\Theta(n^2)$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ne
Merge sort	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ano
Heap sort	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ne
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano

16 Datové typy, seznam, zásobník, fronta, operace s nimi, jejich složitost. Vyhledávací a rozhodovací stromy (binární, AVL, B) jejich specifika a využití, efektivita operací, volba rozptylovací funkce pro specifické typy dat. (A4B36ALG)

16.1 Datové typy

Datový typ definuje v programování druh nebo význam hodnot, kterých smí nabývat proměnná (nebo konstanta). Datový typ je určen oborem hodnot a zároveň výpočetními operacemi, které lze s hodnotami tohoto typu provádět. Datový typ nemůže být určen pouze oborem hodnot, protože existují i datové typy, lišící se pouze v operacích, které je s nimi možné provádět.

Následující výpis je pro jazyk Java.

16.1.1 Primitivní datové typy

typ	popis	velikost
byte	celé číslo	8 bitů
short	celé číslo	16 bitů
int	celé číslo	32 bitů
long	celé číslo	64 bitů
float	reálné číslo	32 bitů
double	reálné číslo	64 bitů
char	znak UNICODE	16 bitů
boolean	logická hodnota	1 bit

16.1.2 Referenční datové typy

- Objekty
- Pole

Hodnota referenční proměnné je odkaz (reference) do paměti na místo, kde je objekt (nebo pole) uložen. V jiných programovacích jazycích se používají pro tento účel pointery (ukazatelů do paměti), v Javě se přímo s pamětí nepracuje, místo pointerů se používají referenční proměnné.

Typ `null` je prázdná hodnota pro referenční typy a znamená, že chybí odkaz na objekt resp. pole.

Jelikož referenční proměnná obsahuje pouze odkaz na objekt, nikoli objekt samotný, přiřazením její hodnoty jiné proměnné se přiřadí opět pouze reference na původní objekt, nikoli jeho samotná data tak, jak to bylo u primitivních datových typů.

16.2 Seznam

V Javě je seznam implementován třídou ArrayList. Ukládá veškeré prvky do běžného pole, které má neměnnou délku. V okamžiku, kdy překročíme délku tohoto pole, tak ArrayList vytvoří nové pole dvojnásobné délky, překopíruje do něj veškeré prvky a původní pole zahodí.

V případě, že je značná část pole nevyužívaná, tak ArrayList vytvoří nové menší pole, do nejž opět veškerá překopíruje a původní pole zahodí (čímž nám ušetří paměť).

Operace

- add(i, e) - vloží prvek `e` do seznamu na index `i` (náročnost O(n))
- remove(i | e) - odstraní prvek `e` nebo to, co je na indexu `i` (náročnost O(n))
- get(i) - vrátí prvek na indexu `i` (náročnost O(1))

16.3 Zásobník

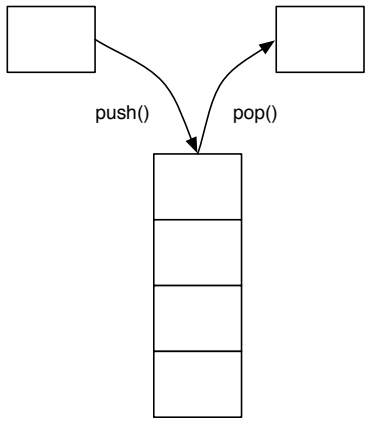


Figure 16.1: Zásobník

Zásobník (stack) je jednou ze základních datových struktur, která se využívá především pro dočasné ukládání dat v průběhu výpočtu. Zásobník data ukládá způsobem, kterému se říká **LIFO** - last in, first out - čili poslední vložený prvek jde na výstup jako první, předposlední jako druhý a tak dále.

Operace

- push - vloží prvek na vrch zásobníku (náročnost O(1))
- pop - odstraní vrchol zásobníku (náročnost O(1))
- top - dotaz na vrchol zásobníku (náročnost O(1))
- isEmpty - dotaz na prázdnost zásobníku

16.4 Fronta

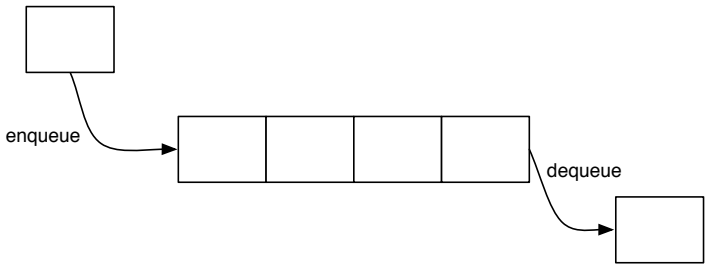


Figure 16.2: Fronta

Fronta (queue) slouží k ukládání a výběru dat takovým způsobem, aby prvek, který byl uložen jako první, byl také jako první vybrán. Tomuto principu se říká FIFO - first in, fist out.

Operace

- addLast (enqueue) – vloží prvek do fronty (náročnost O(1))
- deleteFirst (poll, dequeue) – získá a odstraní první prvek (hlavu) fronty (náročnost O(1))
- getFirst (peek) – získá první prvek fronty (náročnost O(1))
- isEmpty - dotaz na prázdnost fronty

16.5 Stromy

Jedná se o hierarchickou strukturu, kde každý otec má 0 až mnoho dětí a každé dítě právě jednoho otce takovým způsobem, že v této struktuře nejsou cykly. Uzel, který je

praotcem všech ostatních uzlů nazveme kořenem (z pohledu teorie grafů tím vytvoříme orientovaný strom). Uzel, který nemá žádné potomky nazýváme listem.

Být stromem je rekurzívní vlastnost - každý podstrom stromu S je také stromem.

16.5.1 Binární strom

Binární strom je orientovaný graf s jedním vrcholem (kořenem), z něhož existuje cesta do všech vrcholů grafu. Každý vrchol binárního stromu může mít maximálně dva orientované syny a s výjimkou kořene právě jednoho předka. Kořen předka nemá.

V praktickém programování je obvykle binární strom reprezentován dvěma způsoby:

1. pomocí dynamické struktury, kde jsou hrany reprezentovány ukazateli

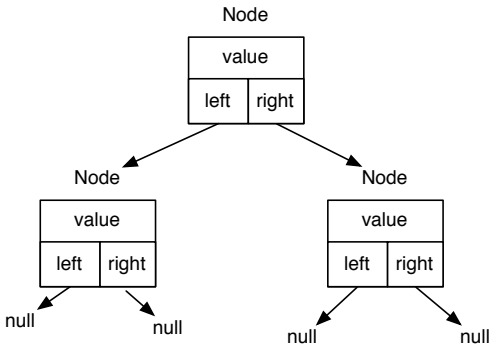


Figure 16.3: Strom pomocí dynamické struktury

2. pomocí pole, kde prvek s indexem i má následníky s indexem 2i+1 a 2i+2 (za předpokladu, že pole je indexováno od 0). Takto je například reprezentovaná halda v algoritmu heapsort (otázka 15).

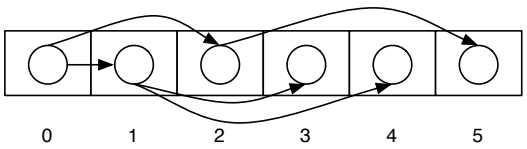


Figure 16.4: Strom pomocí pole

Za zmínku stojí speciální typ binárního stromu - **Vyvážený binární** strom, jehož hloubka listů se od sebe liší maximálně o jedna.

16.5.1.1 Binární vyhledávací strom

Grafické zpracovaní binárního vyhledávacího stromu z předmětu A4B36ALG spolecna/15/bvs.pdf

Binární vyhledávací strom (BST - z angl. Binary Search Tree) je datová struktura založená na binárním stromu, v němž jsou jednotlivé prvky (uzly) uspořádány tak, aby v tomto stromu bylo možné rychle vyhledávat danou hodnotu. To zajišťují tyto vlastnosti:

- Jedná se o binární strom, každý uzel tedy má nanejvýš dva syny — levého a pravého.
- Každému uzlu je přiřazen určitý klíč. Podle hodnot těchto klíčů jsou uzly uspořádány.
- Levý podstrom uzlu obsahuje pouze klíče menší než je klíč tohoto uzlu.
- Pravý podstrom uzlu obsahuje pouze klíče větší než je klíč tohoto uzlu.

Vyhledávání

Vyhledání konkrétní hodnoty v binárním vyhledávacím stromu typicky probíhá rekurzivně. Začíná v kořeni. V každém kroku porovná hledanou hodnotu s klíčem zkoumaného uzlu. Pokud jsou si rovny, hodnota byla nalezena. Je-li hledaná hodnota menší, pokračuje hledání v levém podstromu. Je-li větší, bude hledání pokračovat v pravém podstromu. Díky uspořádání stromu je cesta k hledané hodnotě jednoznačně určena.

Přídání uzlu

Vložení nového uzlu začíná hledáním jeho pozice ve stromu – postupuje se stejně jako při vyhledávání, jako hledaná hodnota se použije klíč vkládaného uzlu. Tato fáze může vést ke dvěma různým výsledkům:

- klíč byl nalezen, strom tedy dotyčnou hodnotu již obsahuje a není třeba ji vkládat (komplikovanější varianty připouštějící vícenásobný výskyt stejného klíče by pokračovaly dál do podstromu připouštějícího rovnost)
- algoritmus narazil na neexistující uzel, nový uzel bude vložen na toto místo, protože sem podle hodnoty svého klíče patří

Odstranění uzlu

- **Odstranění listu:** Odstranění uzlu bez potomků se jednoduše provede odstraněním uzlu ze stromu.
- **Odstranění uzlu s jedním potomkem:** Provede se nahrazením uzlu uzlem potomka.
- **Odstranění uzlu se dvěma potomky:** Nechť se odstraněný uzel nazývá N. Pak je hodnota uzlu N nahrazena nejbližší vyšší (uzel pravého podstromu, který je nejvíc vlevo (nejlevější ☺)), nebo nižší hodnotou (uzel levého podstromu, který je nevíc vpravo (nejpravější ☺)). Takový uzel má nanejvýš jednoho potomka, lze jej tedy ze stromu vyjmout podle jednoho z předchozích pravidel. Obě možnosti ilustruje následující obrázek, kdy je ze stromu odstraněn uzel s klíčem 7. V dobré implementaci je doporučeno obě varianty střídat, jinak dochází k narušení rovnováhy.

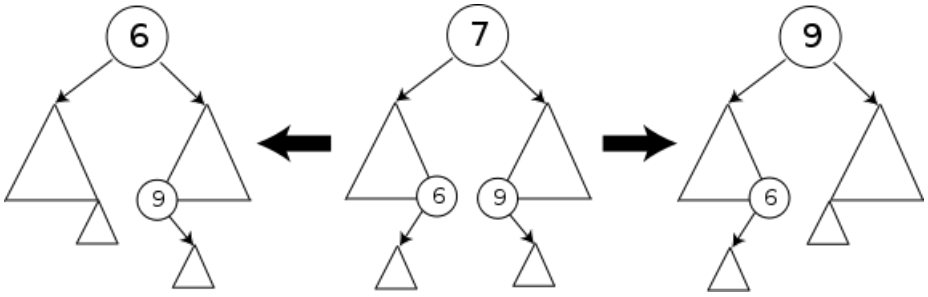


Figure 16.5: Odstranění uzlu se dvěma potomky

16.5.1.2 AVL strom

Grafické zpracovaní AVL stromu z předmětu A4B36ALG spolecna/15/avl.pdf

AVL strom je datová struktura pro uchovávání údajů a jejich vyhledávání. Pracuje v logaritmicky omezeném čase. Jedná se o **samovyvažující se binární vyhledávací strom**.

- Vrchol má maximálně dva následníky (plyne z vlastností binárního stromu, kterým AVL strom je).
- V levém podstromu vrcholu jsou pouze vrcholy s menší hodnotou klíče
- V pravém podstromu vrcholu jsou pouze vrcholy s větší hodnotou klíče
- **Délka nejdelší větve levého a pravého podstromu se liší nejvýše o 1 (vyvážení AVL stromu).**

Poslední vlastnost je demonstrována na následující dvojici stromů.

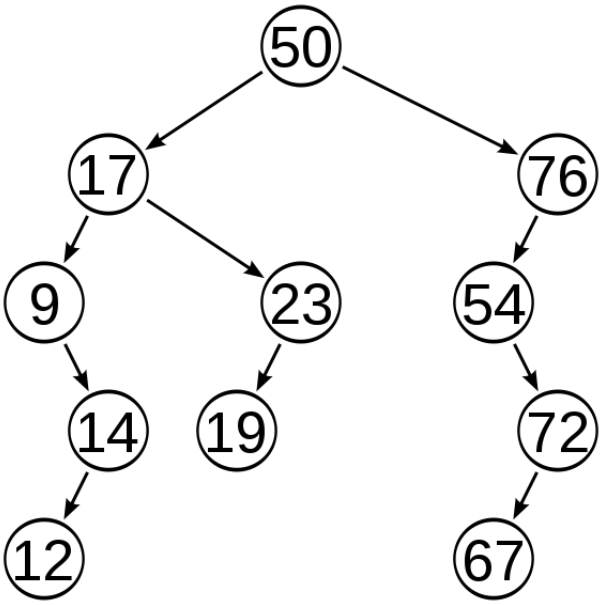


Figure 16.6: Binární vyhledávací strom

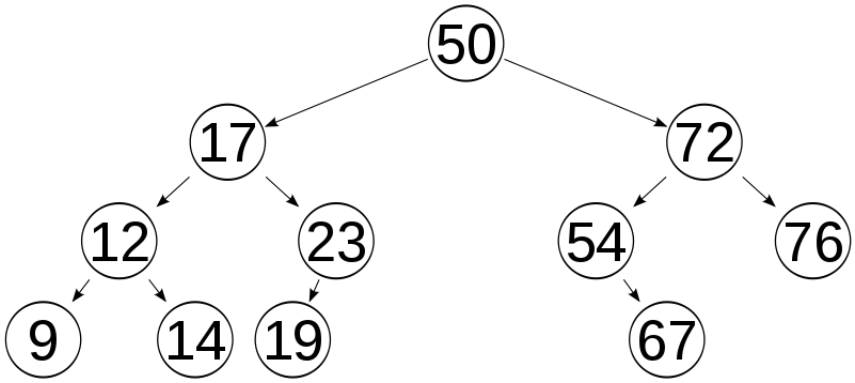


Figure 16.7: AVL strom

První strom nesplňuje vlastnosti AVL stromu v uzlech 9, 76 a 54, kde je rozdíl délky nejdelší větve levého a pravého podstromu 2, 3 a 2, tedy více než 1.

Vyhledávání

Totožné jako u vyhledávání v BVS 16.5.1.1.

Přidání uzlu

Totožné jako u přidávání v BVS 16.5.1.1. Strom se ale po přidání může stát nevyváženým, proto je potřeba provést vyvažovací rotace.

Rotace

Podrobně a obrázky popsáno v spolecna/15/avl.pdf. Postup je následující.

1. Ve směru rotace nahradíte špatně vyvážený uzel potomkem.
2. Pokud by se stalo, že by nově dosazený vrchol měl 3 potomky, tak jeho původního potomka umístíte pod právě sesazený vrchol.

Možná to bude lépe vidět z následujícího obrázku, kam jsme přidali vrchol 93. Uzel, který nesplňuje požadavky AVL je 51 (pravá větev délky 3, levá 1 - rozdíl 2). Ten v kroku `a` zaměníme za 70, tím ale dostáváme pod 70 tři potomky. Jedná se o levou rotaci (typy rotací popsány níž), proto potomka 53 v kroku `b` hodíme nalevo. Hehe, ja vím, ta vedlejší věta důsledková teď nedává moc smysl, tak čistě úvahou - potřebujeme někam udat toho prostředního potomka 53, protože každý vrchol může mít maximálně dva potomky. Doprava pod 84 ho dát nemůžeme - je totiž menší než 70, a tak nemá v pravém stromu uzlu 70 co dělat. Proto ho dáme nalevo, pod 51.

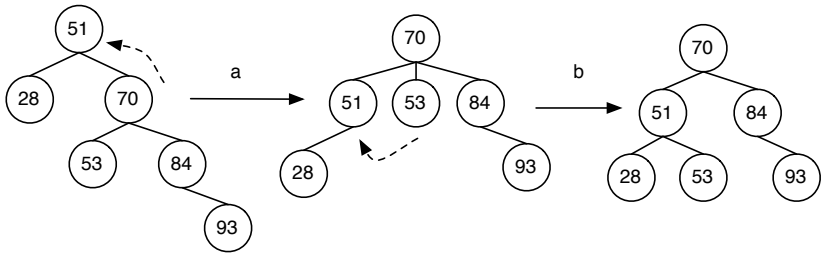


Figure 16.8: Ukázka L rotace

Na předchozím příkladu jsme ilustrovali levou rotaci. Kromě ní existuje i pravá rotace a kombinace obou. Kdy jakou použít? Od přidaného (nebo smazaného, viz dále) uzlu postupujeme směrem ke kořenu a aktualizujeme hloubky podstromů v každém navštíveném uzlu.

- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli dvěma hranami doleva nahoru, provedeme v tomto uzlu L rotaci.

  To byl náš příklad, od přidané 93 jsme postupovali nahoru, kde jsme narazili na rozvážený uzel 51. Doskákali jsme k němu dvakrát doleva nahoru.

- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli dvěma hranami doprava nahoru, provedeme v tomto uzlu R rotaci.

- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli hranami doleva a pak doprava nahoru, provedeme v tomto uzlu LR rotaci (tedy nejprve levou a pak pravou).

- Když narazíme na rozvážený uzel, do kterého jsme bezprostředně došli hranami doprava a pak doleva nahoru, provedeme v tomto uzlu RL rotaci.

Po provedení jedné rotace je AVL strom opět vyvážen.

Odstranění uzlu

Opět totožné jako u odstranění v BVS 16.5.1.1. Poté postupujeme od místa smazání nahoru ke kořeni a aktualizujeme výšky podstromů v každém uzlu.

16.5.2 B-strom

Grafické zpracovaní B-stromu z předmětu A4B36ALG spolecna/15/b-tree.pdf

B-strom je druh stromu, který ale **nesouvisí** s binárními stromy. Jeho definice

- Kořen má nejméně dva potomky, pokud není listem
- Každý uzel kromě kořene a listu má nejméně $\lceil \frac{n}{2} \rceil$ a nejvýše n potomků.
- Každý uzel kromě kořene má nejméně $\lceil \frac{n}{2} \rceil - 1$ a nejvýše $n - 1$ položek.
- Všechny cesty od kořene k listům jsou stejně dlouhé

B-strom je díky těmto vlastnostem vyvážený, operace přidání, vyjmutí i vyhledávání tedy probíhají v logaritmickém čase.

Na následujícím diagramu je ukázka b-stromu řádu 5. Řád určuje, kolik může mít vrchol maximálně potomků, v tomto případě 5. Položek může být maximálně (5-1), tedy 4.

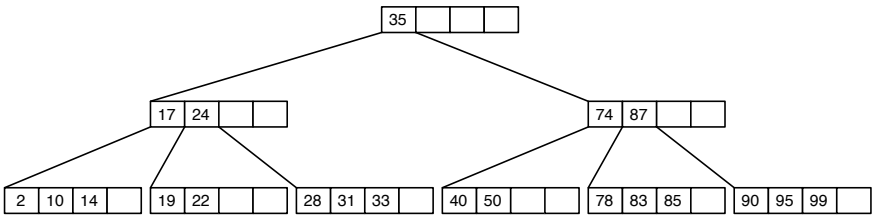


Figure 16.9: Ukázka b-stromu

Operace vyhledávání, přidávání a odstraňování uzlu b-stromu jsou graficky popsány v souboru `spolecna/15/b-tree.pdf`. Jde o to po každé operaci udržovat vlastnosti b-stromu buďto „sléváním" nebo „rozdělováním" vrcholů.

16.5.3 Srovnání stromů

	BVS s n uzly		AVL s n uzly	B-strom s n uzly
Operace	Vyvážený	Možná nevyvážený	Vyvážený	Vyvážený
Find	$O(\log(n))$	$O(n)$	$O(\log(n))$	$O(\log(n))$
Insert	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$	$\Theta(\log(n))$
Delete	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$	$\Theta(\log(n))$

Pár poznámek na závěr

- označení binární strom je nadmnožina pro binární vyhledávací strom a AVL strom

Figure 16.10: Vlastnosti stromů

Vyváženost zajišťuje AVL stromu lepší asymptotickou složitost. V tabulce srovnávající jednotlivé stromy je uveden sloupec „vyvážený binární vyhledávací strom", nicméně to, že náš BVS na vstupu bude skutečně vyvážený, nemáme zaručeno. U AVL stromu to zaručeno je.

- B-strom a AVL strom mají stejnou složitost, každý se ale hodí v jiné situaci. Využití b-stromu může být v aplikacích, kdy není celá struktura uložena v paměti RAM, ale v nějaké sekundární paměti, jako je pevný disk (například databáze). Protože přístup do tohoto typu paměti je náročný na čas (hlavně vyhledání náhodné položky), snažíme se minimalizovat počet přístupů do této paměti.

  #### Příklad

  Máme-li B-strom hloubky 2 a počet potomků každého uzlu je 1001, můžeme do něj uložit milion klíčů a ke každé položce se dostaneme maximálně po dvou diskových operacích.

Algoritmizace Hashing

June 5, 2012

Hashovací tabulky jsou datové struktury, které používají hashovací funkci k vyhledávání dat. Hashovací funkce $h(k)$ zobrazuje množinu klíčů k do intervalu adres $\langle a_{min}, a_{max} \rangle$.

- Volba hashovací funkce pro celá čísla:
  - Multiplikativní (m je prvočíslo)

$$h(k) = round(\frac{k}{2^w \cdot m}) \tag{1}$$

  - Modulární

$$h(k) = k\%n \tag{2}$$

  - Kombinovaná

$$h(k) = round(c \cdot k)\%m, c \in \langle 0, 1 \rangle \tag{3}$$

- Pro řetězce:
  - Hornerovo schéma

$$h(k) = k_n \cdot a^n + ... + k_0 \cdot a^0 \tag{4}$$

0.1 Řešení kolizí

Kolize vzniká když $h(k)$ vrací pro různé klíče stejnou hodnotu.

0.1.1 Hashování se separovanými řetězci

Hlavní myšlenka je, že pro každou možnou adresu vytvoříme spojový seznam a do něj přidáváme všechny prvky s danou adresou. Demonstrace v A.1.

- Výhody
  - Nemusíme znát předem počet prvků
- Nevýhody
  - Potřebujeme dynamické přidělování paměti
  - Potřebuje navíc paměť na ukazatele a na tabulku (heads v Figure A.1)

Složitost operací:

- Vkládání $O(1)$ (nebo složitost vložení do struktury kterou používáme místo spojového seznamu)
- Vyhledání $O(n)$ (nejhorší případ kdy jsou všechny prvky v kolizi)

0.1.2 Otevřené hashování

Hlavní myšlenka je, že pokud mám zhruba odhad počtu prvků které budu vkládat, tak mohu ušetřit na zbytečném ukládání ukazatelů tím, že budu ukládat kolize rovnou do tabulky. To může být provedeno následujícími způsoby:

- Lineární přidávání (když nastane kolize prvek se vloží na další volnou pozici). Demonstrace v Figure A.2.

  Složitost operací:

  - Vkládání $O(n)$ (nejhorší případ kdy jsou všechny prvky v kolizi)
  - Vyhledání $O(n)$ (nejhorší případ kdy jsou všechny prvky v kolizi)

- Dvojité hashování (když nastane kolize použije se druhá hashovací funkce, pomocí níž se spočítá offset a prvek se vloží na pozici kam měl být dán + offset). Demonstrace v Figure A.3.

  Složitost operací:

  - Vkládání $O(n)$ (nejhorší případ kdy jsou všechny prvky v kolizi v první i druhé hash funkci)
  - Vyhledání $O(n)$ (nejhorší případ kdy jsou všechny prvky v kolizi v první i druhé hash funkci)

0.1.3 Srůstající hashování

Hlavní myšlenka je omezení clusterování otevřeného hashování přidáním ukazatele pro každý prvek v tabulce. Možnosti provedení:

- Bez pomocné paměti (prvky se vkládají na pozice v tabulce)
  - LISCH (late insert standard coalesced hashing)
  - EISCH (early insert standard coalesced hashing)
- S pomocnou pamětí (prvky se vkládají na k tomu určené místo tzv. sklep za adresovým prostorem tabulky)
  - LICH (late insert coalesced hashing)
  - EICH (early insert coalesced hashing)
  - VICH (variable insert coalesced hashing)

LISCH

Přidává myšlenku ukazatele na poslední volné místo v tabulce (inicializován na poslední pozici). Při kolizi se prvek přidá na místo kam ukazuje tento ukazatel a ukazatel se opět přesune na poslední volné místo. Prvek na pozici kde vznikla kolize si drží ukazatel na pozici kam byl vložen kolidující prvek, ten si zase drží odkaz na další prvek který má stejný hash atd. Demonstrace ve Figurách A.4, A.5, A.6 a A.7. Porovnání s EISCH v Figure A.11

EISCH

Liší se od LISCH pouze tím, že ukazatele vedou odshora dolů (tzn ukazatel prvku který je na správné pozici vede vždy na jeho poslední kolizi, ta si zase drží odkaz na předchozí kolizi atd). Demonstrace ve Figurách A.8, A.9 a A.10. Porovnání s LISCH v Figure A.11

LICH

Stejný princip jako LISCH s tím rozdílem, že ukazatel začíná v tzv. sklepě což je speciální místo vyhrazené pro ukládání kolizí mimo adresní prostor tabulky. Porovnání ve Figure A.12.

EICH

Stejný princip jako EISCH s tím rozdílem, že ukazatel začíná v tzv. sklepě což je speciální místo vyhrazené pro ukládání kolizí mimo adresní prostor tabulky. Porovnání ve Figure A.12.

VICH

Algoritmus VICH připojuje prvek na konec řetězce, pokud řetězec končí ve sklepě, jinak na místo, kde řetězec opustil sklep. Porovnání ve Figure A.12.

0.1.4 Univerzální hashování

Místo jedná hashovací funkce máme konečnou množinu hashovacích funkcí H mapujících klíče do množiny $\{0, ..., m-1\}$. Množina funkcí H je univerzální, pokud pro každou dvojici různých klíčů $x, y \in U$ je počet hashovacích funkcí z množiny H, pro které $h(x) = h(y)$, nejvýše $|H|/m$. Pravděpodobnost kolize při náhodném výběru funkce $h(k)$ z množiny univerzálních hashovacích funkci H tedy není vyšší než pravděpodobnost kolize při náhodném a nezávislém výběru dvou stejných hodnot z intervalu $\{0, 1, \ldots, m-1\}$ tedy $1/m$. Při prvním spuštění programu jednu náhodně zvolíme. Funkci pak náhodně měníme jen v případě, že počet kolizí převyšuje přípustnou mez. V tomto případě je samozřejmě potřeba přehashovat celou tabulku.

Appendix A

Obrázky

A.1 Hashování se separovanými řetězci

- $h(k) = k \bmod 3$
- posloupnost : 1, 5, 21, 10, 7

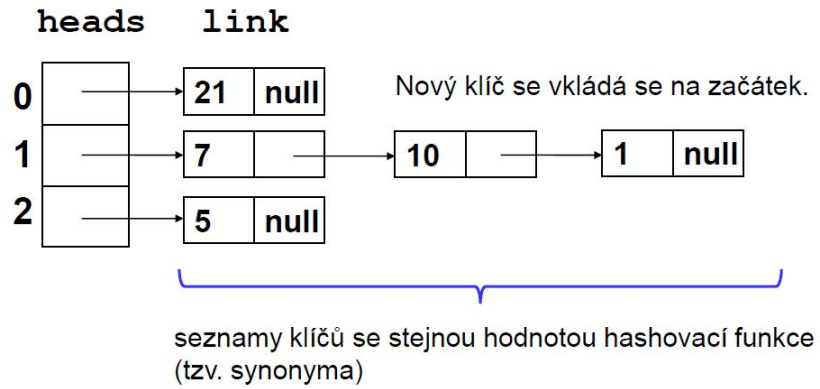


Figure A.1: Hashování se separovanými řetězci

A.2 Otevřená hashování

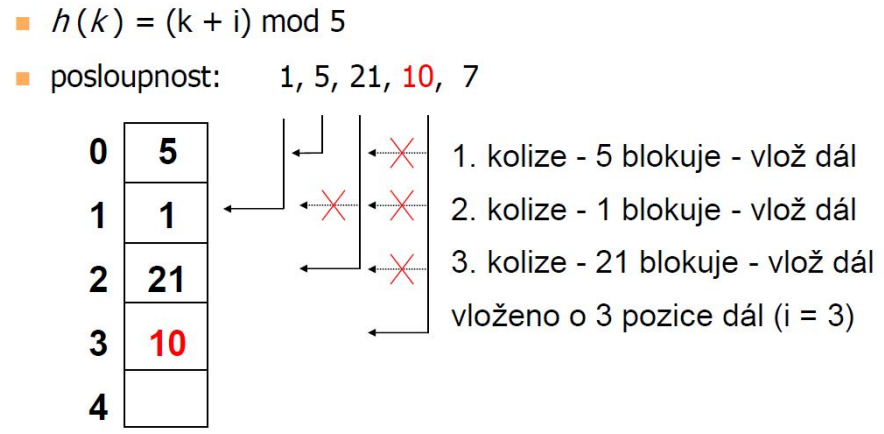


Figure A.2: Lineární přidávání

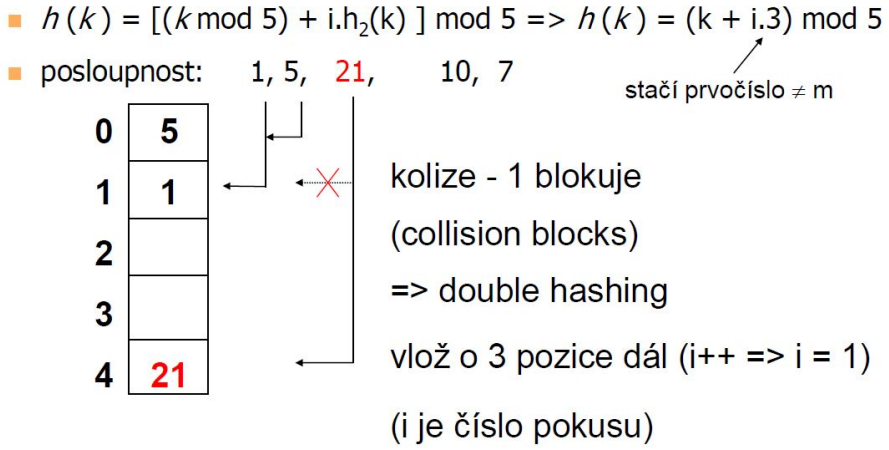


Figure A.3: Dvojité hashování

A.3 LISCH

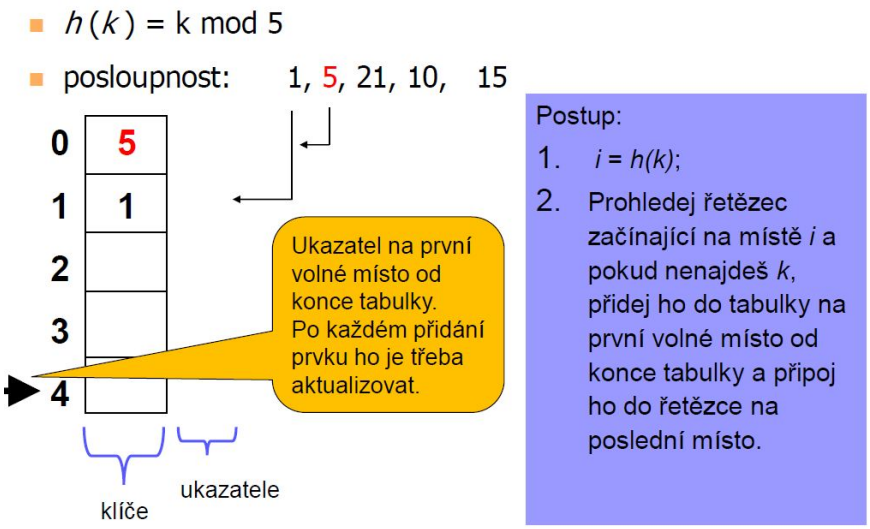


Figure A.4: LISCH 1

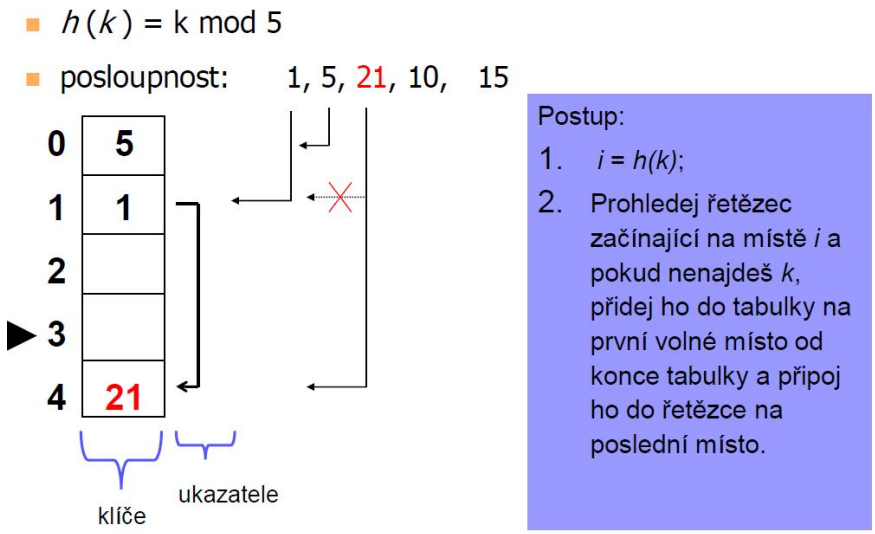


Figure A.5: LISCH 2

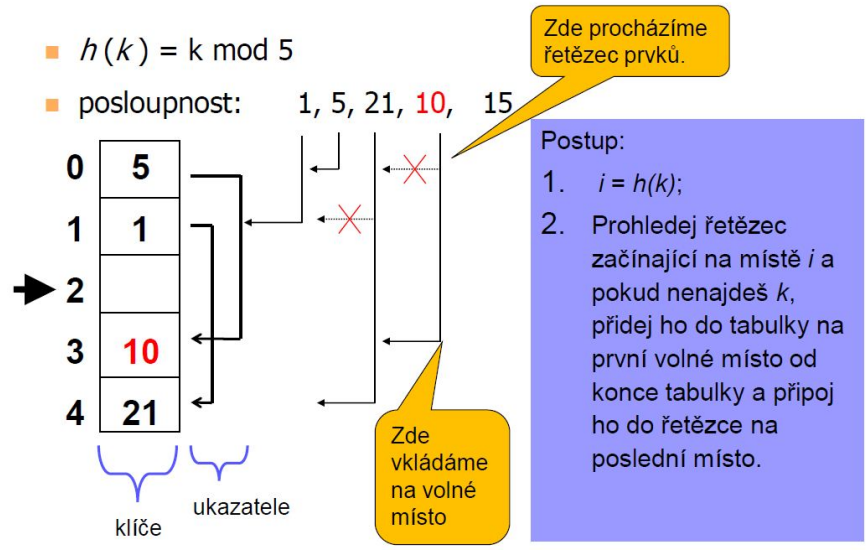


Figure A.6: LISCH 3

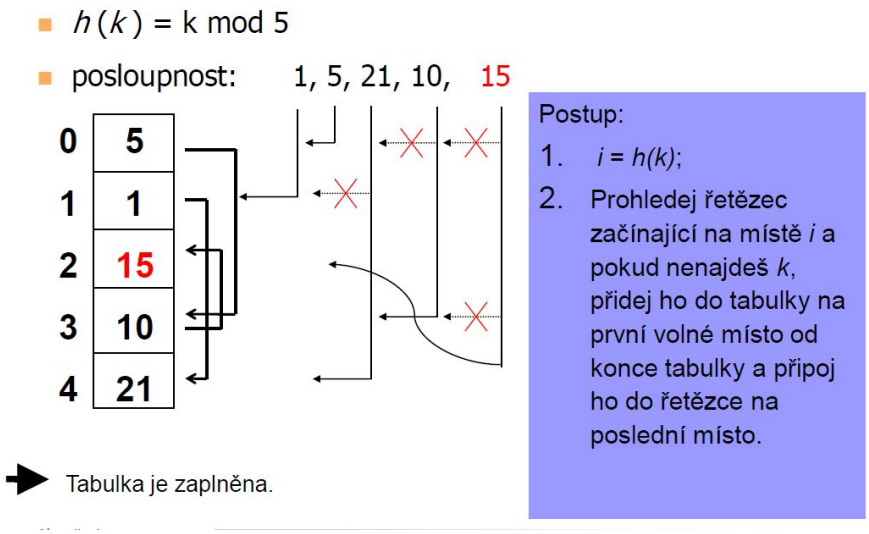


Figure A.7: LISCH 4

A.4 EISCH

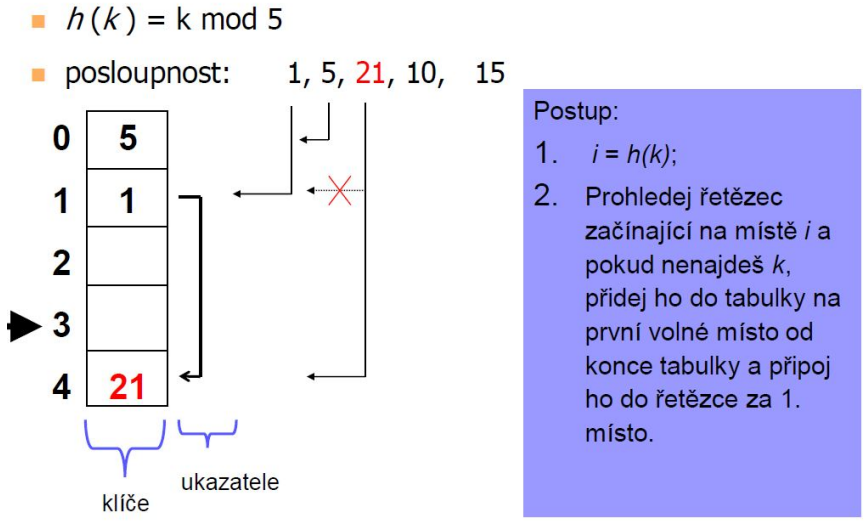


Figure A.8: EISCH 1

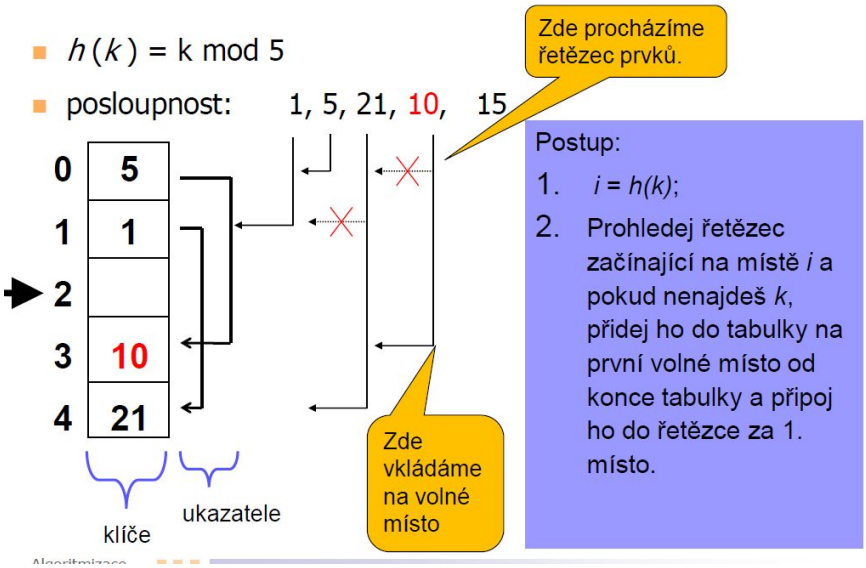


Figure A.9: EISCH 2

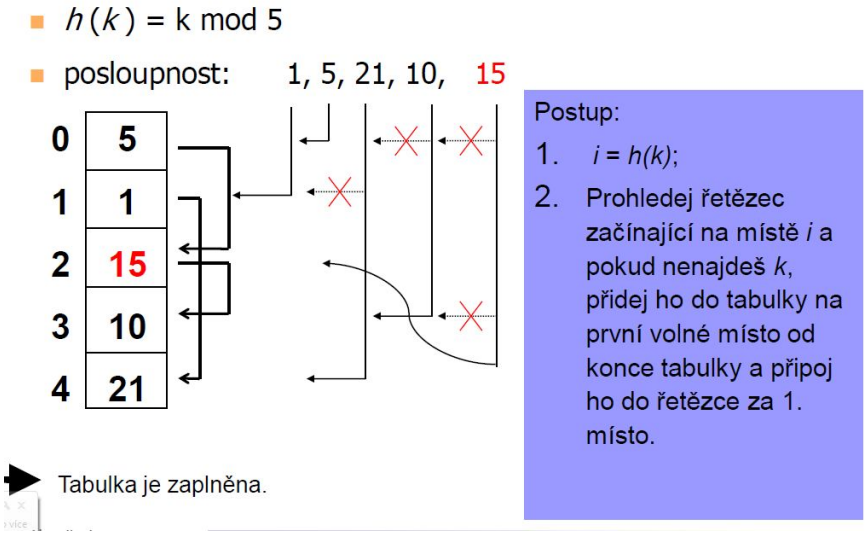


Figure A.10: EISCH 3

A.5 LISCH vs EISCH

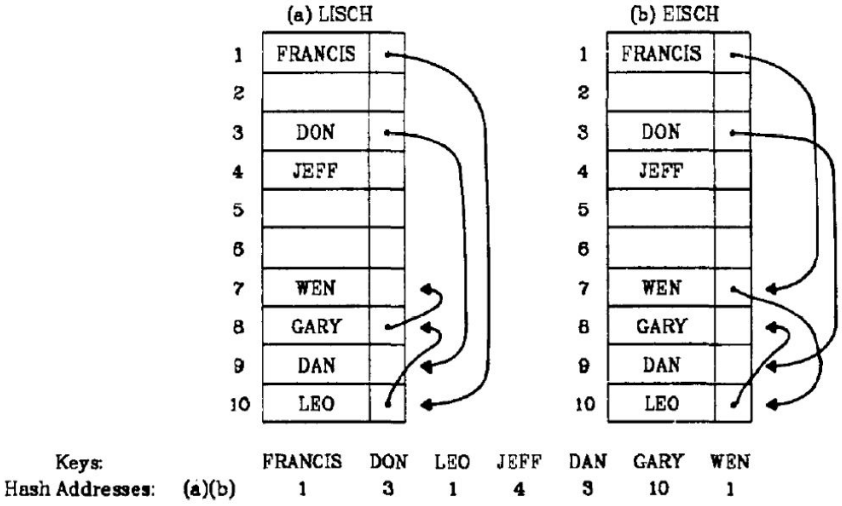


Figure A.11: LISCH vs EISCH

A.6 LICH vs EICH vs VICH

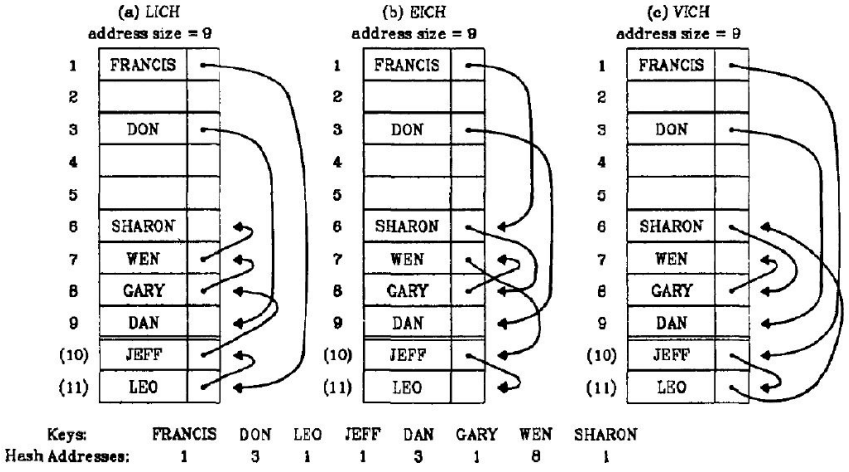


Figure A.12: LICH vs EICH vs VICH

18 Rekurze, základní schéma, základní vlastnosti, souvislost rekurze a iterace, vlastnosti implementace, efektivita. (A4B36ALG)

7.června 2012, Václav Burda

Co říci na začátek:
Rekurze nám dáva možnost definice něčeho nekonečného konečným algoritmem. Rekurzivní volání funkce je tedy takové volání, ke kterému dojde ve chvíli, kdy funkce samotná ještě nedokončila svou činnost.

18.1 Základní schéma

1. inicializační algoritmus - rekurzivní funkce většinou potřebuje nějakou inicializační hodnotu, se kterou zahájí výpočet. To většinou zajišťuje jiná funkce, která není rekurzivní a která danou rekurzivní funkci volá a předává hodnotu, se kterou rekurzivní výpočet začíná
2. kontrola, zda vstupní parametr odpovídá stanoveným podmínkám. Pokud hodnota parametru odpovídá, rekurzivní funkce provede výpočet a vrátí hodnotu
3. rekurzivní funkce redefinuje řešení problému tak, že jej nějakým způsobem zmenší, zjednoduší nebo rozloží na dílčí podproblémy
4. volání funkcí, které řeší daný podproblém - tady nastává přímé nebo nepřímé volání sebe sama
5. sestavení výsledku
6. vrácení vypočtené hodnoty

18.2 Základní vlastnosti

Dělení rekurze 1:

- Lineární - v každé iteraci dojde k pouze jednomu zavolání sebe sama.

- Stromová - v každé iteraci dochází k více dělení na podprogramy.

Dělení rekuzre 2:

- Přímá - v dáné rekurzivní funkci dojde k volání téže funkce.
- Nepřímá - rekurzivní volání, ke kterému dochází přes další funkci (např. A volá B a B volá A).

Volání může probíhat přímo nebo nepřímo. Každá rekurze se nechá přepsat na nerekurzivní tvar například pomocí zásobníku (udržujeme si datovou strukturu, která nahrazuje implicitní hardwarový zásobník použitý při rekurzivním volání).

18.3 Vlastnosti implementace

Rekurzí lze dosáhnout zjednodušení řešené úlohy, ale je nutné brát v úvahu následující úskalí:

- Každé další volání rekurzívní funkce prohlubuje zapouzdření a tedy zabírá paměťový prostor a procesorový čas.
- V případě chybně ošetřeného ukončení rekurze dochází k vučerpání veškérých volných prostředků a k havárii programu.
- Použití může vést ke zvýšení složitosti výpočtu.

Rekurze musí obsahovat:

- Ukončovací podmínku.
- V každém kroku rekurze musí dojít ke zjednodušení problému.
- V algoritmu se nejprve musí ověřit, zda nenastala koncová situace. Když ne, provede se rekurzivní krok.

18.4 Souvislost rekurze a iterace

Příklad výpočtu Faktoriálu pomocí rekurze:

```
function Faktorial ( integer X )
   if X < 0 then return "Chybny argument" end if
   if X = 0 then return 1 end if
   return Faktorial(X-1) * X
end function
```

a pomocí iterace:

```
function Faktorial (integer X)
     integer nfact
     if X < 0 then return "Chybny argument" end if
     nfact = 1
     for i = 1 to X do
           nfact = nfact * i
     end for
     return nfact
end function
```

18.5 Efektivita

Hlavní nevýhodou rekurze je u stromové struktury několikanásobné počítání stejných výpočtů (v případě Fibonacciho posloupnosti jde o exponenciální složitost). Tento nedostatek lze vyřešit zavedením tabulky (pole), do které ukládáme již spočítané výsledky (ale potřebujeme další paměť pro tuto tabulku).

18.6 Zdroje

- http://cs.wikipedia.org/wiki/Rekurze#Efektivita_rekurzivn.C3.ADch_algoritm.C5.AF
- http://cs.wikipedia.org/wiki/Rekurzivní_funkce_(programování)
- http://service.felk.cvut.cz/courses/Y36ALG/balik/Alg5Balik2008.pdf
- http://www.algoritmy.net/article/23275/Rekurze-11

Společná část: 19 - PSI

Náhodná veličina a náhodný vektor. Distribuční funkce, hustota a pravděpodobnostní funkce náhodné veličiny. Střední hodnota a rozptyl náhodné veličiny a jejich odhady. Sdružené charakteristiky náhodného vektoru. Korelace a nezávislost náhodných veličin. Metoda maximální věrohodnosti. Základní principy statistického testování hypotéz. Markovské řetězce, klasifikace stavů.

1 Základní pojmy pravděpodobnosti

1.1 Laplaceova (klasická) pravděpodobnost

- **Náhodný pokus** má $n \in \mathbb{N}$ různých, vzájemně se vylučujících výsledků, které jsou stejně možné.
- **Elementární jevy** = výsledky nahodného pokusu
- **Množina všech elementárńich jevů**: Ω
- **Jev** je podmnožina všech elementárních jevů ($A \subseteq \Omega$)
- **Pravděpodobnost jevu** A :
$$P(A) = \frac{|\ A\ |}{|\ \Omega\ |}$$
- **Jevové pole**: všechny jevy pozorovatelné v náhodném pokusu, zde expΩ (=množina všech podmnožin množiny Ω)

1.2 Kolmogorovova pravděpododobnost

- **Elementárních jevů** (=prvků množiny Ω) může být nekonečně mnoho, nemusí být stejně pravděpodobné
- **Jevy** jsou podmnožiny množiny Ω, ale ne nutně všechny. Tvoří podmnožinu $\mathcal{A} \subseteq exp\Omega$, která splňuje podmínky σ-algebry (viz. 1.3).
- **Pravděpodobnost** není určena strukturou jevů jako u Laplaceova modelu, je to funkce $P : \mathcal{A} \to \langle 0, 1\rangle$, splňující podmínky:

  (P1) $P(\mathbf{1}) = 1$,

  (P2)$P\left(\bigcup_{n\in\mathbb{N}} A_n\right) = \sum_{n\in\mathbb{N}} P(A_n)$, pokud jsou množiny (=jevy) A_n, $n \in \mathbb{N}$, po dvou neslučitelné
- **Pravděpodobnostní prostor** je trojice $(\Omega, \mathcal{A}, P)$, kde Ω je neprázdná množina, $\mathcal{A}$ je σ-algebra podmnožin množiny Ω a P je pravděpodobnost.

1.3 σ-algebra

σ-algebra je teoretický koncept výběru jistých podmnožin dané množiny, který splňuje pevně definované podmínky. Koncept σ-algebry umožňuje například zavést míru, čehož se dále využívá zejména v matematické analýze k budování pojmu integrál a právě v teorii pravděpodobnosti [wikipedia]. Systém podmnožin $\mathcal{A}$ nějáké množiny Ω musí splňovat podmínky:

1. $\emptyset \in \mathcal{A}$
2. $A \in \mathcal{A} \Rightarrow \bar{A} \in \mathcal{A}$ (uzavřenost vůči doplňku)

3. $(\forall n \in \mathbb{N} : A_n \in \mathcal{A}) \Rightarrow \bigcup_{n\in\mathbb{N}} A_n \in \mathcal{A}$) (uzavřenost vůči sjednocení)

Nejmenší σ-algebra podmnožin $\mathbb{R}$, která obsahuje všechny intervaly, se nazývá **Borelova σ-algebra**. Obsahuje všechny intervaly otevřené, uzavřené i polouzavřené, i jejich spočetná sjednocení, a některé další množiny, ale je menší´než exp$\mathbb{R}$. Její prvky nazýváme borelovské množiny.

2 Náhodná veličina a náhodný vektor

2.1 Náhodná veličina

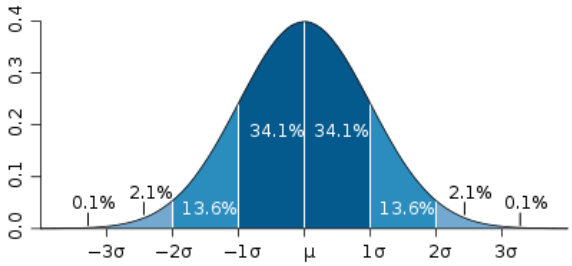
Je na pravděpodobnostním prostoru $(\Omega, \mathcal{A}, P)$ měřitelná funkce $X : \Omega \to \mathbb{R}$ (přiřazuje každému jevu jevového pole reálné číslo [wikipedia]).
Náhodné veličiny lze rozdělit na nespojité (diskrétní) a spojité. Diskrétní veličiny mohou nabývat pouze spočetného počtu hodnot (konečného i nekonečného), zatímco spojité veličiny nabývají hodnoty z nějakého intervalu (konečného nebo nekonečného) [wikipedia].
Příklad: Havárie aut označíme cenou škody a můžeme se ptát, jak je pravděpodobné, že havárie dosáhne určité škody.
Pro každý interval I platí

$$X^{-1}(I) = \{\omega \in \Omega \mid X(\omega) \in I\} \in \mathcal{A}$$

Popisuje ji **Rozdělení pravděpodobnosti** náhodné veličiny X:

$$P_X(I) = P[X \in I] = P(\{\omega \in \Omega \mid X(\omega) \in I\})$$

to je funkce, která učuje pravděpodobnost toho, že náhodná veličina nabyde určité hodnoty.

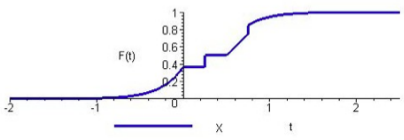


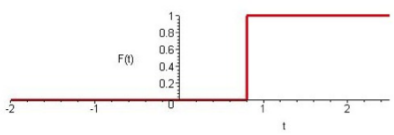
Místo pravděpodobnostní funkce, můžeme použít úspornější **Distribuční funkci** (F_X), která se omezuje na intervaly tvaru $I = (-\infty, t\rangle, t \in \mathbb{R}$

$$P[X \in (-\infty, t\rangle] = P[X \leq t] = P_X((-\infty, t\rangle) = F_X(t)$$

Různými kombinacemi distribuční funkce ($F_X : \mathbb{R} \to \langle 0, 1\rangle$) můžeme plně nahradit pravděpodobnostní funkci.
Vlastnosti distribuční funkce:

- neklesající
- zprava spojitá
- $\lim_{t\to-\infty} F_x(t) = 0$, $\lim_{t\to\infty} F_x(t) = 1$





Distribuční funkce pro absolutně spojitou veličinu:

$$F_X(t) = \int_{-\infty}^{t} f_X(u)du,$$

kde f_X je tzv. **hustota náhodné veličiny**. Je to nezáporná funkce ($f_X : \mathbb{R} \to \langle 0, \infty)$) a splňuje $\int_{-\infty}^{\infty} f_X(u)du = 1$

Náhodné veličiny $X_1, \ldots, X_n$ jsou **nezávislé**, pokud pro všechny intervaly $I_1, \ldots, I_n$ jsou jevy $X_1 \in I_1, \ldots, X_n \in I_n$ nezávislé, tj.

$$P\left[X_1 \in I_1, \ldots, X_n \in I_n\right] = \prod_{i=1}^{n} P\left[X_i \in I_i\right]$$

2.2 Náhodný vektor (n-rozměrná náhodná veličina)

Je na pravděpodobnostním prostoru $(\Omega, \mathcal{A}, P)$ měřitelná funkce $\boldsymbol{X} : \Omega \to \mathbb{R}^n$. Používáme ho v případech, kdy je k popisu výsledku náhodného pokusu nutné použít více čísel [wikipedia].
Příklad: Chceme popsat vztah např. mezi výškou a váhou osob. K tomu potřebujeme více informací nežjen popis jednotlivých náhodných veličin.
Pro každý n-rozměrný interval I platí

$$\boldsymbol{X}^{-1}(I) = \{\omega \in \Omega \mid \boldsymbol{X}(\omega) \in I\} \in \mathcal{A}$$

Lze psát

$$\boldsymbol{X}(\omega) = (X_1(\omega), \ldots, X_n(\omega)),$$

kde zobrazení $X_k : \Omega \to \mathbb{R}, k = 1, \ldots, n$ jsou náhodné veličiny.
Náhodný vektor lze považovat za vektor náhodných veličin $\boldsymbol{X} = (X_1, \ldots, X_n)$.
Je popsán **Sdruženým rozdělením pravděpodobnosti**:

$$P_{\boldsymbol{X}}(I_1 \times \ldots \times I_n) = P[X_1 \in I_1, \ldots, X_n \in I_n] = P(\{\omega \in \Omega \mid X_1(\omega) \in I_1, \ldots, X_n(\omega) \in I_n\}),$$

kde $I_1, \ldots, I_n$ jsou intervaly v $\mathbb{R}$. Z toho výplývá pravděpodobnost pro libovolnou borelovskou množinu I v $\mathbb{R}^n$

$$P_{\boldsymbol{X}}(I) = P[\boldsymbol{X} \in I] = P(\{\omega \in \Omega \mid \boldsymbol{X}(\omega) \in I\})$$

Opět můžeme použít úspornější **Sdruženou distribuční funkci ($F_{\boldsymbol{X}}$)**

$$P[X_1 \in (-\infty, t_1\rangle, \ldots, X_n \in (-\infty, t_n\rangle] = P[X_1 \leq t_1, \ldots, X_n \leq t_n] = P_{\boldsymbol{X}}((-\infty, t_1\rangle \times \ldots \times (-\infty, t_n\rangle) = F_{\boldsymbol{X}}(t_1, \ldots, t_n)$$

Vlastnosti distribuční funkce:

- neklesající (ve všech proměnných)
- zprava spojitá (ve všech proměnných)
- $\lim_{t_1 \to \infty, \ldots, t_n \to \infty} F_{\boldsymbol{X}}(t_1, \ldots, t_n) = 1$
- $\lim_{t_1 \to -\infty, \ldots, t_n \to -\infty} F_{\boldsymbol{X}}(t_1, \ldots, t_n) = 0$

2.3 Obecné náhodné veličiny

Náhodné veličiny nemusí být reprezentovány pouze reálným čísly, ale třeba i čísly komplexními. V některých případech se používájí i jiné něž numerické hodnoty, například “rub”, “líc”, ”kámen”, ”papír” atp.

3 Základní charakteristiky náhodných veličin a náhodných vektorů

3.1 Střední hodnota

Značení E nebo μ. Jedná se o tzv. charakteristiku polohy (angl. measures of central tendency)

3.1.1 Střední hodnota náhodné veličiny

Je definována zvlášť pro:

- **diskrétní** náhodnou veličinu U s oborem hodnot $\mathbb{R}$:

$$EU = \mu_U = \sum_{t \in \mathbb{R}} t \cdot p_U(t)$$

- **spojitou** náhodnou veličinu V:

$$EV = \mu_V = \int_{-\infty}^{\infty} t \cdot f_V(t) dt$$

3.1.2 Střední hodnota náhodného vektoru

$$E\boldsymbol{X} = (EX_1, \ldots, EX_n)$$

3.2 Rozptyl (disperze)

Značení σ^2 , D , var. Jedná se o charakteristiku variability (angl. measures of central tendency).

3.2.1 Rozptyl náhodné veličiny

Je to vlastně střední hodnota kvadrátu odchylky od střední hodnoty.

$$DX = E\left((X - EX)^2\right) = E\left(X^2\right) - (EX)^2$$

3.2.2 Rozptyl náhodného vektory

$$D\boldsymbol{X} = (DX_1, \ldots, DX_n)$$

3.3 Další číselné charakteristiky náhodného vektoru

Pro jednorozměrné náhodné veličiny střední hodnota a rozptyl dávají **dostatečnou** informaci pro výpočet rozptylu jeho lineárních funkcí (lineární kombinace různých náhodných veličin):

$$E(X + Y) = EX + EY$$
$$E(X - Y) = EX - EY$$
$$D(X + Y) = DX + DY$$
$$D(X - Y) = DX + DY$$

Pro náhodný vektor to ale nestačí, a proto zavádíme další charakteristiky:

$$E(X + Y) = EX + EY$$

$$D(X + Y) = DX + DY + 2cov(X, Y),$$

kde $cov(X, Y)$ je kovariance náhodných veličin X, Y.

3.3.1 Kovariace a korelace

Kavariace určuje míru statistické závislosti mezi náhodnými veličinami. Je definována jako střední hodnota součinu odchylek obou náhodných veličin X, Y od jejich středních hodnot (druhý vzorec není tak srozumitelný, ale je jednodušší pro výpočet)

$$cov(X,Y) = E((X - EX)(Y - EY)) = E(XY) - EXEY$$

Vlastnosti kovariace:

$$cov(X,X) = DX,$$
$$cov(Y,X) = cov(X,Y)$$
$$cov(aX + b, cY + d) = a\,c\,cov(X,Y)$$
$$cov(X,Y) = 0 - pro\ nezávislé\ veličiny$$

Při výpočtech je místo kovariace výhodnější používat **korelaci** (což je kovariace pro normované náhodné veličiny)

$$\varrho(X,Y) = cov(norm\,X, norm\,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = E(norm\,X \cdot norm\,Y)$$

Korelace nabývá hodnot ⟨-1,-1⟩, pro $\varrho = 1$ je mezi X, Y přímá lineární závislost, pro $\varrho = -1$ nepřímá lineární závislost. Pro $\varrho = 0$ říkáme, že jsou veličiny **nekorelované**. Zároveň to znamená, že jsou lineárně nezávislé, nikoliv obecně nezávislé (je to nutná podmínka pro obecnou nezávislost, ale nikoliv postačující).
Pro náhodný vektor $\boldsymbol{X} = (X_1, \ldots, X_2)$ definujeme **kovariační matici:**

$$\Sigma_{\boldsymbol{X}} = \begin{bmatrix} cov(X_1,X_1) & cov(X_1,X_2) & \ldots & cov(X_1,X_n) \\ cov(X_2,X_1) & cov(X_2,X_2) & \ldots & cov(X_2,X_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(X_n,X_1) & cov(X_n,X_2) & \ldots & cov(X_n,X_n) \end{bmatrix} = \begin{bmatrix} DX_1 & cov(X_1,X_2) & \ldots & cov(X_1,X_n) \\ cov(X_2,X_1) & DX_2 & \ldots & cov(X_2,X_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(X_n,X_1) & cov(X_n,X_2) & \ldots & DX_n \end{bmatrix}$$

a **korelační matici:**

$$\varrho_{\boldsymbol{X}} = \begin{bmatrix} 1 & \varrho(X_1,X_2) & \ldots & \varrho(X_1,X_n) \\ \varrho(X_2,X_1) & 1 & \ldots & \varrho(X_2,X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \varrho(X_n,X_1) & \varrho(X_n,X_2) & \ldots & 1 \end{bmatrix}$$

4 Odhady základních číselných charakteristik

Typicky jsou číselné parametry rozdělení skryté a nebývají přímo měřitelné (kvůli velikosti celého souboru).
Výběrový soubor rozsahu n označujeme jako **náhodný výběr** $\boldsymbol{X} = (X_1, \ldots, X_n)$.
Statistika je každá měřitelná funkce, definovaná na náhodném výběru libovolného rozsahu. Následující statistiky se používají pro odhady číselných parametrů.

4.1 Výběrový průměr

z náhodného výběru $\boldsymbol{X} = (X_1, \ldots, X_n)$ je nestranný konzistentní odhad střední hodnoty:

$$\bar{X}_n = \frac{1}{n}\sum_{j=1}^{n} X_j$$

4.2 Výběrový rozptyl

náhodného výběru $\boldsymbol{X} = (X_1, \ldots, X_n)$ je nestranný konzistentní odhad střední rozptylu:

$$S_X^2 = \frac{1}{n-1}\sum_{j=1}^{n} \left(X_j - \bar{X}_n\right)^2$$

5 Metoda maximální věrohodnosti

Často nechceme odhadnout jen střední hodnotu či rozptyl, ale i jiné parametry rozdělení. Rozdělení náhodné veličiny X závisí na vektoru parametrů $\boldsymbol{\vartheta} = (\vartheta_1, \ldots, \vartheta_k) \in \Pi$, kde $\Pi \subseteq \mathbb{R}^k$ je **parametrický prostor,** tj. množina všech přípustných hodnot parametrů. Myšlenkou této metody je, že hledáme takové hodnoty parametrů, které by nejlépe vysvětlovali realizaci náhodného výběru, tj. při kterých by pozorované výsledky byly "nejméně nepravděpodobné".

5.1 Metoda maximální věrohodnosti pro diskrétní rozdělení

Nechť $\boldsymbol{x} = (x_1, \ldots, x_n)$ je realizace náhodného výběru z diskrétního rozdělení s pravděpodobnostní funkcí $p_X(.; \boldsymbol{\vartheta})$ závislou na vektoru parametrů $\boldsymbol{\vartheta} \in \Pi$. Pak definujeme věrohodnost realizace diskrétního rozdělení (angl. likelihood) $L : \Pi \to \langle 0, 1 \rangle$ vztahem

$$L(\boldsymbol{\vartheta}) = p_{\boldsymbol{X}}(\boldsymbol{x}; \boldsymbol{\vartheta}) = \prod_{j=1}^{n} p_X(x_j; \boldsymbol{\vartheta})$$

Je třeba rozlišovat mezi pravděpodobností a věrohodností! Pravděpodobnost je určena pro předpovídání výsledků budoucího pokusu při známém pravděpodobnostním modelu a tudíž má definiční obor jevy z nějáké σ-algebry. Naopak věrohodnost vyhodnocuje sérii pokusů již realizovaných a dovoluje jim přizpůsobit neznámé parametry pravděpodobnostního modelu, a proto je definována na prostoru Π všech možných hodnot parametrů rozdělení.

Metoda maximální věrohodnosti považuje za správný odhad parametrů takové hodnoty, které maximalizují

věrohodnost. Protože výpočet skoro vždy vede na hledání nulové derivace, často se maximalizujeme logaritmus věrohodnosti (je to jednodušší na derivování).

5.2 Metoda maximální věrohodnosti pro spojité rozdělení

Nechť $\boldsymbol{x} = (x_1, \ldots, x_n)$ je realizace náhodného výběru ze spojitého rozdělení se spojitou hustotou $f_X(.; \boldsymbol{\vartheta})$ závislou na vektoru parametrů $\boldsymbol{\vartheta} \in \Pi$. Pak definujeme věrohodnost realizace spojitého rozdělení $\Lambda : \Pi \to \langle 0, \infty \rangle$ vztahem

$$\Lambda(\boldsymbol{\vartheta}) = f_{\boldsymbol{X}}(\boldsymbol{x}; \boldsymbol{\vartheta}) = \prod_{j=1}^{n} f_X(x_j; \boldsymbol{\vartheta})$$

Metoda maximální věrohodnosti opět považuje za správný odhad parametrů takové hodnoty, které maximalizují věrohodnost.

20 Entropie, vzájemná entropie a podmíněná entropie diskrétních a spojitých rozdělení, základní vlastnosti a význam. Kódování zpráv, Kraftova-MacMillanova nerovnost. Souvislost entropie a střední délky kódového slova. Kódy s optimální střední délkou. Informační kanál a jeho kapacita. Shannonova věta o kódování. (A0B01PSI)

20.1 Entropie

20.1.1 Entropie diskrétního rozdělení

Za předpokladu, že X je diskrétní náhodná veličina s pravděpodobnostní funkcí p_X, je entropie diskrétní náhodné veličiny X rovna:

$$H(X) = -\sum_{x \in X} p_x(x) \log_2 p_x(x). \qquad (1.1)$$

Funkce H se nazývá Shannonova entropie nebo pouze entropie.

Příspěvek výsledku s pravděpodobností p je dán funkcí $\iota(p) = -p \log_2 p$. Ta je nezáporná a má nulové limity v 0 a 1. Důsledkem je, že entropie *diskrétního* náhodného pokusu je vždy nezáporná.

20.1.2 Vzájemná entropie diskrétního rozdělení

Dána vzorcem

$$H(X,Y) = -\sum_{x \in X} \sum_{y \in Y} p_{XY}(x,y) \log_2 p_{XY}(x,y).$$

Platí nerovnost

$$H(X,Y) \leq H(X) + H(Y).$$

20.1.3 Podmíněná entropie diskrétního rozdělení

Dána vzorcem

$$H(Y \mid X = x) = -\sum_{y \in Y} p_{Y|X}(y|x) \log_2 p_{Y|X}(y|x).$$

20.1.4 Entropie spojitého rozdělení

Entropie spojité náhodné veličiny X s hustotou pravděpodobnosti $f(x)$ je definována takto:

$$H(X) = \int_{-\infty}^{\infty} f(x) \log_2 f(x) dx.$$

Zatímco entropie diskrétního veličiny je **absolutní** mírou neurčitosti, ve spojité verzi je entropie **relativní** mírou neurčitosti vzhledem ke zvolenému sytému souřadnic.

20.1.5 Vzájemná entropie spojitého rozdělení

Dána vzorcem

$$H(X,Y) = -\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) \log_2 f(x,y) dx dy.$$

20.1.6 Podmíněná entropie spojitého rozdělení

Dány vzorci

$$H(Y \mid X) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) \log_2 \frac{f(x,y)}{g(x)} dx dy$$

a

$$H(X \mid Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) \log_2 \frac{f(x,y)}{h(x)} dx dy,$$

kde $f(x,y)$ je sdružená hustota pravděpodobnosti veličin x a y a *g(x), h(x)* jsou marginální hustoty pravděpodobnosti veličin x a y, pro které platí:

$$g(x) = \int_{-\infty}^{\infty} f(x,y) dy$$

a

$$h(y) = \int_{-\infty}^{\infty} f(x,y)dx.$$

20.1.7 Význam a vlastnosti entropie

Entropie je střední hodnota informace, kterou dostaneme, pokud se dovíme, který z k disjunktnch jevů nastal, jestliže jejich pravděpodobnosti byly p_1,..., p_k.

Shannonova entropie H:$D_h \to \langle 0, \infty \rangle$daná vzorcem (1.1) je jediná reálná funkce na D_h následujících vlastností:

1. H nezávisí na permutaci argumentů,
2. funkce $p \mapsto H(p, 1-p)$ je spojitá,
3. $H(1/2, 1/2) = 1$,
4. $H(p_1, p_2, p_3, ..., p_n) = H(p_1 + p_2, p_3, ..., p_n) + (p_1 + p_2)H(\frac{p_1}{p_1+p_2}, \frac{p_2}{p_1+p_2})$.

Entropie $H(X) = 0$ tehdy a jen tehdy, jsou-li všechy pravděpodobnosti kromě jedné rovny nule a jedna pravděpodobnost rovna jedné. Entropie dosahuje svého maxima, jsou li všechny pravděpodobnostu stejné.

20.2 Kódování zpráv

Definice: Nechť X je náhodná veličina s výběrovým prostorem $\mathcal{X}$. Nechť $\mathcal{D}$ je konečná abeceda a $\mathcal{D}^\star = \bigcup_{n=1}^{\infty} \mathcal{D}^n$. **Kód** pro X je zobrazení $\mathcal{C} : \mathcal{X} \to \mathcal{D}^*$.

Definice: **Střední délka** kódu C je $L(C) = \sum_{x \in X} p_X(x) l(C(x))$, kde $l(C(x))$ značí délku řetězce $C(x)$.

Příklad 1: Nechť $p_X(i) = \frac{1}{3}$, $i = 1, 2, 3$. Mějme tento binární kód:
$C(1) = 0$, $C(2) = 10$, $C(3) = 11$.
Zřejmě $L(C) = \frac{5}{3} = 1.\bar{6}$, přičemž $H(X) = \log 3 \doteq 1.58$.

Příklad 2: Nechť $p_X(i) = 2^{-i}$, $i = 1, 2, 3$ a $p_X(4) = 2^{-3}$. Mějme tento binární kód:
$C(1) = 0$, $C(2) = 10$, $C(3) = 110$, $C(4) = 111$.
Zřejmě $L(C) = H(X) = 1.75$.

20.2.1 Třídy kódů

Definice: Kód C je

- **nesingulární**, pokud je $\mathcal{C} : \mathcal{X} \to \mathcal{D}^*$prosté zobrazení.
- **jednoznačně dekódovatelný**, pokud je jeho rozšíření $\mathcal{C}^*$nesingulární, kde $\mathcal{C}^* : \mathcal{X}^* \to \mathcal{D}^*$ je definováno pomocí $\mathcal{C}^*(x_1...x_n) = \mathcal{C}(x_1)...\mathcal{C}(x_n)$, $x_1...x_n \in \mathcal{X}^*$.

- **instantní**, pokud žádné kódové slovo $\mathcal{C}(x)$ není počátečním úsekem kódového slova $\mathcal{C}(x')$ pro $x, x' \in \mathcal{X},\ x \neq x'$.

Vztahy mezi kódy:

1. Každý **instantní kód** je **jednoznačně dekódovatelný**.
2. Každý **jednoznačně dekódovatelý kód** je **nesingulární**.

	x	*singulární*	*nesingulární*	*jednoznačně d.*	*instantní*
	1	0	0	01	0
Třídy kódů	2	0	1	00	10
	3	0	00	11	110
	4	0	01	110	111

- neinstantní jednoznačně dekódovatelný kód obecně umožňuje dekódovaní až po přečtení **celého** rozšířeného kódového slova $C(x_1...x_n)$
- instantní kód umožňuje dekódování **ihned** po obdržení kódového slova

20.3 Kraftova-MacMillanova nerovnost

Věta (Kraft, 1949) Délky slov $l_1, ..., l_m$ libovolného **instantního** d-znakového kódu splňují nerovnost

$$\sum_{i=1}^{m} d^{-l_i} \leq 1.$$

Obráceně, splňují-li $l_1, ..., l_m \in \mathbb{N}$ tuto nerovnost, potom existuje **instantní** d-znakový kód s délkami slov $l_1, ..., l_m$.

Věta (MacMillan, 1956) Délky slov $l_1, ..., l_m$ libovolného **jednoznačně dekódovatelného** d-znakového kódu splňují nerovnost

$$\sum_{i=1}^{m} d^{-l_i} \leq 1.$$

Obráceně, splňují-li $l_1, ..., l_m \in \mathbb{N}$ tuto nerovnost, potom existuje **jednoznačně dekódovatelného** d-znakový kód s délkami slov $l_1, ..., l_m$.

Kombinací těchto dvou nerovností dostaneme:

Věta (Kraft-MacMillanova) Jednoznačně dekódovatelné kódování s předepsanou délkou slov existuje právě tehdy, když existuje instantní kó se stejnou délkou.

20.4 Souvislost entropie a střední délky kódovaného slova

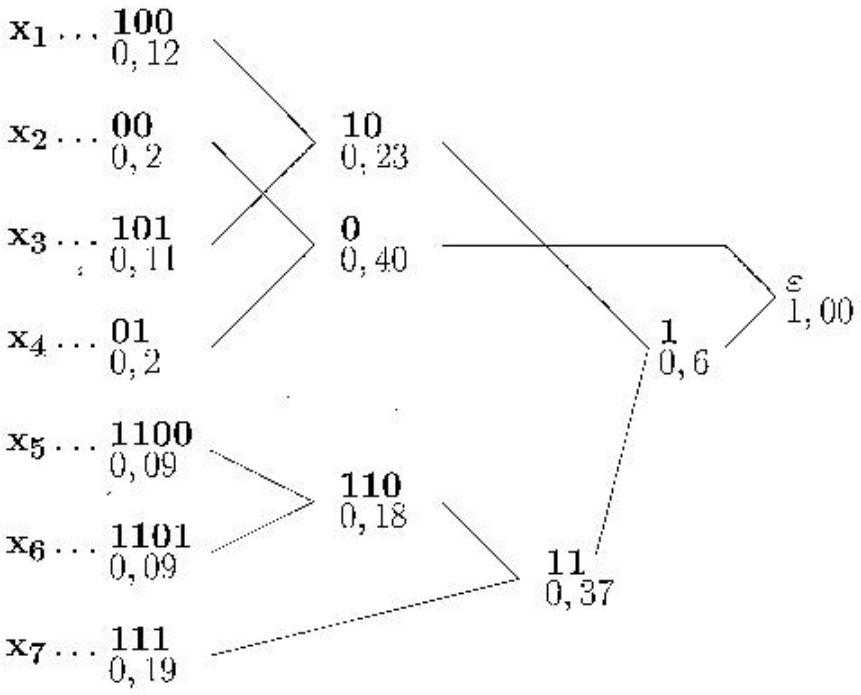
Jsou-li pravděpodobnosti výsledků známé a různé, můžeme to využít k úspornějšímu kódování. U úsporného kódu budou všechny použité znaky přibližně stejně pravděpodobné. Entropie určuje teoretickou dolní mez průměrné délky zprávy při nejúspornějším kódování. Tím nám ukazuje meze možné komprese dat. Častěji známe rozdělení jen přibližně, a proto nedosáhneme maximální účinnosti komprese.

20.5 Kódy s optimální střední délkou

20.5.1 Huffmanův kód

- Je to optimální kód, tedy instantní kód minimalizující střední délku na množině všech jednoznačně dekódovatelných kóků.
- Výsledný kód není určen jednoznačně (např. bitovou inverzí získáme jiný optimální kód).
- Jednoznačně není určena ani délka kódových slov.
- Aplikace : závěrečné zpracování formátů JPEG, MP3, DEFLATE, PKZIP

Příklad:



20.5.2 LZ algoritmy

- autoři Lempel a Ziv publikovali 2 základní varianty algoritmu, a to LZ77 a LZ78
- algoritmus má mnoho různých variací, jako je např. Lempel-Ziv-Welsch LZW (komprese v Unixu, ZIP, RAR, GIF, PDF,...)
- jde o třídu adaptivních kompresních algoritmů se slovníkem
- **optimalita**: LZ77 a LZ78 asymptoticky dosahují rychlosti entropie pro stacionární ergodické zdroje

LZ78 (Stromová verze LZ)

- řetězec $x_1...x_n \in X^n$ je sekvenčně testován na výskyt **nejkratších řetězců**, které se nevyskytly v předchozím kroku
- každý takový řetězec je označen a uložen do **slovníku**
- díky minimalitě ukládaného řetězce jsou jeho **prefixy** již ve slovníku: řetězec $xi_i...x_k$ byl uložen do slovníku před $x_i...x_kx_{k+1}$

Kód tvoří posloupnost dvojic (U_k, x_k), kde

- $x_k$je poslední znak řetězce $x_i...x_lx_k$,
- $U_k$je ukazatel na odpovídající prefix $x_i...x_l$.

Příklad

$X = \{A, B\}$, řetězec ABBABBABBBAABABAA

Dostaneme následující řetězce: A, B, BA, BB, AB, BBA, ABA, BAA

Výsledný kód: 0A 0B 2A 2B 1B 4A 5A 3A

20.6 Informační kanál a jeho kapacita

Definice

Informační kanál je trojice $K = \langle X, \mathbf{P}, Y \rangle$, kde

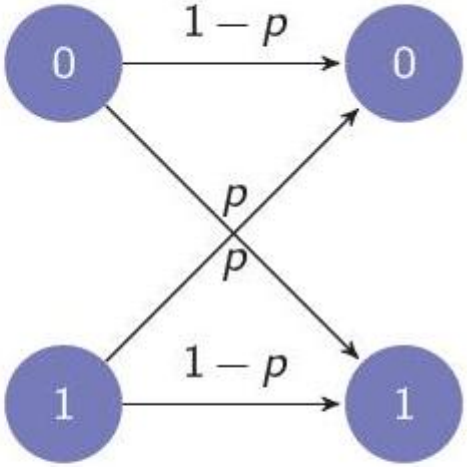
- X je m-prvková vstupní abeceda
- Y je n-prvkovávýstupní abeceda
- $\mathbf{P}$ je matice $m \times n$ podmíněných pravděpodobností:

$$\mathbf{P} = \begin{pmatrix} p_{Y|X}(y_1|x_1) & p_{Y|X}(y_2|x_1) & \dots & p_{Y|X}(y_n|x_1) \\ p_{Y|X}(y_1|x_2) & p_{Y|X}(y_2|x_2) & \dots & p_{Y|X}(y_n|x_2) \\ & \dots\dots\dots\dots\dots\dots & & \\ p_{Y|X}(y_1|x_m) & p_{Y|X}(y_2|x_m) & \dots & p_{Y|X}(y_n|x_m) \end{pmatrix}$$

Příklad

Binární symetrický kanál

$$\mathcal{X} = \mathcal{Y} = \{0, 1\}$$



$$\mathbf{P} = \begin{pmatrix} 1-p & p \\ p & 1-p \end{pmatrix}$$

Vstup a výstup kanálu

Vstup: Informační zdroj X s pravděpodobnostmi

$$(p_X(x_1) \dots p_X(x_m))$$

Výstup: Informační zdroj Y s pravděpodobnostmi

$$(p_Y(y_1) \dots p_Y(y_n)) = (p_X(x_1) \dots p_X(x_m)).\mathbf{P}$$

Přenositelnost zdroje X daným kanálem popisuje **vzájemná informace**

$$I(X;Y) = H(X) - H(X|Y)$$

20.7 Shannonova věta o kódování

20.7.1 Shannonova věta o kódování za přítomnosti šumu

Tato věta říká, že existuje kódování opravující chyby, které dovoluje přenést informaci reálným kanálem (kanál s přítomností šumu) rychlostí libovolně blízkou kapacitě kanálu.

20.7.2 Shannonova věta o kódování bez přítomnosti šumu

Mějme $f : W \to A*$ kód se zdrojovými slovy $w_1, w_2, \ldots, w_m$, délek l_i, s pravděpodobností p_i vyslání slova w_i a kódovými slovy $a_i = f(w_i)$. Pak **průměrná délka kódového slova** je:

$$f = \sum_{i=1}^{m} p_i l_i.$$

21 Deterministický konečný automat, jazyk přijímaný konečným automatem. (A4B01JAG)

21.1 Jazyky - úvod

21.1.1 Abeceda

Konečnou neprázdnou množinu Σ budeme nazývat *abecedou*. Prvky množiny Σ nazýváme symboly, písmeny apod.

21.1.2 Slovo nad abecedou

Pro danou abecedu Σ *slovo nad* Σ je libovolná konečná posloupnost prvků abecedy Σ. Tedy např. pro $\Sigma = \{a, b\}$ jsou *aab*, *b*, *bbaba* slova nad Σ.

Prázdné slovo, značíme je ϵ, je posloupnost, která neobsahuje ani jeden symbol.

21.1.3 Délka slova

Je dáno slovo nad abecedou Σ. *Délka slova* je rovna délce posloupnosti, tj. počtu symbolů, které se ve slově nacházejí. Délku slova u značíme $|u|$.

Tedy, délka slova *aab* je rovna 3, délka *b* je 1, délka prázdného slova ϵ je 0.

21.1.4 Zřetězení slov

Je dána abeceda Σ. Pro dvě slova u, v nad abecedou Σ definujeme operaci *zřetězení* takto: Je-li $u = a_1a_2 \dots a_n$ a $v = b_1b_2 \dots b_k$, pak

$$u \cdot v = a_1a_2 \dots a_nb_1b_2 \dots b_k.$$

Často znak pro operaci zřetězení vynecháváme; píšeme tedy uv místo přesnějšího $u \cdot v$.

Zřetězení slov je asociativní operace na množině všech slov nad danou abecedou, prázdné slovo ϵ je neutrální prvek této operace.

$\Sigma^\star$, Σ^+. Označíme $\Sigma^\star$ množinu všech slov nad abecedou Σ. (Tj. prázdné slovo patří do $\Sigma^\star$) Pak $\Sigma^\star$ spolu s operací zřetězení tvoří monoid, jehož neutrálním prvkem je prázdné slovo ϵ.

Označíme Σ^+množinu všch neprázdných slov nad abecedou Σ. (Tj. $\Sigma^+ = \Sigma^\star \setminus \{\epsilon\}$.) Pak $\Sigma^\star$ spolu s operací zřetězení tvoří pologrupu.

Zřetězení slov není komutativní. Např. pro $u = aab$ a $v = b$ je $uv = aabb$, ale $vu = baab$.

Pro libovolná slova u a v nad stejnou abecedou platí:

$$|uv| = |u| + |v|\,.$$

Je-li slovo nad abecedou Σ, pak

$$u^0 = \epsilon,$$

$$u^{i+1} = uu^i \text{ pro každé } i.$$

21.1.5 Podslovo

Je dáno slovo u. Řekneme, že slovo w je podslovem slova u, jestliže existují slova x, y taková, že

$$u = xwy.$$

21.1.6 Prefix slova

Je dáno slovo u. Řekneme, že slovo w je prefix slova u, jestliže existuje slovo y takové, že

$$u = wy.$$

21.1.7 Jazyk nad abecedou

Je dána abeceda Σ. *Jazyk* L nad abecedou Σ je libovolná množina slov, tj. $L \subseteq \Sigma^\star$.

Je-li Σ abeceda, pak množina všech slov $\Sigma^\star$ je spočetná. Jazyků, jako podmnožin spočetné množiny, je víc - nespočetně mnoho.

21.2 Deterministické konečné automaty

21.2.1 Použití

Konečné automaty se používají v různých oborech. Jako příklady můžeme uvést překladače, dále se používají při zpracování přirozeného jazyka, pří návrzích hardwaru.

Zhruba řečeno konečný automat obsahuje konečnou množinu stavů Q, konečnou množinu vstupů Σ, přechodovou funkci δ a počáteční stav q_0. V některých případech ještě i množinu výstupních symbolů Y a výstupních funkcí.

21.2.2 Příklad 1

Uvažujme zjednodušený příklad automatu na kávu. Automat přijímá mince 1 Kč, 2 Kč a 5 Kč. Automat vydává jediný druh kávy, káva stojí 7 Kč. Automat na tlačítko s vrátí nevyužité peníze. Tento příkad uvedeme podrobněji.

Zde $Q = \{0,1,2,3,4,5,6\}$, $\Sigma = \{1,2,5,s\}$, $Y = \{K,0,1,2,3,4,5,6\}$, přechodová a výstupní funkce jsou dány následující tabulkou:

V prvním sloupci jsou stavy, ve kterých se automat může nacházet, v prvním řádku jsou vstupní symboly. V řádku odpovídajícím stavu q a sloupci se vstupen a je dvojice (nový stav, výstup). (K znamená kávu, číslo udává vrácené peníze).

	1	2	5	s
0	1/0	2/0	5/0	0/0
1	2/0	3/0	6/0	0/1
2	3/0	4/0	0/K	0/2
3	4/0	5/0	1/K	0/3
4	5/0	6/0	2/K	0/4
5	6/0	0/K	3/K	0/5
6	0/K	1/K	4/K	0/6

21.2.3 Obecně rozlišujeme čtuři typy automatů

Mealyho automat, Mooreův automat, akceptor a automat bez výstupu. Dále se budeme zabývat hlavně tzv. akceptory.

21.2.4 Mealyho automat

Mealyho automat je šestice $(Q, \Sigma, Y, \delta, q_0, \lambda)$. kde Q, Σ, Y a q_0 mají stejný význam jako v 21.2.1, přechodová funkce je zobrazení δ: $Q \times \Sigma \to Q$ a výstupní funkce je zobrazení λ: $Q \times \Sigma \to Y$.

21.2.5 Moorův automat

Moorův automat je šestice $(Q, \Sigma, Y, \delta, q_0, \beta)$. kde Q, Σ, Y, δ a q_0 mají stejný význam jako v 21.2.4, β je zobrazení β: $Q \to Y$ (říká se mu značkovací funkce).

21.2.6 Akceptor, též DFA

DFA je pětice $(Q, \Sigma, \delta, q_0, F)$, kde Q, Σ, δ a q_0 mají stejný význam jako v 21.2.5 a $F \subseteq Q$ je množina koncových (též přijímajících) stavů.

Jedná se vlastně o Mooreův automat, kde množina výstupních symbolů má dva prvky, totiž $Y = \{0, 1\}$, a proto značkovací funkci β nahrazujeme množinou těch stavů, kterým značkovací funkce přiřazuje 1.

21.2.7 Automat bez výstupu

Automat bez výstupu je „společnou částí" všech výše uvedených automatů; tj. jedná se o (Q, Σ, δ, q_0).

21.2.8 Stavový diagram

Kromě tabulky, můžeme konečný automat zadat též stavovým diagramem.

Je dán konečný automat s množinou stavů Q, množinou vstupních symbolů Σ, přechodovou funkcí δ. *Stavovým diagramem* nazýváme orientovaný ohodnocený graf, jehož vrcholy jsou stavy (tj. $V = Q$) a orientovaná hrana vede z vrcholu q do vrcholu p právě tehdy, když $\delta(q,\ a) = p$; v tomto případě je hrana ohodnocena vstupním symbolem a pro Moorův automat a DFA, nebo dvojicí $a/\lambda(q,\ a)$ v případě, že se jedná o Mealyho automat.

Jesliže se jedná o Mooreův automat, vrcholy stavového diagramu jsou navíc ohodnoceny značkovací funkcí β. Pro akceptor, tj DFA, označujeme pouze množinu koncových stavů, a to buď šipkou mířící ze stavu ven nebo jiným označením stavů, které patří do množiny F. Počáteční stav q_0 je označován šipkou mířící do něj.

21.2.9 Rozšířená přechodová funkce

Je dán automat (Q, Σ, δ). Rozšířená přechodová funkce $\delta^\star$: $Q \times \Sigma^\star \to Q$ je definovaná induktivně takto:

1. $\delta^\star(q, \epsilon) = q$, pro všechna $q \in Q$,
2. $\delta^\star(q, ua) = \delta(\delta^\star(q, u), a)$, pro všechna $q \in Q$, $a \in \Sigma$, $u \in \Sigma^\star$.

21.2.10 Jazyk přijímaný konečným automatem

Je dán DFA $M = (Q, \Sigma, \delta, q_0, F)$. Řekneme, že slovo $u \in \Sigma^\star$ je *přijímáno* automatem M právě tehdy, když

$$\delta^\star(q_0, u) \in F.$$

Množina všech slov, které automat přijímá, se nazývá *jazyk přijímaný M*, značíme ji *L(M)*. Tedy,

$$L(M) = \{\omega | \delta^\star(q_0, \omega) \in F\}.$$

21.2.11 Regulární jazyky

Každý jazyk, který je přijímán některým DFA, nazveme *regulární jazyk*. Třídu všech regulárních jazyků označujeme **Reg**.

21.2.12 Pumping lemma pro regulární jazyky

Pro každý regulární jazyk L nad abecedou Σ (tj. jazyk, který je přijímán nějakým DFA) existuje přirozené číslo n s touto vlastností:

Jestliže nějaké slovo $u \in L$ je delší než n (tj. $|u| > n$), pak u lze rozdělit na tři slova $u = xwy$ tak, že

1. $w \neq \epsilon$,
2. $|xw| \leq n$,
3. $xw^i y \in L$ pro každé přirozené číslo $i = 0, 1, \ldots$.

Důkaz: Předpokládejme, že jazyk L je regulární. Tedy existuje DFA M, který tento jazyk přijímá. Označme n počet jeho stavů. Vezměme libovolné slovo $u \in L$ délky větší než n. Sled ve stavovém diagramu, který odpovídá práci automatu nad slovem u, musí obsahovat cyklus (má větší délku než je počet vrcholů – stavů). Označme x slovo, které odpovídá té části sledu než poprvé vstoupíme do cyklu, w slovo, které odpovídá jednomu průchodu tímto cyklem, a y slovo odpovídající zbylé části sledu.

Není těžké se přesvědčit, že slova x, w, y splňují vlastnosti z pumping lemmatu.

Využití pumping lemmatu: Jazyk $L = \{0^m 1^m | m \geq 0\}$ není regulární jazyk.

Kdyby L byl regulární jazyk, muselo by existovat přirozené číslo n s vlastností z 21.2.12. Položme $u = 0^n 1^n$. Zřejmě $u \in L$ a $|u| = 2n > n$. Tedy $u = xwy$, $w \neq \epsilon$, $|xw| \leq n$ a $xw^2 y \in L$. To ale není možné; slovo w by muselo obsahovat jen 0, protože délka slova xw je menší nebo rovna n a prefix slova u délky n je 0^n. Navíc slovo w není prázdné, a tedy $w = 0^k$ pro $0 < k \leq n$. Pak ale slovo $xw^2 y$ je rovno $0^{n+k}1^n$ a nemá stejný počet 0 i 1, tj. neleží v jazyce L. Spor.

21.2.13 Ekvivalentní automaty

Řekneme, že dva automaty M_1 a M_2 jsou *ekvivalnetní*, jestliže přijímají stejný jazyk, tj. jestliže $L(M_1) = L(M_2)$.

21.2.14 Dosažitelné stavy

Je dán DFA $M = (Q, \Sigma, \delta, q_0, F)$. Řekneme, že stav $q \in Q$ je *dosažitelný*, jestliže existuje slovo $u \in \Sigma^\star$ takové, že $\delta^\star(q_0, u) = q$. Jinými slovy, stav q je dosažitelný, jestliže je dosažitelný z počátečního stavu q_0 ve stavovém diagramu M (tj. z q_0 vede do q orientovaný sled).

Je zřejmé, že stavy, které jsou nedosažitelné, nemají vliv na jazyk, který daný automat přijímá.

21.2.15 Ekvivalence stavů $\sim$

Máme DFA M $= (Q, \Sigma, \delta, q_0, F)$. Řekneme, že dva stavy p, $q \in Q$ jsou *ekvivalentní*, jestliže pro každé slovo $u \in \Sigma^\star$ platí

$$\delta^\star(p,u) \in F \text{ iff } \delta^\star(q,u) \in F.$$

Fakt, že dva stavy p a q jsou ekvivalentní, zapisujeme $p \sim q$.

21.2.16 Redukovaný automat

Je dán DFA $M = (Q, \Sigma, \delta, q_0, F)$. Řekneme, že M je *redukovaný*, jestliže nemá nedosažitelné stavy a žádné jeho dva různé stavy nejsou ekvivalentní. (To znamená, že ekvivalence $\sim$ je identická ekvivalence.)

21.2.17 Konstrukce relace $\sim$

Konstruujeme relace $\sim_i$, $i = 0, 1, \ldots$, na množině všech stavů Q takto:

- $p \sim_0 q$ právě tehdy, když buď $p,q \in F$ nebo $p,q \notin F$;
- je-li $i \geq 0$, pak $p \sim_{i+1} q$ právě tehdy, když $p \sim_i q$ a pro každé $a \in \Sigma$ máme $\delta(p, a) \sim_i \delta(q, a)$.

Věta: Platí

$$\sim_0 \supseteq \sim_1 \supseteq \ldots \supseteq \sim_i \supseteq \ldots$$

Existuje k takové, že $\sim_k$ je rovna $\sim_{k+1}$. Pak pro každé $j \geq 1$ platí $\sim_k = \sim_{k+j}$ a tedy $\sim_k = \sim$.

21.2.18 Algoritmus redukce

Je dán DFA $M = (Q, \Sigma, \delta, q_0, F)$.

1. Zkonstruujeme množinu Q' všech dosažitelných stavů automatu M. Postupujeme např. hledáním do šířky ze stavu q_0 ve stavovém diagramu.
2. Podle předchozího odstavce zkonstruujeme ekvivalenci $\sim$ pro DFA $M' = (Q, \Sigma, \delta, q_0, F \cap Q')$.
3. Vytvoříme DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, kde $Q_1 = Q'/\sim = \{[q]_\sim \,|q \in Q'\}$, $q_1 = [q_0]_\sim$, $\delta_1([q]_\sim, a) = [\delta(q,a)]_\sim$ a $F_1 = \{[q]_\sim \,|q \in F \cap Q'\}$.

Automat M_1 vznikl takto: za stavy jsme vzali třídy ekvivalence $\sim$, počáteční stav je třída, ve které leží původní počáteční stav q_0, přechodová funkce „pracuje" na třídách (což je možné vzhledem k vlastnosti $\sim$) a množina koncových stavů je množina těch tříd, ve kterých leží koncové stavy automatu M'.

21.2.19 Příklad

Je dán DFA M následující tabulkou:

	a	b
1	2	3
2	2	4
3	3	5
4	2	7
5	6	3
6	6	6
7	7	4
8	2	3
9	9	4

Nalezněte redukovaný automat k DFA M.

Řešení: Nejprve najdeme všechny dosažitelné stavy automatu M. Jsou to stavy {1,2,3,4,5,6,7}. Tedy $Q' = \{1,2,3,4,5,6,7\}$, $F' = F = \{3,5,6\}$.

Automat M' je dán tabulkou:

	a	b
1	2	3
2	2	4
3	3	5
4	2	7
5	6	3
6	6	6
7	7	4

Vytvoříme rozklad R_0 ekvivalence $\sim_0$:

$$O = \{1, 2, 4, 7\} \quad F = \{3, 5, 6\}.$$

Platí

$\delta(1, a) = 2 \in O$, $\delta(2, a) = 2 \in O$, $\delta(4, a) = 2 \in O$, $\delta(7, a) = 7 \in O$, $\delta(1, b) = 3 \in F$, $\delta(2, b) = 4 \in O$

Tedy musíme množinu O rozdělit na dvě podmnožiny, a to $\{1\}$ a $\{2, 4, 7\}$. Dále

$\delta(3, a) = 3 \in F$, $\delta(5, a) = 6 \in F$, $\delta(6, a) = 6 \in F$, $\delta(3, b) = 5 \in F$, $\delta(5, b) = 3 \in F$, $\delta(6, b) = 6 \in F$.

Proto množinu F nedělíme.

Rozklad odpovídající ekvivalenci $\sim_1$ je

$$A = \{1\}, \; O = \{2, 4, 7\}, \; F = \{3, 5, 6\}.$$

Výpočet zahrneme do tabulky

	a	b	$\sim_0$	a	b	$\sim_1$
1	2	3	O	O	F	A
2	2	4	O	O	O	O
3	3	5	F	F	F	F
4	2	7	O	O	O	O
5	6	3	F	F	F	F
6	6	6	F	F	F	F
7	7	4	O	O	O	O

Analogicky postupujeme k vytvoření ekvivalence $\sim_1$. Výpočet již zkrátíme jen do tabulky.

	a	b	$\sim_0$	a	b	$\sim_1$	a	b	$\sim_2$
1	2	3	O	O	F	A	O	F	A
2	2	4	O	O	O	O	O	O	O
3	3	5	F	F	F	F	F	F	F
4	2	7	O	O	O	O	O	O	O
5	6	3	F	F	F	F	F	F	F
6	6	6	F	F	F	F	F	F	F
7	7	4	O	O	O	O	O	O	O

Z tabulky vyplývá , že $\sim_1$=$\sim_2$. Proto $\sim_1$=$\sim$ je hledaná ekvivalence.

Máme tedy tři třídy ekvivalence, a to A, O a F. Redukovaný automat M_1 je dán tabulkou

	a	b
A	O	F
O	O	O
F	F	F

Není těžké nahlédnout, že automat M_1 přijímá jazyk $L = \{bu | u \in \{a, b\}^\star\}$.

Věta: Automat M i k němu redukovaný automat M_1 přijímají stejný jazyk, tj. jsou ekvivalentní.

Věta: Dva DFA M_1 a M_2 přijímají stejný jazyk (tj. jsou ekvivalentní) právě tehdy, když jejich odpovídající redukované automaty se liší pouze pojmenováním stavů.

21.2.20 Nerodova věta

Je dán jazyk L nad abecedou Σ. Pak L je regulární jazyk (tj. je přijímán nějakým DFA) právě tehdy, když existuje ekvivalence R na množině všech slov $\Sigma^\star$ taková, že

1. L je sjednocení některých tříd ekvivalence R.
2. R splňuje následující podmínku: Je-li uRv pro u, $v \in \Sigma^\star$, pak pro každé slovo $w \in \Sigma^\star$ platí $uwRvw$.
3. R má pouze konečně mnoho tříd ekvivalence.

Poznamenejme, že druhá podmínka vlastně říká, že ekvivalence R je pravá kongruence monoidu $(\Sigma^\star, \cdot, \epsilon)$.

Důkaz: Jestliže je jazyk L regulární, pak existuje DFA $M = (Q, \Sigma, \delta, q_0, F)$, takový, že $L = L(M)$. Definujme relaci R na $\Sigma^\star$ takto:

$$uRv \quad \text{iff} \quad \delta^\star(q_0, u) = \delta^\star(q_0, v).$$

Takto definovaná relace splňuje všechny podmínky Nerodovy věty.

Předpokládejme, že pro jazyk L existuje ekvivalence R splňující všechny podmínky z Nerodovy věty. Definujme DFA $M = (Q, \Sigma, \delta, q_0, F)$ takto:

$$Q = \{[u]_R \,|\, u \in \Sigma^\star\}, \quad q_0 = [\epsilon]_R, \quad F = \{[u]_R \,|\, [u]_R \subseteq L\};$$

$$\delta([u]_R, a) = [ua]_R \qquad \text{pro každé } a \in \Sigma.$$

Pak DFA M přijímá jazyk L.

Poznámka: Podobně jako pumping lemma i Nerodova věta se dá použít k tomu, abychom ukázali, že některý jazyk není regulární. Navíc je ale možné Nerodovu větu použít i pro konstrukci automatu, který daný jazyk přijímá.

22 Regulární výrazy a regulární jazyky, Kleeneova věta. Algoritmická složitost úloh souvisejících s regulárními jazyky. (A4B01JAG)

22.1 Regulární jazyky

Regulární jazyky viz kapitola 21 Deterministický konečný automat, jazyk přijímaný konečným automatem.

22.1.1 Uzávěrkové vlastnosti třídy regulárních jazyků

Třída regulárních jazyků je uzavřena na sjednocení, průnik, doplněk i rozdíl.

Přesněji, jestliže L_1 a L_2 jsou regulární jazyky, pak také $L_1 \cup L_2$, $L_1 \cap L_2$, $\bar{L_1} = \Sigma^\star \setminus L_1$ a $L_1 \setminus L_2$ jsou také regulární jazyky.

22.1.1.1 Zřetězení jazyků

Jsou dány jazyky L_1 a L_2 nad abecedou Σ. *Zřetězení* jazyků L_1 a L_2 je jazyk L_1L_2 definovaný

$$L_1L_2 = \{uv | u \in L_1, v \in L_2\}.$$

Tvrzení: Třída regulárních jazyků je uzavřena na zřetězení. Přesněji, jsou-li jazyky L_1 a L_2 regulární, je regulární i jazyk L_1L_2.

22.1.1.2 Operace $\star$

Je dán jazyk L nad abecedou Σ. Definujeme $L_0 = \{\epsilon\}$, $L^{i+1} = L^iL$ pro $i \geq 0$. Pak operace $\star$ pro jazyk L ($L^\star$) je definována

$$L^\star = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots = \bigcup_{i=0}^{\infty} L^i.$$

Poznamenejme, že operaci $\star$ se též říká *Kleeneho operátor*.

Tvrzení: Třída regulárních jazyků je uzavřena na operaci $\star$. Přesněji, je-li jazyk L regulární, je regulární i jazyk $L^\star$.

22.2 Regulární výrazy

Regulární výrazy slouží k ještě jinému popisu regulárních jazyků. Právě regulární výrazy daly jméno třídě jazyků přijímaných konečnými automaty (ať už deterministickými nebo nedeterministickými).

22.2.0.3 Regulární výrazy nad abecedou

Je dána abeceda Σ. Množina všech regulárních výrazů nad Σ je definována:

- $\emptyset$ je regulární výraz,
- ϵ je regulární výraz,
- $\mathbf{a}$ je regulární výraz pro každé písmeno $a \in \Sigma$,
- jsou-li $\mathbf{r_1}$ a $\mathbf{r_2}$, pak $\mathbf{r_1} + \mathbf{r_2}$, $\mathbf{r_1 r_2}$ a $\mathbf{r_1^\star}$ jsou regulární výrazy.

22.2.0.4 Jazyk odpovídající regulárnímu výrazu

Každému regulárnímu výrazu nad abecedou Σ odpovídá jazyk nad abecedou Σ a to takto:

- Regulárnímu výrazu $\emptyset$ odpovídá jazyk $\emptyset$.
- Regulárnímu výrazu ϵ odpovídá jazyk $\{\epsilon\}$.
- Je-li $a \in \Sigma$, pak regulárnímu výrazu $\mathbf{a}$ odpovídá jazyk $\{a\}$.
- Jestliže regulárnímu výrazu $\mathbf{r_1}$ odpovídá jazyk L_1 a regulárnímu výrazu $\mathbf{r_2}$ odpovídá jazyk L_2, pak regulárnímu výrazu $\mathbf{r_1} + \mathbf{r_2}$ odpovídá jazyk $L_1 \cup L_2$ a regulárnímu výrazu $\mathbf{r_1 r_2}$ odpovídá jazyk $L_1 L_2$.
- Jestliže regulárnímu výrazu $\mathbf{r}$ odpovídá jazyk L, pak regulárnímu výrazu $\mathbf{r}^\star$ odpovídá jazyk $L^\star$.

Věta: Každý regulární výraz nad abecedou Σ odpovídá regulárnímu jazyku (nad abecedou Σ), tj. jazyku, který je přijímán konečným automatem.

Důkaz: Regulárním výrazům $\emptyset$, ϵ, $\mathbf{a}$ (pro $a \in \Sigma$) odpovídají po řadě jazyky $\emptyset$, $\{\epsilon\}$, $\{a\}$. Všechny tyto jazyky jsou přijímány konečným automatem.

O třídě jazyků přijímaných konečnými automaty víme, že je uzavřena na sjednocení, zřetězení a Kleeneho operaci $\star$. To znamená, že jsou-li jazyky odpovídající regulárním výrazům $\mathbf{r}$, $\mathbf{r_1}$ a $\mathbf{r_2}$ přijímány konečnými automaty, pak totéž platí i pro jazyky odpovídající regulárním výrazům $\mathbf{r_1} + \mathbf{r_2}$, $\mathbf{r_1 r_2}$ a $\mathbf{r}^\star$.

22.2.0.5 Kleeneho věta

Každý jazyk přijímaný konečným automatem je možné popsat regulárním výrazem.

Důkaz: Je dán DFA $M = (Q, \Sigma, \delta, q_0, F)$, který přijímá jazyk L. Pro jednoduchost označme množinu stavů $Q = \{1, \ldots, n\}$ a počátěčni stav $q_0 = 1$. Pro $k = 0, 1, \ldots, n$ definujeme množiny slov $R_{i,j}^{(k)}$ takto

$R_{i,j}^{(k)}$ je množina těch slov w, které $\delta^\star(i, w) = j$ a sled z i do j prochází pouze přes stavy $1, \ldots, k$.

Platí $R_{i,j}^{(0)} = \{a \in \Sigma | \delta(i, a) = j\}$, což je konečná množina písmen. Proto umíme množinu $R_{i,j}^{(0)}$ popsat regulárním výrazem.

Jestliže všechny množiny slov $R_{i,j}^{(k)}$ umíme popsat regulárním výrazem $\mathbf{r}_{i,j}^k$, pak pro množinu slov $R_{i,j}^{(k+1)}$ platí

$$R_{i,j}^{(k+1)} = R_{i,j}^{(k)} \cup R_{i,k+1}^{(k)} \left(R_{k+1,k+1}^{(k)}\right)^\star R_{k+1,j}^{(k)}.$$

Tedy $R_{i,j}^{(k+1)}$ popíšeme regulárním výrazem $\mathbf{r}_{i,j}^k + \mathbf{r}_{i,k+1}^k \left(\mathbf{r}_{k+1,k+1}^k\right)^\star \mathbf{r}_{k+1,j}^k$, což je opět regulární výraz.

Navíc jazyk L je sjednocení všech množin $R_{1,j}^{(n)}$ pro $j \in F$. Proto jazyku L odpovídá regulární výraz $\Sigma_{j \in F} \mathbf{r}_{1,j}^n$.

22.2.0.6 Aplikace regulárních výrazů

1. Program *grep* (Global search for Regular Expression and Print).
2. Využití v editorech.
3. Využití v programovacích jazycích.
4. Využití při syntaktické analýze v překladačích.

Poznámka: Zavedli jsme regulární výrazy tak, jak jsou definovány v teorii konečných automatů. Při praktickém použití regulárních výrazů v computer science se používá jiné značení, a navíc se zavádí rozšířené regulární výrazy, které pak už nepopisují jen regulární jazyky. Víc o těchto regulárních výrazech najdete na webové stránce Pavla Satrapy *http://www.nti.tul.cz/ satrapa/docs/regvyr/*.

22.2.0.7 Některé rovnosti mezi regulárními výrazy

Jsou-li $\mathbf{r}$, $\mathbf{p}$ a $\mathbf{q}$ regulární výrazy, pak platí následující rovnosti (to znamená: regulární výraz odpovídající levé straně a regulární výraz odpovídající pravé straně popisují stejný jazyk):

1. $\mathbf{p} + \mathbf{q} = \mathbf{p} + \mathbf{q}$,
2. $\mathbf{r}\ (\mathbf{p} + \mathbf{q}) = \mathbf{r}\ \mathbf{p} + \mathbf{r}\ \mathbf{q}$,

3. $(\mathbf{p} + \mathbf{q})\ \mathbf{r} = \mathbf{p}\ \mathbf{r} + \mathbf{q}\ \mathbf{r}$,
4. $(\mathbf{r}^\star)^\star = \mathbf{r}^\star$,
5. $(\mathbf{p} + \mathbf{q})^\star = (\mathbf{p}^\star \mathbf{q}^\star)^\star$,
6. $(\mathbf{p} + \mathbf{q})^\star = (\mathbf{p}^\star + \mathbf{q})^\star$,
7. $(\mathbf{p} + \mathbf{q})^\star = (\mathbf{p}^\star \mathbf{q})^\star\, \mathbf{p}^\star$,
8. $\mathbf{r}^\star = \epsilon + \mathbf{r}\mathbf{r}^\star$,
9. $\mathbf{r}\mathbf{r}^\star = \mathbf{r}^\star\mathbf{r}$,
10. $(\mathbf{p}\mathbf{q})^\star = \epsilon + \mathbf{p}\,(\mathbf{q}\mathbf{p})^\star\,\mathbf{q}$,
11. $(\mathbf{p}\mathbf{q})^\star\,\mathbf{p} = \mathbf{p}\,(\mathbf{q}\mathbf{p})^\star$.

22.3 Další uzávěrkové vlastnosti třídy regulárních jazyků

22.3.0.8 Homomorfismus

Jsou dány dvě abecedy Σ, Γ a zobrazení h, které každému písmenu $a \in \Sigma$ přiřadí slovo $h(a)$ nad abecedou Γ.

Zobrazení h rozšíříme na zobrazení, které každému slovu $u \in \Sigma^\star$ přiřazuje slovo nad Γ takto:

- $h(\epsilon) = \epsilon$,
- $h(ua) = h(u)\,h(a)$.

Obraz jazyka L nad Σ je definován $h(L) = \cup\{h(w)\,|w \in L\}$.

Takto definované zobrazení h se nazývá *homomorfismus*.

Příklad: $\Sigma = \{0,1\}$, $\Gamma = \{a,b\}$ a $h(0) = ab^2$, $h(1) = bab$. Pak $h(010) = ab^2babab^2 = ab^2\,(ba)^2\,b^2$. Homomorfní obraz jazyka $L = \left\{10^k | k \geq 0\right\}$ je $h(L) = \left\{bab\,(ab^2)^k\,|k \geq 0\right\}$.

22.3.0.9 Substituce

Obecnější pojem než homomorfismus je tzv. substituce. Jsou dány dvě abecedy Σ, Γ a zobrazení σ, které každému písmenu $a \in \Sigma$ přiřadí jazyk nad abecedou Γ.

Analogicky jako pro homomorfismus zobrazení σ rozšíříme na zobrazení, které každému slovu $u \in \Sigma^\star$ přiřazuje jazyk nad Γ takto:

- $\sigma(\epsilon) = \{\epsilon\}$,
- $\sigma(ua) = \sigma(u)\,\sigma(a)$.

Obraz jazyka L nad Σ je $\sigma(L) = \cup\{\sigma(w)\,|w \in L\}$.

Takto definované zobrazení σ se nazývá *substituce*.

Příklad: $\Sigma = \{0,1\}$, $\Gamma = \{a,b\}$, $\sigma(0) = L_1 = \{a^n | n \geq 0\}$, $\sigma(1) = L_2 = \{b^n | n \geq 0\}$. Pak $\sigma(01) = L_1L_2 = \{a^nb^m | n,m \geq 0\}$.

22.3.0.10 Věta

Třída regulárních jazyků je uzavřena na homomorfismy. Jinými slovy, jestliže L je regulární jazyk nad abecedou Σ a h je homomorfismus z Σ do Γ, pak $h(L)$ je regulární jazyk nad abecedou Γ.

Poznamenejme, že obdobná věta platí i pro substituce. Je-li L regulární jazyk nad abecedou Σ a σ je taková substituce z Σ do Γ, že každý z jazyků $\sigma(a)$ pro $a \in \Sigma$ je regulární jazyk nad Γ, pak jazyk $\sigma(L)$ je také regulární jazyk nad Γ.

22.3.0.11 Věta

Třída regulárních jazyků je uzavřena na inversní homomorfismy. Jinými slovy, jestliže h je homomorfismus a L je regulární jazyk nad abecedou Γ, pak jazyk $h^{-1}(L)$ je regulární jazyk nad abecedou Σ.

Připomeňme, že $h^{-1}(L) = \{u \in \Sigma^\star | h(u) \in L\}$.

Příklad: Uvažujme jazyk L nad abecedou $\Gamma = \{a, b\}$ popsaný regulárním výrazem $(\mathbf{00} + \mathbf{1})^\star$ a homomorfismus h, kde $h(a) = 01$ a $h(b) = 10$.

Pak $h^{-1}(L)$ je jazyk nad abecedou $\Sigma = \{a, b\}$ popsaný regulárním výrazem $(\mathbf{ba})^\star$.

22.3.0.12 Reverse

Je dán jazyk L nad abecedou Σ. Pak jazyk L^R definovaný

$$L^R = \left\{ w^R | w \in L \right\}.$$

se nazývá *reverse* jazyka L.

Věta: Třída regulárních jazyků je uzavřena na reverse, přesněji: jestliže L je regulární jazyk nad abecedou Σ, pak je regulární i jazyk L^R.

22.3.0.13 Levý kvocient

Máme dva jazyky L a L_1 nad abecedou Σ. Pak *levý kvocient* je jazyk

$$L_1 \setminus L = \{v | \exists u \in L_1, uv \in L\}.$$

Příklad: Uvažujme jazyky L_1 a L_2 nad abecedou $\Sigma = \{0, 1\}$, kde $L_1 = \left\{ 0^k 10^n | k, n \geq 0 \right\}$, $L_2 = \{10^m 1 | m \geq 0\}$.

Pak $L_2 \setminus L_1 = \emptyset$ a $L_1 \setminus L_2 = \{0^q 1 | q \geq 0\}$.

Věta: Třída regulárních jazyků je uzavřena na levé kvocienty. Přesněji, jestliže L a L_1 jsou regulární jazyky, pak i $L_1 \setminus L$ je regulární jazyk.

22.3.0.14 Pravý kvocient

Máme dva jazyky L a L_2 nad abecedou Σ. Pak *pravý kvocient* je jazyk

$$L/L_2 = \{v | \exists u \in L_2, vu \in L\}.$$

Příklad: Uvažujme jazyky L_1 a L_2 nad abecedou $\Sigma = \{0, 1\}$, kde $L_1 = \left\{0^k 10^n | k, n \geq 0\right\}$, $L_2 = \{10^m 1 | m \geq 0\}$.

Pak $L_2/L_1 = \left\{10^k | k \geq 0\right\}$ a $L_1/L_2 = \emptyset$.

Věta: Třída regulárních jazyků je uzavřena na pravé kvocienty. Přesněji, jestliže L a L_2 jsou regulární jazyky, pak i L/L_2 je regulární jazyk.

22.4 Algoritmická řešitelnost úloh pro regulární jazyky

Pro následující otázky týkající se konečných automatů a jimi přijímaných jazyků existují algoritmy, které dají správnou odpověď.

1. Pro daný konečný automat M (ať deterministický nebo nedeterministický) a slovo $w \in \Sigma^\star$ rozhodnout, zda $w \in L(M)$.
2. Pro daný konečný automat M (ať deterministický nebo nedeterministický) rozhodnout, zda $L(M) \neq \emptyset$.
3. Pro daný konečný automat M rozhodnout, zda $L(M) = \Sigma^\star$.
4. Pro dva konečné automaty M_1 a M_2 rozhodnout, zda $L(M_1) = L(M_2)$.

Tvrzení: Je dán deterministický konečný automat M s n stavy. Pak

1. Jazyk $L(M)$ je neprázdný právě tehdy, když M přijímá slovo w délky $|w| < n$.
2. Jazyk $L(M)$ je nekonečný právě tehdy, když M přijímá slovo v délky $n \leq |v| < 2n$.

23 Gramatiky, regulární gramatiky a bezkontextové gramatiky, bezkontextové jazyky. Zásobníkové automaty a jejich vztah k bezkontextovým jazykům. Vlastnosti bezkontextových gramatik, lemma o vkládání. (A4B01JAG)

23.1 Gramatiky

23.1.1 Hierarchie gramatik

23.1.1.1 Definice

Gramatika je uspořádaná čtveřice $G = (N, \Sigma, S, P)$, kde

- N je konečná množina tzv. *neterminálů;*
- Σ je konečná neprázdná množina tzv. *terminálů*, platí $N \cap \Sigma = \emptyset$;
- $S \in N$ je startovací symbol;
- P je konečná množina pravidel typu $\alpha \to \beta$, kde α a β jsou slova nad $N \cup \Sigma$ taková, že α obsahuje alespoň jeden neterminál.

23.1.1.2 Příklad

V programovacích jazycích se často vyskytují definice typu číslo v Backus-Naurově formě:

- $\langle$číslo$\rangle$ ::= $\langle$číso bez zn.$\rangle$ | + $\langle$číslo bez zn.$\rangle$ | − $\langle$číslo bez zn.$\rangle$
- $\langle$číslo bez zn.$\rangle$ ::= $\langle$číslice$\rangle$ | $\langle$číslice$\rangle$ $\langle$číslo bez zn.$\rangle$
- $\langle$číslice$\rangle$ ::= 0|1|2|3|4|5|6|7|8|9

Jedná se o speciální příklad gramatiky: Označíme S = $\langle$číslo$\rangle$, A = $\langle$číslo bez zn.$\rangle$ a B = $\langle$číslice$\rangle$. Pak se jedná o gramatiku, kde

$$N = \{S, A, B\}, \quad \Sigma = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

a pravidla P jsou

- $S \to A$, $S \to +A$, $S \to -A$;
- $A \to B$, $A \to BA$;
- $B \to 0$, $B \to 1, \ldots$, $B \to 9$.

23.1.1.3 Přímé odvození

Je dána gramatika $G = (N, \Sigma, S, P)$. Řekneme, že δ se *přímo odvodí* z γ, značíme $\gamma \Rightarrow_G \delta$, jestliže existuje v P pravidlo $\alpha \to \beta$ a slova $\varphi, \psi \in (N \cup \Sigma^\star)$ taková, že $\gamma = \varphi\alpha\psi$ a $\delta = \varphi\beta\psi$.

Zhruba řečeno, ve slově γ najdeme některý výskyt podslova α, které tvoří levou stranu pravidla $\alpha \to \beta$ z P . Slovo δ dostaneme tak, že zvolený výskyt α (v γ) nahradíme slovem β (tj. pravou stranou pravidla).

23.1.1.4 Odvození

Je dána gramatika $G = (N, \Sigma, S, P)$. Řekneme, že δ se *odvodí* z γ, jestliže

- buď $\gamma = \delta$,
- nebo existuje posloupnost přímých odvození
- $$\gamma \Rightarrow \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G \ldots \Rightarrow_G \gamma_k = \delta.$$
- Tento fakt značíme $\gamma \Rightarrow_G^\star \delta$.

23.1.1.5 Jazyk generovaný gramatikou

Řekneme, že slovo $w \in \Sigma^\star$ je *generováno* gramatikou G, jestliže $S \Rightarrow_G^\star w$.

Jazyk $L(G)$ *generovaný* gramatikou G se skládá ze všech slov generovaných gramatikou G, tj.

$$L(G) = \{w \in \Sigma^\star | S \Rightarrow_G^\star w\}.$$

23.1.1.6 Konvence

- Neterminály značíme obvykle velkými písmeny $A, B, X, Y, \ldots$
- Terminály značíme obvykle malými písmeny ze začátku abecedy $a, b, c, d, \ldots$
- Slova z $(N \cup \Sigma)^\star$ obvykle značíme řeckými písmeny $\alpha, \beta, \ldots$
- Terminální slova, tj. slova z $\Sigma^\star$, značíme malými písmeny z konce abecedy $u, w, x, y, \ldots$

Obvykle v textu vynecháváme jméno gramatiky, je-li z kontextu jasné o jakou gramatiku se jedná. Píšeme proto $\Rightarrow$ a $\Rightarrow^\star$ místo $\Rightarrow_G$ a $\Rightarrow_G^\star$.

23.1.1.7 Chomského hierarchie

Podle podmínek, které klademe na pravidla dané gramatiky rozlišujeme gramatiky a jimi generované jazyky na:

- *Gramatiky typu 0* jsou gramatiky tak, jak jsme je zavedli v odstavci 23.1.1.1. Jazyky generované gramatikami typu 0 se nazývají *jazyky typu 0*.
- *Gramatiky typu 1* též *kontextové gramatiky* jsou takové gramatiky, kde každé pravidlo v P je tvaru

  $$\alpha A \beta \to \alpha \gamma \beta,$$

  kde $\alpha, \beta, \gamma \in (N \cup \Sigma)^\star$, A je neterminál a $\gamma \neq \epsilon$. Jedinou výjimku tvoří pravidlo $S \to \epsilon$, pak se S nevyskytuje na pravé straně žádného pravidla.

  Jazyky generované gramatikami typu 1 se nazývají *jazyky typu 1*, též *kontextové jazyky*.
- *Gramatiky typu 2* též *bezkontextové gramatiky* (což zkracujeme na CFG) jsou takové gramatiky, kde každé pravidlo v P je tvaru

  $$A \to \gamma,$$

  kde $\gamma \in (N \cup \Sigma)^\star$ a A je neterminál.

  Jazyky generované gramatikami typu 2 se nazývají *bezkontextové jazyky* nebo j*azyky typu 2*.

  Příklad

  S $\to$ aSb

  S $\to$ ab

  Tato gramatika generuje jazyk $L = \{a^n b^n | n > 1\}$

- *Gramatiky typu 3* neboli *regulární gramatiky* (též *pravé lineární gramatiky*) jsou takové gramatiky, kde každé pravidlo v P je tvaru

$$A \to wB,\ A \to w,$$

kde A, B jsou neterminály a w je terminální slovo.

Jazyky generované gramatikami typu 3 se nazývají *regulární jazyky* nebo *jazyky typu 3*.

Příklad

S → aA|bB|ε

A → a|bB

B → a|b|aA

Poznamenejme, že regulární jazyky již byly definovány jako ty jazyky, které jsou přijímány konečnými automaty — později ukážeme, že je to správně, totiž, že každý jazyk typu 3 je přijímán konečným automatem.

23.1.1.8 Nevypouštěcí gramatiky

Gramatiku $G = (N, \Sigma, S, P)$ nazveme *nevypouštěcí*, jestliže neobsahuje žádné pravidlo typu $A \to \epsilon$.

Tvrzení: Ke každé bezkontextové gramatice G existuje nevypouštěcí gramatika G_1 taková, že

$$L(G) = L(G_1) - \{\epsilon\}.$$

Důsledek: Označme L_i třídu jazyků typu i. Pak platí:

$$L_3 \subseteq L_2 \subseteq L_1 \subseteq L_0.$$

23.2 Regulární jazyky a regulární gramatiky

23.2.1 Tvrzení

Ke každému regulárnímu jazyku L existuje regulární gramatika G, která ho generuje.

23.2.2 Lemma

Ke každé gramatice G typu 3 existuje gramatika G_1 typu 3 generující stejný jazyk a taková, že má pravidla pouze tvaru

$$A \to aB,\ A \to \epsilon.$$

Přidáním nových neterminálů je možné pravidlo $A \to wB$, kde $w = a_1 a_2 \ldots a_k$, nahradit posloupností pravidel $A \to a_1 X_1,\ X_1 \to a_2 X_2, \ldots, X_{k-1} \to a_k B$.

23.2.3 Tvrzení

Ke každé gramatice G typu 3 existuje konečný automat M takový, že

$$L(G) = L(M).$$

23.2.4 Věta

Gramatiky typu 3 generují právě třídu regulárních jazyků.

23.3 Bezkontextové gramatiky

Připomeňme, že bezkontextová gramatika (CFG) je gramatika $G = (N, \Sigma, S, P)$, která obsahuje pouze pravidla typu

$$A \to \gamma, \quad \text{kde } \gamma \in (N \cup \Sigma)^\star \text{ a } A \text{ je neterminál.}$$

Dále připomeňme, že ke každé CFG gramatice G existuje nevypouštěcí CFG gramatika G_1 taková, že

$$L(G_1) = L(G) - \{\epsilon\}.$$

23.3.1 Tvrzení

Máme dánu bezkontextovou gramatiku $G = (N, \Sigma, S, P)$ a v ní derivaci

$$S \Rightarrow_G^\star \alpha A \beta \Rightarrow_G^\star w,$$

pro $\alpha, \beta \in (\Sigma \cup N)^\star$, $A \in N$ a $w \in \Sigma^\star$.
Pak existují slova $u, x, v \in \Sigma^\star$ taková, že

$$w = uxv \quad \text{a} \quad \alpha \Rightarrow_G^\star u, \quad A \Rightarrow_G^\star x, \quad \beta \Rightarrow_G^\star v.$$

23.3.2 Redukovaná bezkontextová gramatika

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$, pro kterou $L(G) \neq \emptyset$. Řekneme, že G je *redukovaná*, jestliže splňuje tyto dvě podmínky:

1. Ke každému neterminálu A existuje aspoň jedno terminální slovo w takové, že $A \Rightarrow_G^\star w$.
2. Ke každému neterminálu A existují slova $\alpha, \beta \in (N \cup \Sigma)^\star$ tak, že $S \Rightarrow_G^\star \alpha A \beta$.

23.3.3 Tvrzení

Ke každé bezkontextové gramatice $G = (N, \Sigma, S, P)$, pro kterou $L(G) \neq \emptyset$, existuje redukovaná gramatika G_1 taková, že $L(G_1) = L(G)$.

23.3.4 Algoritmus redukce CFG

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$.

1. Sestrojíme množinu $V = \{A | A \in N, A \Rightarrow_G^\star w, w \in \Sigma^\star\}$:

$$V_0 = \Sigma,$$

$$V_{i+1} = V_i \cup \{A | \text{ existuje } \alpha \in V_i^\star \text{ takové, že } A \Rightarrow_G^\star \alpha\}.$$

   Platí

$$V_O \subseteq V_1 \subseteq V_2 \subseteq \ldots \subseteq (N \cup \Sigma).$$

   Proto existuje n takové, že $V_n = V_{n+1}$. Položíme $V = V_n - \Sigma$.

   Jestliže $S \notin V$, pak $L(G) = \emptyset$ a redukovaná gramatika ke gramatice G neexistuje.

   Definujeme $G' = (V, \Sigma, S, P')$: do P' dáme pouze ta pravidla z P, která obsahují neterminály z množiny V.

2. Pro gramatiku $G' = (V, \Sigma, S, P')$ zkonstruujeme množinu

$$U = \{A | A \in V, \text{ existují } \alpha, \beta \in (V \cup \Sigma)^\star \text{ tak, že } S \Rightarrow_G^\star \alpha A \beta\}.$$

$$U_0 = \{S\},$$

$$U_{i+1} = U_i \cup \{A | \text{existují } B \in U_i, \alpha, \beta \in (V \cup \Sigma)^\star \text{ tak, že } B \Rightarrow_G^\star \alpha A \beta\}.$$

   Platí

$$U_0 \subseteq U_1 \subseteq U_2 \subseteq \ldots \subseteq V.$$

   Proto existuje n takové, že $U_n = U_{n+1}$. Položíme $U = U_n$.

   Hledaná gramatika je gramatika $G_1 = (U, \Sigma, S, P_1)$, kde P_1 je množina všech pravidel z P' (a tedy i z P), které obsahují neterminály pouze z množiny U.

Platí: gramatika $G_1 = (U, \Sigma, S, P_1)$ je redukovaná a generuje stejný jazyk jako původní gramatika $G = (N, \Sigma, S, P)$.

23.3.5 Poznámky

- Uvědomte si, že redukovaná CFG gramatika „nemá zbytečné neterminály".
- Je obtížné ke dvěma bezkontextovým gramatikám zjistit, zda generují stejný jazyk. Redukce gramatik nám k rozhodnutí nepomůže.
- Kroky předchozího postupu nelze zaměnit. Kdybychom nejprve hledali množinu neterminálů U a pak teprve z ní vybírali ty neterminály, ze kterých je možné odvodit terminální slovo, výsledná gramatika by nemusela splňovat druhou podmínku z 23.3.2.

23.3.6 Levá derivace, levé odvození

Přímé odvození se nazývá *levé*, jestliže se přepisuje ten neterminál, který je nejvíc „vlevo", tj. $uA\beta \Rightarrow_G u\delta\beta$, kde $u \in \Sigma^\star$ a $A \to \delta$ je pravidlo gramatiky.

Derivace (odvození) se nazývá *levá*, jestliže se skládá pouze z levých přímých odvození.

Obdobně definujeme pravé přímé odvození a pravou derivaci.

23.3.7 Tvrzení

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$. Pak pro každou derivaci $S \Rightarrow_G^\star w$ existuje levá derivace terminálního w z S taková, že používá stejná pravidla jako původní derivace (pouze možná v jiném pořadí).

23.3.8 Derivační strom (parse tree)

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$. *Derivační strom* (anglicky *parse tree*) je kořenový strom, takový, že:

1. Každý vrchol, který není list, je ohodnocen neterminálem.
2. Každý list je ohodnocen terminálem nebo prázdným slovem ϵ. V případě, že je list ohodnocen prázdným slovem ϵ, jedná se o jediný následník (svého předchůdce).
3. Jestliže některý vrchol, který není list, je ohodnocen neterminálem A a má následníky (v pořadí od leva do prava) $X_1, X_2, \ldots, X_k$, $X_i \in N \cup \Sigma$, pak $A \to X_1 X_2 \ldots X_k$ je pravidlo gramatiky G.

Řekneme, že derivační strom *dává*, nebo *má za výsledek* slovo w, jestliže w je ohodnocení listů derivačního stromu (čteno od leva do prava).

23.3.9 Tvrzení

1. Pro každou derivaci $S \Rightarrow_G^\star w$ existuje derivační strom s výsledkem w.
2. Ke každému derivačnímu stromu s výsledkem w existuje aspoň jedna derivance $S \Rightarrow_G^\star w$ (takových derivací může být více).

3. Ke každému derivačnímu stromu s výsledkem w existuje právě jedna levá (právě jedna pravá) derivace w z S.

23.3.10 Jednoznačné a víceznačné bezkontextové gramatiky

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$. Řekneme, že G je *jednoznačná*, jestliže pro každé slovo w generované gramatikou G existuje jediný derivační strom s výsledkem w (tj. existuje jediná levá derivace w z S).

V opačném případě mluvíme o *víceznačné* gramatice.

23.3.11 Víceznačný jazyk

Bezkontextový jazyk L se nazývá *víceznačný* (též *podstatně víceznačný*), jestliže každá bezkontextová gramatika, která ho generuje, je víceznačná.

Například jazyk $L = \left\{a^i b^j c^k d^l | i = j \text{ nebo } k = l\right\}$ je podstatně víceznačný.

23.4 Zásobníkové automaty

Zhruba řečeno, zásobníkový automat se skládá z řídící jednotky, která je v jednom z možných stavů, ze vstupní pásky se čtecí hlavou a ze zásobníku. Na základě toho, v jakém stavu se automat nachází, co hlava čte na vstupní pásce a jaký symbol je na vrcholu zásobníku, automat udělá akci: přejde do nového stavu, posune čtecí hlavu o jedno políčko doprava nebo stojí (to v případě, že automat reagoval na prázdné slovo) a vrchol zásobníku nahradí zásobníkovým slovem.

23.4.1 Definice

Zásobníkový automat je sedmice $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde

- Q je konečná množina stavů,
- Σ je konečná množina vstupních symbolů,
- Γ je konečná množina zásobníkových symbolů,
- δ přiřazuje každé trojici (q, a, X), $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, konečnou množinu dvojic (p, α), kde $p \in Q$ a $\alpha \in \Gamma^\star$. Formálně:

  $$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to P_f\left(Q \times \Gamma^\star\right).$$

  ($P_f(A)$ značí množinu všech konečných podmnožin množiny A.)
- $q_0 \in Q$ je počáteční stav,
- $Z_0 \in \Gamma$ je počáteční zásobníkový symbol a
- $F \subseteq Q$ je množina koncových stavů.

Uvědomte si, že zásobníkový automat tak, jak byl definován, je nedeterministický.

23.4.2 Situace zásobníkového automatu

Je dán zásobníkový automat $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. *Situace* zásobníkového automatu je trojice (q, u, γ), kde q je stav, u je vstupní slovo a γ je zásobníkové slovo.

Znamená to, že zásobníkový automat je ve stavu q, na vstupní pásce má slovo u a v zásobníku slovo γ s tím, že první písmeno γ je na vrcholu zásobníku.

23.4.3 Jeden krok práce zásobníkového automatu - relace $\vdash_A$

Je dán zásobníkový automat $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, který je v situaci $(q, au, X\gamma)$, kde $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$. Pak A *přejde* do situace $(p, u, \alpha\gamma)$ pro $(p, \alpha) \in \delta(q, a, X)$. Značíme

$$(q, au, X\gamma) \vdash_A (p, u, \alpha\gamma) \quad \text{iff} \quad (p, \alpha) \in \delta(q, a, X).$$

23.4.4 Relace $\vdash_A^\star$

Jeden krok zásobníkového automatu rozšíříme na konečný počet. Automat A *přejde ze situace* S *do situace* S', píšeme $S \vdash^\star S'$, právě tehdy, když buď $S = S'$ nebo existuje konečný počet situací $S_1, S_2, \ldots, S_n$ takových, že

$$S \vdash_A S_1, S_1 \vdash_A S_2, \ldots, S_n \vdash_A S'.$$

23.4.5 Jazyk přijímaný prázdným zásobníkem

Je dán zásobníkový automat $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. *Jazyk přijímaný prázdným zásobníkem* $N(A)$ je definován takto:

$$N(A) = \{u \in \Sigma^\star | (q_0, u, Z_0) \vdash_A^\star (p, \epsilon, \epsilon), p \in Q\}.$$

Zhruba řečeno, zásobníkový automat začne v počátečním stavu q_0, na vstupní pásce má slovo u a na zásobníku pouze počáteční zásobníkový symbol Z_0. Slovo je přijato, když po jeho přečtení je (může být) zásobník vyprázdněn.

23.4.6 Jazyk přijímaný koncovým stavem

Je dán zásobníkový automat $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. *Jazyk přijímaný koncovým stavem* $L(A)$ je definován takto:

$$L(A) = \{u \in \Sigma^\star | (q_0, u, Z_0) \vdash_A^\star (p, \epsilon, \gamma), p \in F\}.$$

Zhruba řečeno, zásobníkový automat začne v počátečním stavu q_0, na vstupní pásce má slovo u a na zásobníku pouze počáteční zásobníkový symbol Z_0. Slovo je přijato, když po jeho přečtení je (může být) automat v některém koncovém stavu. To, zda je současně vyprázdněn zásobník nebo není, nehraje roli.

23.4.7 Tvrzení

Ke každému zásobníkovému automatu A existuje zásobníkový automat B takový, že

$$N(A) = L(B).$$

23.4.8 Tvrzení

Ke každému zásobníkovému automatu A existuje zásobníkový automat B takový, že

$$L(A) = N(B).$$

23.4.9 Věta

Ke každé bezkontextové gramatice $G = (N, \Sigma, S, P)$ existuje zásobníkový automat A takový, že

$$L(G) = N(A).$$

Nástin důkazu: Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$. Zkonstruujeme zásobníkový automat s jedním stavem q takto:

- $Q_A = \{q\}, q_0 = q$,
- $\Gamma_A = N \cup \Sigma$,
- $Z_0 = S$,
- $\delta_A(q, \epsilon, X) = \{(q, \alpha) \,|X \to \alpha \in P, X \in N\}$,
- $\delta_A(q, a, a) = \{(q, \epsilon)\}$, pro $a \in \Sigma$.

Zhruba řečeno, je-li na vrcholu zásobníku automatu A neterminál X, nahradíme ho v zásobníku některým pravidlem gramatiky G. Je-li na vrcholu zásobníku terminál $a \in \Sigma$, tak v případě, že a je též čten čtecí hlavou, odstraníme ho z vrcholu zásobníku a hlavu posuneme o jedno políčko doprava. Jestliže se terminální písmeno na vrcholu zásobníku nerovná prvnímu čtenému symbolu, automat se neúspěšně zastaví.

Dá se dokázat, že zásobníkový automat A přijme slovo $u \in \Sigma^\star$ prázdným zásobníkem právě tehdy, když je slovo u vygenerováno gramatikou G.

23.4.10 Věta

Ke každému zásobníkovému automatu A existuje bezkontextová gramatika G taková, že

$$N(A) = L(G).$$

Důkaz přechozí věty (jedná se o opačnou implikaci k větě 23.4.9) je obtížnější. Je třeba ho rozdělit do dvou kroků. Nejprve se dokáže, že pro každý zásobníkový automat A existuje zásobníkový automat B s jedním stavem takový, že $N(A) = N(B)$.

Pak už je jednoduché pro zásobníkový automat B s jedním stavem vytvořit bezkontextovou gramatiku G, která generuje stejná slova jako zásobníkový automat B přijal prázdným zásobníkem. Jedná se vlastně o opačný postup jako v důkazu věty 23.4.9.

23.4.11 Deterministický zásobníkový automat

O zásobníkovém automatu $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ řekneme, že je *deterministický*, jestliže splňuje následující dvě podmínky:

- Pro každé $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$ a $X \in \Gamma$ je $\delta(q, a, X)$ nejvýše jednoprvková (tj. $|\delta(q, a, X)| \leq 1$).
- Jestliže pro nějaké $q \in Q$ a $X \in \Gamma$ je $\delta(q, \epsilon, X)$ neprázdné, pak pro každé $a \in \Sigma$ je $\delta(q, a, X)$ prázdná množina.

Uvědomte si, že předchozí dvě podmínky zajišťují, že v každém okamžiku máme vždy nejvýše jednu možnost, jak pokračovat.

23.4.12 Jazyky přijímané deterministickým zásobníkovým automatem

Stejně jako u (nedeterministických) zásobníkových automatů rozlišujeme i u deterministických zásobníkových automatů příjímání koncovým stavem a přijímání prázdným zásobníkem. Tj. pro daný deterministický zasobníkový automat $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ je

$$L(A) = \{u | (q_0, u, Z_0) \vdash_A^\star (p, \epsilon, \gamma), p \in F\}$$

$$N(A) = \{u | (q_0, u, Z_0) \vdash_A^\star (p, \epsilon, \epsilon)\}.$$

23.4.13 Tvrzení

Pro každý deterministický zásobníkový automat A existuje deterministický zásobníkový automat B takový, že

$$N(A) = L(B).$$

Jinými slovy, každý jazyk přijímaný deterministickým zásobníkovým automatem prázdným zásobníkem je také přijímán (nějakým) deterministickým zásobníkovým automatem koncovým stavem.

23.4.14 Bezprefixový jazyk

je jazyk, který je příjímán nějakým deterministickým zásobníkovým automatem prázdným zásobníkem.

Obdoba tvrzení 23.4.8 pro deterministické zásobníkové automaty neplatí. Jestliže totiž deterministický zásobníkový automat A přijme slovo u prázdným zásobníkem, pak nemůže prázdným zásobníkem přijmout žádné slovo uv, kde $v \neq \epsilon$.

23.4.15 Deterministický jazyk

je jazyk, který je přijímán některým deterministickým zásobníkovým automatem koncovým stavem.

23.5 Vlastnosti bezkontextových jazyků

23.5.1 Chomského normální tvar

Je dána bezkontextová gramatika $G = (N, \Sigma, S, P)$. Řekneme, že gramatika G je v *Chomském normálním tvaru*, jestliže má pouze pravidla tvaru

$$A \to BC, A \to a \quad \text{pro } A, B, C \in N, a \in \Sigma.$$

23.5.2 Věta

Pro každou bezkontextovou gramatiku G existuje bezkontextová gramatika G' v Chomského normálním tvaru taková, že

$$L\left(G'\right) = L\left(G\right) - \{\epsilon\}.$$

23.5.3 CYK

Jedná se o algoritmus, který pro danou bezkontextovou gramatiku G v Chomského normálním tvaru a pro dané terminální slovo w rozhodne, zda $w \in L\left(G\right)$.

Označíme $G = (N, \Sigma, S, P)$ a $w = a_1 a_2 \dots a_k$. Postupně vytváříme množiny $X_{i,j}$ pro $1 \leq i \leq j \leq k$, kde

$$X_{i,j} = \{A \in N | A \Rightarrow_G^\star a_i a_{i+1} \dots a_j\}.$$

Platí

$$A \in X_{i,i} \quad \text{iff} \quad A \to a_i \in P.$$

Navíc

$$X_{1,k} = \{A \in N | A \Rightarrow_G^\star a_1 a_2 \dots a_k = w\}.$$

Dále si uvědomte, že $A \Rightarrow_G^\star a_i a_{i+1} \dots a_j$ iff existují neterminály B, C takové, že

$$A \to BC \in P, \text{ kde} \quad \text{buď} \quad B \Rightarrow_G^\star a_i \quad \text{a} \quad C \Rightarrow_G^\star a_{i+1} \dots a_j$$

$$\text{nebo } B \Rightarrow_G^\star a_i a_{i+1} \quad \text{a} \quad C \Rightarrow_G^\star a_{i+2} \dots a_j$$

$$\text{nebo } B \Rightarrow_G^\star a_i a_{i+1} a_{i+2} \quad \text{a} \quad C \Rightarrow_G^\star a_{i+3} \dots a_j$$

$$\dots$$

$$\text{nebo } B \Rightarrow_G^\star a_i \dots a_{i+j-1} \quad \text{a} \quad C \Rightarrow_G^\star a_{j,j}.$$

Předpokládejme, že máme zkonstruovány všechny množiny $X_{p,q}$, kde $q-p<n$. Pak množiny $X_{i,j}$ pro $j-i=n$ utvoříme takto:

$$A \in X_{i,j} \text{ iff } \exists A \to BC \in P \text{ tak, že} \quad \text{buď} \quad B \in X_{i,i} \quad \text{a} \quad C \in X_{i+1,j}$$

$$\text{nebo} \quad B \in X_{ii+1} \quad a \quad C \in X_{i+2,j}$$

$$\text{nebo} \quad B \in X_{i,i+2} \quad a \quad C \in X_{i+3,j}$$

$$\ldots$$

$$\text{nebo} \quad B \in X_{i,j-1} \quad a \quad C \in X_{j,j}$$

Začínám tedy konstrukcí množin $X_{i,i}$, $i=1,2,\ldots,k$, následuje pak $k-1$ množin $X_{i,i+1}$, $i=1,2,\ldots,k-1$, atd. dvě množiny $X_{1,k.-1}$, $X_{2,,k}$ a nakonec jednu množinu $X_{1,k}$ a to podle následujícího postupu:

$$X_{i,j}=\{A \in N | \exists A \to BC \in P \text{ tak, že } B \in X_{i,i+m}, C \in X_{i+m+1,j}\}.$$

Platí $w \in L(G)$ právě tehdy, když $S \in X_{1,k}$.

23.5.4 Příklad

Je dána gramatika G pravidly

$$S \to AB|BC$$

$$A \to BA|a$$

$$B \to CC|b$$

$$C \to AB|a$$

Pomocí algoritmu CYK rozhodněte, zda slovo *aabab* je generováno bezkontextovou gramatikou G.

Řešení: Konstrukci množin $X_{i,j}$ pro $1 \le i \le j \le 5$ si znázorníme do tabulky. Tabulka bude mít 5 řádků a 5 sloupců, kde vyplněných bude jen ta část, která se nachází „pod diagonálou". Poslední řádek obsahuje pět množin $X_{1,1}$, $X_{2,2}$, $X_{3,3}$, $X_{4,4}$ a $X_{5,5}$. Předposlední řádek obsahuje čtyři množiny $X_{1,2}$, $X_{2,3}$, $X_{3,4}$ a $X_{4,5}$. Řádek, který je třetí od spodu (a také shora) obsahuje tři množiny $X_{1,3}$, $X_{2,4}$ a $X_{3,5}$. Řádek, který je čtvrtý od spodu (a druhý shora) obsahuje dvě množiny $X_{1,4}$ a $X_{2,5}$. Nejvyšší řádek obsahuje jednu množinu $X_{1,5}$.

S,C				
S,A,C	B			
B	B	S,C		
B	S,C	S,A	S,C	
A,C	A,C	B	A,C	B
a	a	b	a	b

Z předchozí tabulky také můžeme odvodit derivace slova *aabab* gramatikou G. Jedna z takových derivací je např. tato:

$$S \Rightarrow AB \Rightarrow aB \Rightarrow aCC \Rightarrow aaC \Rightarrow aaAB \Rightarrow aaBAB \Rightarrow aabAB \Rightarrow aabaB \Rightarrow aabab.$$

23.5.5 Pumping lemma pro bezkontextové gramatiky

Pro každý bezkontextový jazyk L existuje kladné přirozené číslo m takové, že jestliže některé slovo z obsažené v jazyce L má délku alespoň m, pak z lze psát ve tvaru $z = uvwxy$, kde

- $|vwx| \leq m$, (tj. prostřední část není příliš dlouhá),
- $vx \neq \epsilon$ (tj. aspoň jedno ze slov v, x není prázdné),
- pro všechna $i \geq 0$ platí $uv^iwx^iy \in L$, (tj. v a x se dají do slova z „napumpovat" a stále dostaneme slovo z jazyka L).

23.5.6 Využití Pumping lemmatu pro bezkontextové gramatiky

Ukážeme, že jazyk $L = \{0^n1^n2^n | n \geq 0\}$ není bezkontextový.

Zdůvodnění: Předpokládejme, že jazyk L je bezkontextový. Pak existuje kladné číslo m z Pumping lemmatu. V jazyce L leží slovo $z = 0^m1^m2^m$. Podle Pumping lemmatu existují slova u, vw, x, y taková, že

$$0^m1^m2^m = uvwxy,\ |vx| > 0,\ |vwx| \leq m \text{ a } uv^iwx^iy \in L \text{ pro } i \geq 0.$$

Ukážeme, že slovo $uv^0wx^0y = uwy \notin L$. To bude hledaný spor.

Podmínka $|vwx| \leq m$ znamená, že slovo vwx buď neobsahuje písmeno 2 nebo neobsahuje písmeno 0.

Jestliže vwx neobsahuje písmeno 2, pak slova v, x obsahují pouze 0 nebo 1, to je nejvýše dva ze tří písmen 0,1,2.

Jestliže vwx neobsahuje písmeno 0, pak slova v, x obsahují pouze 1 nebo 2, to je nejvýše dva ze tří písmen 0,1,2.

To ale znamená, že v obou případech nemůže slovo uwy (tj. slovo z, ze kterého jsme vypustili slova v a x) obsahovat stejný počet všech tří písmen 0,1,2.

23.6 Uzávěrkové vlastnosti bezkontextových jazyků

23.6.1 Tvrzení

Bezkontextové jazyky jsou uzavřeny na sjednocení.

To znamená, jsou-li L_1 a L_2 dva bezkontextové jazyky, pak také jazyk $L_1 \cup L_2$ je bezkontextový.

23.6.2 Tvrzení

Bezkontextové jazyky jsou uzavřeny na zřetězení.

To znamená, jsou-li L_1 a L_2 dva bezkontextové jazyky, pak také jazyk $L_1 L_2$ je bezkontextový.

23.6.3 Tvrzení

Bezkontextové jazyky jsou uzavřeny na Kleeneho operaci $\star$.

To znamená, je-li L bezkontextový jazyk, pak také jazyk $L\star$ je bezkontextový.

23.6.4 Tvrzení

Bezkontextové jazyky jsou uzavřeny na reverzi.

To znamená, je-li L bezkontextový jazyk, pak také jazyk L^R je bezkontextový.

23.6.5 Tvrzení

Bezkontextové jazyky nejsou uzavřeny na průnik.

To znamená, jsou-li L_1 a L_2 dva bezkontextové jazyky, pak jazyk $L_1 \cap L_2$ nemusí být bezkontextový.

23.6.6 Tvrzení

Bezkontextové jazyky nejsou uzavřeny na doplněk.

To znamená, je-li L bezkontextový jazyk, pak jeho doplněk $\bar{L}$ nemusí být bezkontextový.

23.6.7 Tvrzení

Třída bezkontextových jazyků je uzavřena na průniky s regulárními jazyky.

To znamená, je-li L bezkontextový jazyk a R regulární jazyk, pak jazyk $L \cap R$ je bezkontextový.

23.6.8 Tvrzení

Třída bezkontextových jazyků je uzavřena na substituce.

24 Turingovy stroje. (A4B01JAG)

Turingův stroj si můžeme představit takto: skládá se

- z řídící jednotky, která se může nacházet v jednom z konečně mnoha stavů,
- potenciálně nekonečné pásky rozdělené na jednotlivá pole a
- hlavy, která umožňuje číst obsah polí a přepisovat obsah polí pásky.

Na základě informace X, která je přečtena na pásce, a na základě stavu q, ve kterém se nachází řídící jednotka Turingova stroje, se řídící jednotka přesune do stavu p, pole pásky přepíše na Y a hlava se přesune buď doprava nebo doleva (tato akce je popsána tzv. přechodovou funkcí).

24.1 Formální definice

Turingův stroj je sedmice $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, kde

- Q je konečná množina stavů,
- Σ je konečná množina vstupních symbolů,
- Γ je konečná množina páskových symbolů, přitom $\Sigma \subset \Gamma$,
- B je prázdný symbol (též nazývaný *blank*), jedná se o páskový symbol, který není vstupním symbolem, (tj. $B \in \Gamma \setminus \Sigma$),
- δ je přechodová funkce, tj. parciální zobrazení z množiny $Q \times \Gamma$ do množiny $Q \times \Gamma \times \{L, R\}$, (zde L znamená pohyb hlavy o jedno pole doleva, R znamená pohyb hlavy o jedno pole doprava),
- $q_0 \in Q$ je počáteční stav a
- $F \subseteq Q$ je množina koncových stavů.

24.2 Konfigurace

Konfiguraci Turingova stroje plně popisuje páska, pozice hlavy na pásce a stav, ve kterém se nachází řídící jednotka. Jestliže na pásce jsou v prvních k polích symboly $X_1 X_2 \dots X_k$, všechna pole s větším číslem již obsahují pouze B, řídící jednotka je ve stavu q a hlava čte symbol X_i, tak danou konfiguraci zapisujeme

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_k.$$

24.3 Počátek práce Turingova stroje

Na začátku práce se Turingův stroj nachází v počátečním stavu q_0, na pásce má na prvních n polích vstupní slovo $a_1 a_2 \dots a_n$ ($a_i \in \Sigma$), ostatní pole obsahují blank B a hlava čte první pole pásky, tj. symbol a_1. Tedy formálně je počáteční konfigurace $q_0 a_1 \dots a_n$.

24.4 Krok Turingova stroje

Předpokládejme, že se Turingův stroj nachází v konfiguraci $X_1 X_2 \dots X_{i-1} q X_i \dots X_k$. Pak v jednom kroku udělá následující:

Jestliže $\delta(q, X_i) = (p, Y, R)$, stroj se přesune do stavu p, na pásku napíše symbol Y a hlavu posune o jedno pole doprava. Formálně to zapisujeme:

$$X_1 X_2 \dots X_{i-1} q X_i \dots X_k \vdash X_1 X_2 \dots X_{i-1} Y p X_{i+1} \dots X_k.$$

Jestliže $\delta(q, X_i) = (p, Y, L)$, a hlava nečte nejlevnější pole, stroj napíše na pásku Y (místo X_i) a posune hlavu o jedno pole doleva. Formálně to zapisujeme:

$$X_1 X_2 \dots X_{i-1} q X_i \dots X_k \vdash X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_k.$$

Jestliže $\delta(q, X_i) = (p, Y, L)$, a hlava čte první pole pásky, tj. pole, které je "nejvíce vlevo", nebo jestliže $\delta(q, X_i)$ není definováno, stroj se neúspěšně zastaví.

24.5 Výpočet Turingova stroje

je posloupnost jeho kroků, která začíná v počáteční konfiguraci. Tedy jedná se o reflexivní a tranzitivní uzávěr $\vdash^\star$ relace $\vdash$ (na množině všech konfigurací daného Turingova stroje).

24.6 Jazyk přijímaný Turingovým strojem

Vstupní slovo $w \in \Sigma^\star$ je *přijato* Turingovým strojem právě tehdy, když se Turingův stroj při práci na slově w dostane do koncového stavu. Tedy formálně: slovo $w \in \Sigma^\star$ je *přijato Turingovým strojem* právě tehdy, když

$$q_0 w \vdash^\star \alpha q \beta \quad \text{pro nějaké } q \in F \text{ a } \alpha, \beta \in \Gamma^\star.$$

Množina slov $w \in \Sigma^\star$, které Turingův stroj přijímá, se nazývá *jazyk přijímaný* Turingovým strojem M a značíme ho $L(M)$.

Věta: Turingovy stroje přijímají právě třídu jazyků typu 0. Přesněji:

Ke každé gramatice G typu 0 existuje Turingův stroj M takový, že

$$L(G) = L(M).$$

Ke každému stroji M existuje gramatika G typu 0 taková, že

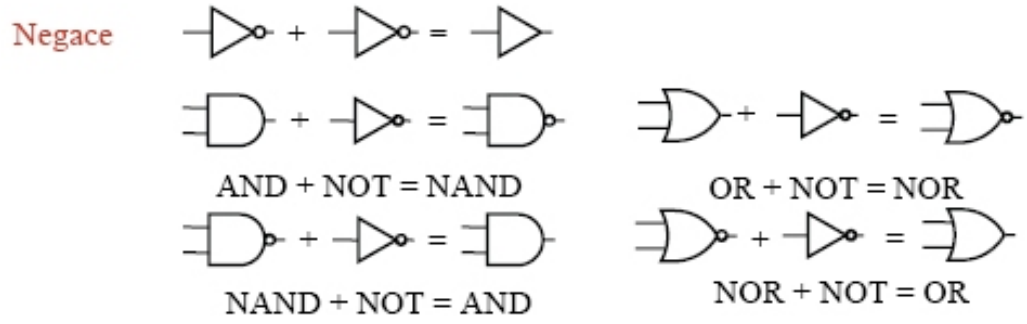
$$L(M) = L(G).$$

Kombinační logické obvody, hazardy. Minimalizace logických funkcí. Kombinační obvody výpočetní techniky: multiplexory, demultiplexory, dekodéry, komparátory, sčítačky, obvody zrychleného přenosu. Programovatelné logické obvody. (A0B35SPS)

25.1 Kombinační logické obvody

- jejich výstupní stav je dán pouze kombinací vstupních stavů
- nejjednodušší jsou základní logické funkce AND, NAND, OR, XOR, NOT

Buffer	$Y = a$	Invertor	$Y = \overline{a}$
AND	$Y = ab$	NAND	$Y = \overline{ab}$
OR	$Y = a + b$	NOR	$Y = \overline{a + b}$
XOR	$Y = a\overline{b} + \overline{a}b$ $Y = a \oplus b$	XNOR	$Y = ab + \overline{a}\,\overline{b}$ $Y = a \equiv b$



25.1.1 Hazardy

- hazard = chybový stav, kombinační obvod má na výstupu chybnou hodnotu
- nežádoucí
- rozdělujeme na
  - **statický hazard:** může dojít k jednorázové chybě na výstupu, vzniká často nestejně dlouhou cestou signálu. Každé logické hradlo má v reálných podmínkách určitou dobu zpoždění, jsou-li tedy na vstup nějakého hradla přivedeny signály každý s odlišným zpožděním, v jednu chvíli hradlo rozhoduje např. na základě jednoho aktuálního vstupu a jednoho "zastaralého", který ještě nestačil přes cestu s dlouhým zpožděním dorazit.

– **dynamický hazard:** může dojít k opakovaným chybám, obvod může oscilovat mezi stavy 0 a 1 na předem nedefinované frekvenci. K tomuto jevu dochází např. u neplatných vstupů přivedených na některé klopné obvody, nebo zavedením zpětné vazby.

25.2 Minimalizace logických funkcí

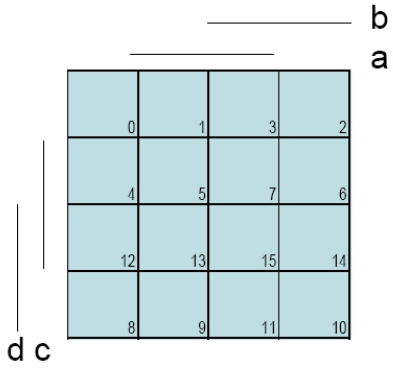
Snaha o snížení počtu potřebných operací (a při realizaci tedy logických hradel) k vykonání funkce.

K minimalizaci lze využít:

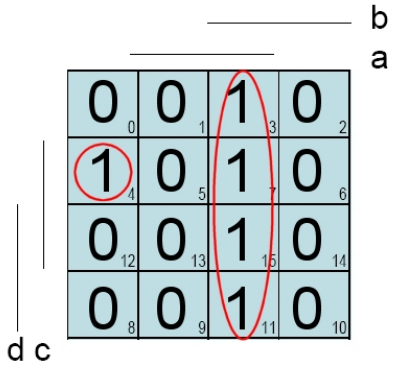
- úpravy pomocí booleovy algebry s využitím pravidel:

Vlastnost	AND OR 0 1	OR AND 1 0
Komutativita	$a + b = b + a$	$a \cdot b = b \cdot a$
Identita	$a + 0 = a$	$a \cdot 1 = a$
Distributivita	$a + (b \cdot c) = (a + b) \cdot (a + c)$	$a \cdot (b + c) = a \cdot b + a \cdot c$
Komplementarita	$a + a' = 1$	$a \cdot a' = 0$
Idempotence	$a + a = a$	$a \cdot a = a$
Agresivita	$a + 1 = 1$	$a \cdot 0 = 0$
Dvojí negace	$(a')' = a$	
Asociativita	$a + (b + c) = (a + b) + c$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
DeMorgan	$\mathbf{(a + b)' = a' \cdot b'}$	$\mathbf{(a \cdot b)' = a' + b'}$
Absorpce	$a + (a \cdot b) = a$	$a \cdot (a + b) = a$
Sloučení	$(x \cdot y) + (x \cdot y') = x$	$(x + y) \cdot (x + y') = x$

- minimalizaci pomocí Karnaughovy mapy:
  pravdivostní tabulku funkce přepíšeme do Karnaughovy mapy, skupiny shodných hodnot lze potom použít k definici funkce jednodušším zápisem. Karnaughova mapa pro zápis funkce 4 proměnných, čára u proměnné popisuje stav, kdy je proměnná rovna 1. (např. pozice označená 0 - všechny proměnné jsou v 0, pozice označená 5 - proměnné a a c jsou v 1)



přílkad pro funkci f = ab¬c¬d + abc¬d + abcd + ab¬cd + ¬a¬bc¬d:

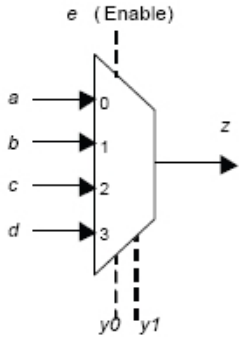


z mapy je vidět, že nám k popisu funkce stačí jednodušší výraz (získaný vytvořením smyček)
f = ab + ¬a¬bc¬d

25.3 Kombinační obvody výpočetní techniky

Multiplexor

je obvod, který má n vstupů, adresní vstupy a jeden výstup. Adresa určuje, který ze vstupů se přepošle na výstup.



Demultiplexor

je opak multiplexoru, obvod má jeden vstup, adresní vstupy a n výstupů. Adresa určuje, na který výstup se přepošle vstup.

Dekodér

je totéž jako demultiplexor, pouze má místo vstupu konstantu.

Komparátor

je obvod, který zkoumá dvě binární čísla přivedená na jeho vstupy A a B na rovnost či nerovnost. Má zpravidla 3 výstupy, těm je přiřazena funkcionalita A == B, A > B, A < B.

Poloviční jednobitová binární sčítačka

je obvod, který zvládne sečíst dvě jednobitová čísla. Na jeho výstupu je součet čísel a přetečení do vyššího řádu (binárně 1 + 0 = 1, ale 1 + 1 = 10, výstup 0, přetečení 1).

Úplná jednobitová binární sčítačka

se od poloviční liší tím, že má jako další vstup také přetečení z nižšího řádu, je tedy možné sčítačky řetězit a počítat s n-bitovými čísly (a ne pouze s jednobitovými).

Obvody zrychleného přenosu

Jelikož přenosy u zřetězeného sčítání "probublávají" postupně do vyšších řádů, sčítání je omezeno prodlevou jednotlivých hradel. To má negativní vliv na rychlost sčítání větších čísel, kdy celková prodleva narůstá s bitovou velikostí čísel lineárně. Proto se používají sčítačky s tzv. Carry Look-Ahead, kdy přenosy mezi řády nejsou propojeny přímo, ale obsluhuje je vnější logika, která se zároveň snaží i predikovat stavy přetečení paralelním sčítáním na několika řádech najednou.

25.4 Programovatelné logické obvody

Jedná se o obvody, které nemají předem určenou funkcionalitu, tu lze programovat. V principu takový obvod obsahuje veliké množství různých hradel a programováním se pak myslí vytvoření spojů mezi jednotlivými hradly. Tato spojení pak definují, jakou logickou funkci bude daný obvod vykonávat. Mezi nejznámější druhy patří (C)PLD ((Complex) Programmable logic device) a dnes rozšířené obvody FPGA (tzv. hradlová pole).

Hlavním rozdílem obvodů FPGA oproti PLD je jejich využívání Look-Up tabulek, čili místo toho, aby byla logika realizována ze základních hradel, ukládají obvody FPGA pravdivostní tabulky.

Samotné nastavení propojů mezi hradly může být buď jednorázové, nebo ukládáno do paměti. Tato paměť může být permanentní, není to ale podmínkou, často se nastavení hradlových polí rekonfigurují při spuštění systému.

Výhody oproti klasické logice jsou zřejmé - zrychlení a snížení nákladů na vývoj aplikací a možnost změnit realizovaný obvod (a to i za běhu).

Konečný automat a jeho minimalizace. Syntéza asynchronních sekvenčních logických obvodů jako kombinačních obvodů se zpětnou vazbou. Struktura základních synchronních klopných obvodů. Syntéza sekvenčních logických obvodů používaných v počítačích.

26.1 Konečný automat a jeho minimalizace

26.1.1 Konečný automat

šestice (Q,Σ,Y,δ,q_0,ω) kde:

Q ... konečná množina všech stavů

Σ ... konečná množina vstupních všech symbolů

Y ... konečná množina výstupních všech symbolů

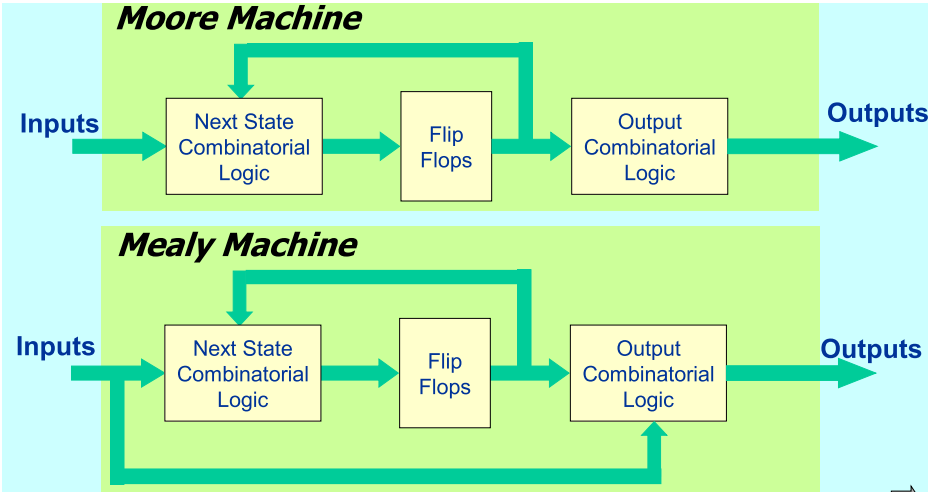
δ ... přechodová funkce $\delta : Q \times \Sigma \mapsto Q$

$\mathbf{q_0}$ ... počáteční stav, $q_0 \epsilon Q$ $\delta(q_0, a) = p$, $a \epsilon \Sigma$, $p \epsilon \Sigma$

ω ... výstupní funkce:

Moore $\omega : Q \mapsto Y$

Mealy $\omega : \Sigma \times Q \mapsto Y$



- Ke každému Mealyho automatu M existuje Moorův automat M' podobný (= nejsme schopni rozlišit jejich chování z vnějšího pozorování)

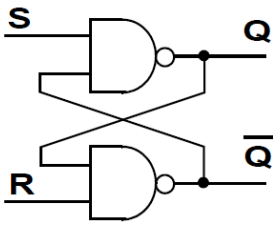
26.1.2 Minimalizace

Nic jsem nenašel, asi bych se znímil co to jsou DFA (akceptor) ,řekl že jsou velmi často používané a minimalizoval pouze DFA, viz JAG

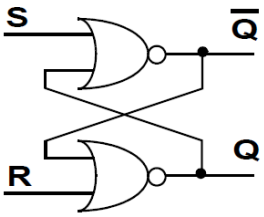
26.2 Základních asynchroní obvody

latch = jednobitová paměť bez hodinového signálu (clok)

26.2.1 RS

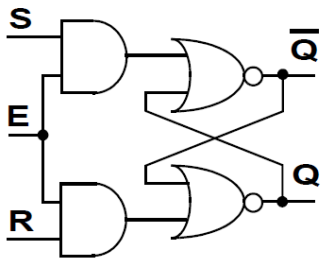


S	R	Q	$\overline{Q}$	Function
0	0	1-?	1-?	Indeterminate State
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q^+	$\overline{Q}^+$	Storage State



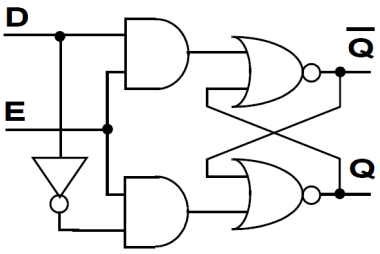
S	R	Q	$\overline{Q}$	Function
0	0	Q^+	$\overline{Q}^+$	Storage State
0	1	0	1	Reset
1	0	1	0	Set
1	1	0-?	0-?	Indeterminate State

26.2.2 RS s enabled

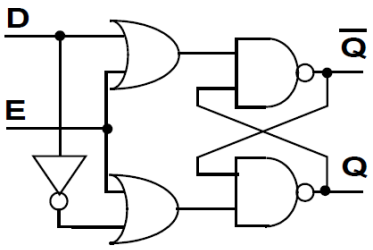


E	S	R	Q	$\overline{Q}$	Function
0	x	x	Q^+	$\overline{Q}^+$	Storage State
1	0	0	Q^+	$\overline{Q}^+$	Storage State
1	0	1	0	1	Reset
1	1	0	1	0	Set
1	1	1	0-?	0-?	Indeterminate State

26.2.3 D latch



E	D	Q	Function
0	x	Q^+	Storage State
1	0	0	Transparent Mode
1	1	1	Transparent Mode



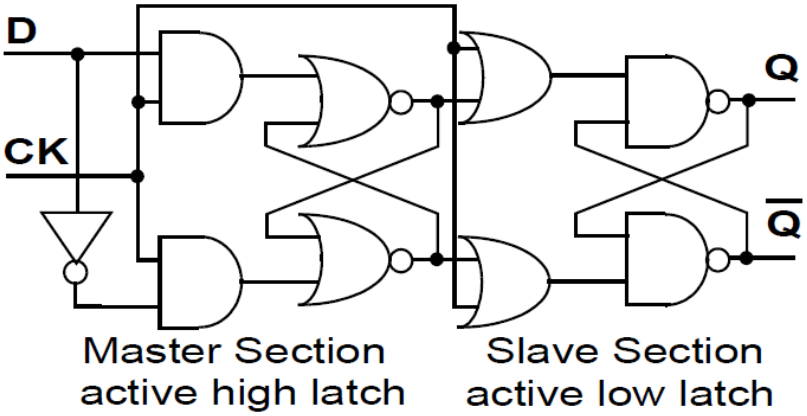
E	D	Q	Function
1	x	Q^+	Storage State
0	0	0	Transparent Mode
0	1	1	Transparent Mode

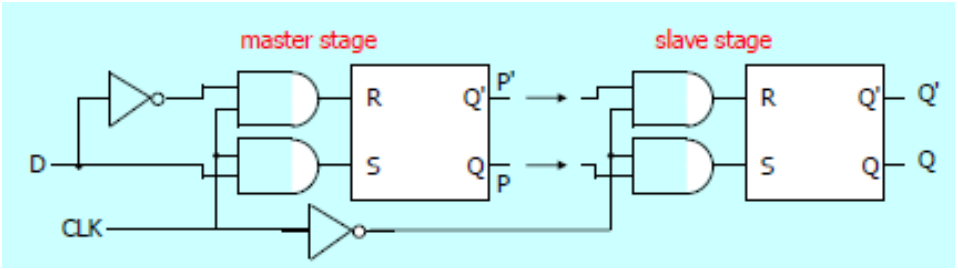
26.3 Synchronních klopné obvody

flip-flop = klopný obvod řízený hodinovým signálem(clok)

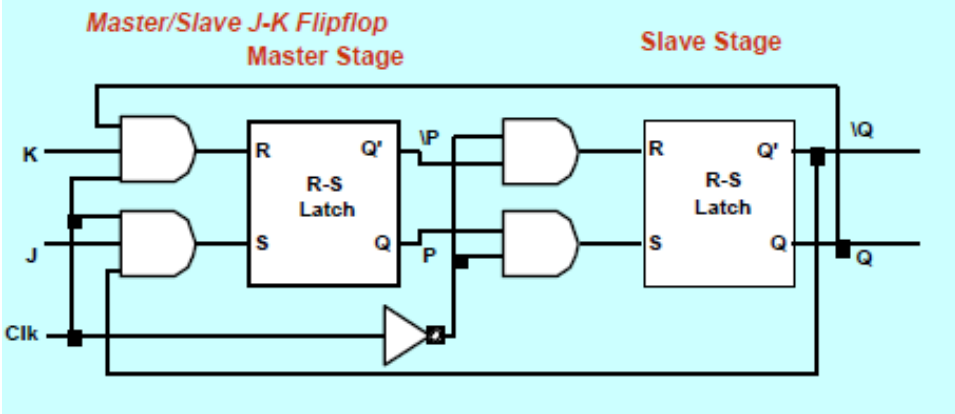
26.3.1 D flip-flop

Na obrázku je schéma, které reaguje na sestupnou hranu, jinak se chová jako D latch





26.3.1.1 J-K Flip Flop

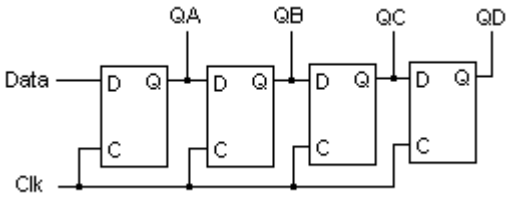


J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}$(t)

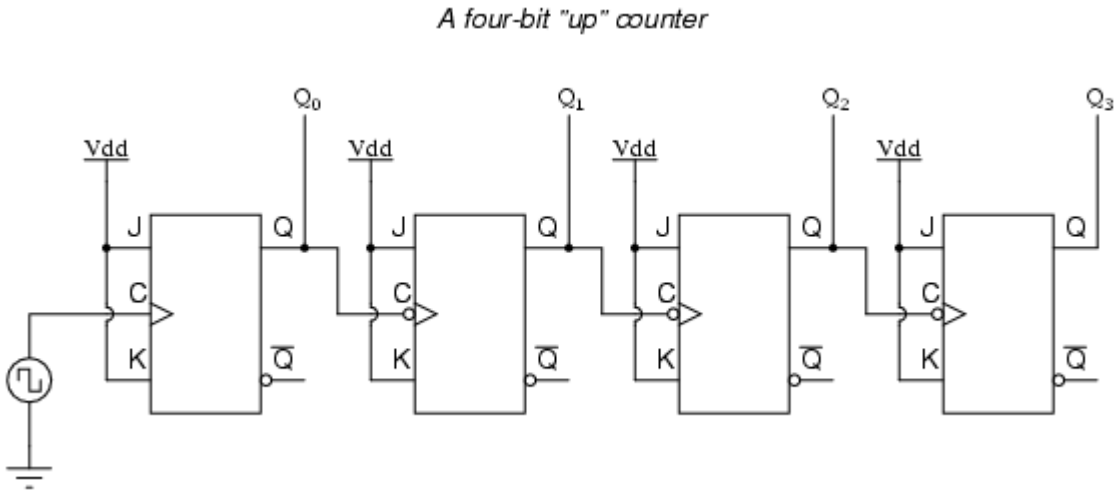
26.4 Sekvenčních logických obvodů používaných v počítačích

- Binární a dekadické čítače, Grayovyčítače
- posuvné registry
- řadiče přerušení

26.4.1 4-bit posuvný registr

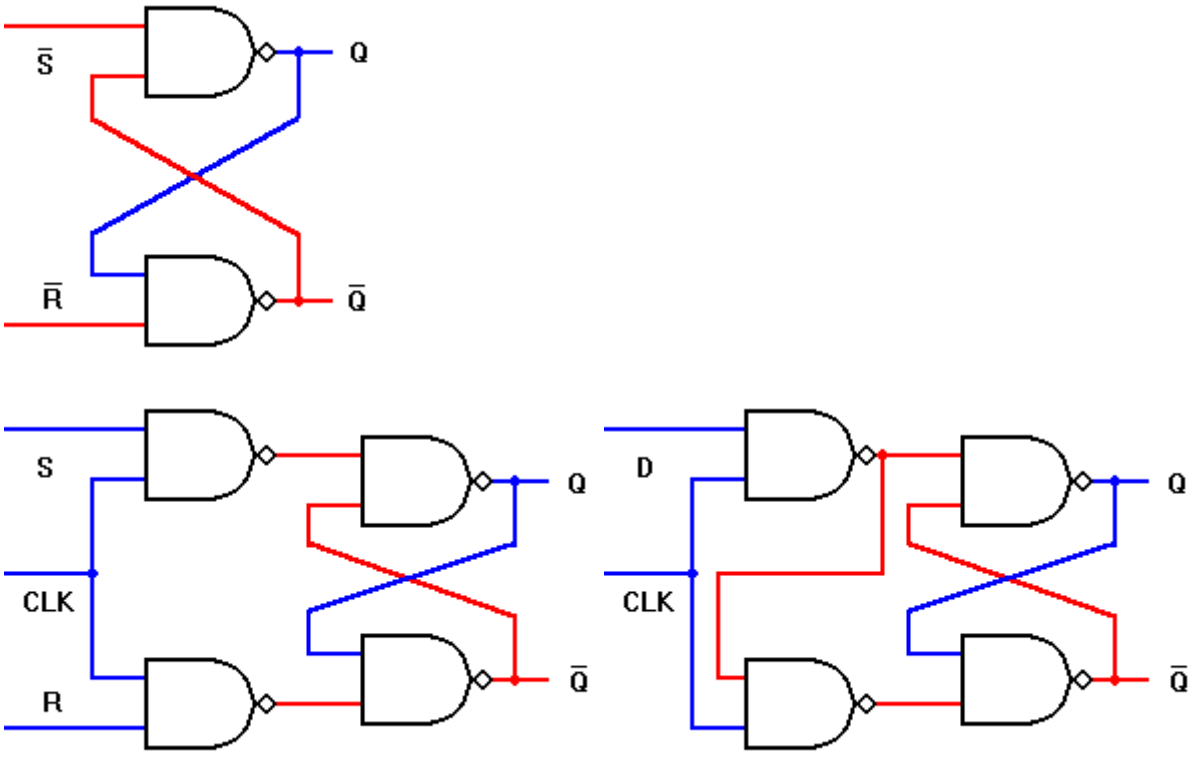


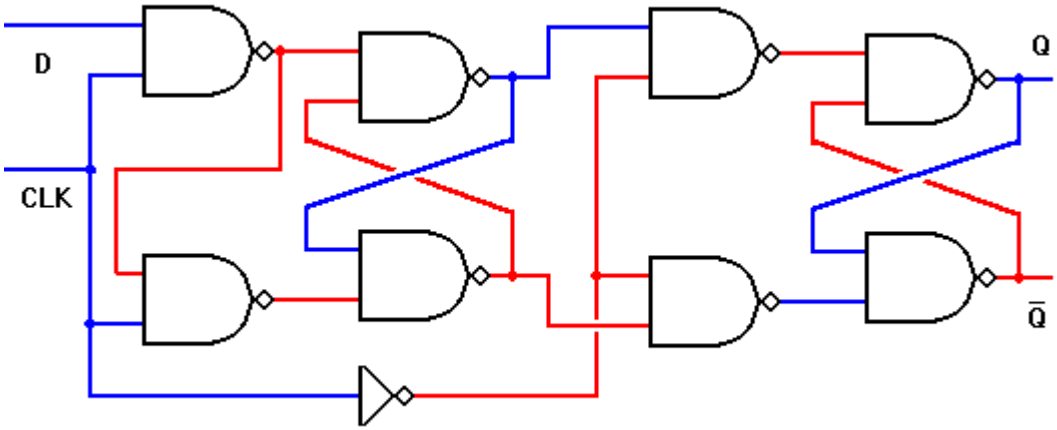
26.4.2 4-bitový vzestupný čítač



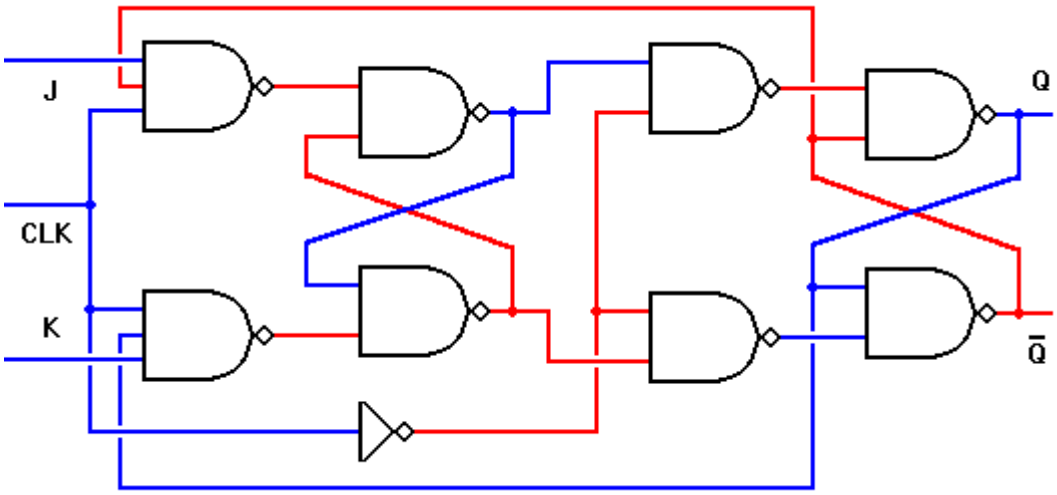
26.5 Dolpnění

Tady jsou latche a flip-flopy pouze z NANDů, ty se možná lépe pamatují





Pro JK níže - stavy se mění pouze na sestupnou hranu (v případě clocku)



Pevný a programovatelný řadič. Mikroprogramový automat. Klasická architektura počítače, von Neumannova a harvardská architektura. Struktura CPU, datové a adresní registry, čítač instrukcí, ukazatel zásobníku, typy instrukcí (A0B35SPS)

27.1 Řadiče

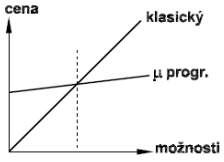
- řadič anglicky control unit
- jasně odlišitelná část systému, která řídí nějaký úkon
  - např.: řadiče displaye, jednotka řídící teplotu vody , atd..
- v CPU se stará o řízení toku dat a o řízení práce všech jednotek, zejména ALU, a to v závislosti na právě vykonávané instrukci

27.1.1 Programovatelný řadič

- varianta sekvenčního obvodu ralizovaná přes paměť
- flexibilní

27.1.2 Pevný řadič = Řadič klasický, též obvodově realizovaný, tedy tzv. obvodový

- rychlejší
- automat realizovaný přes sekvenční obvody
- ve velmi jednoduchých případech levnější

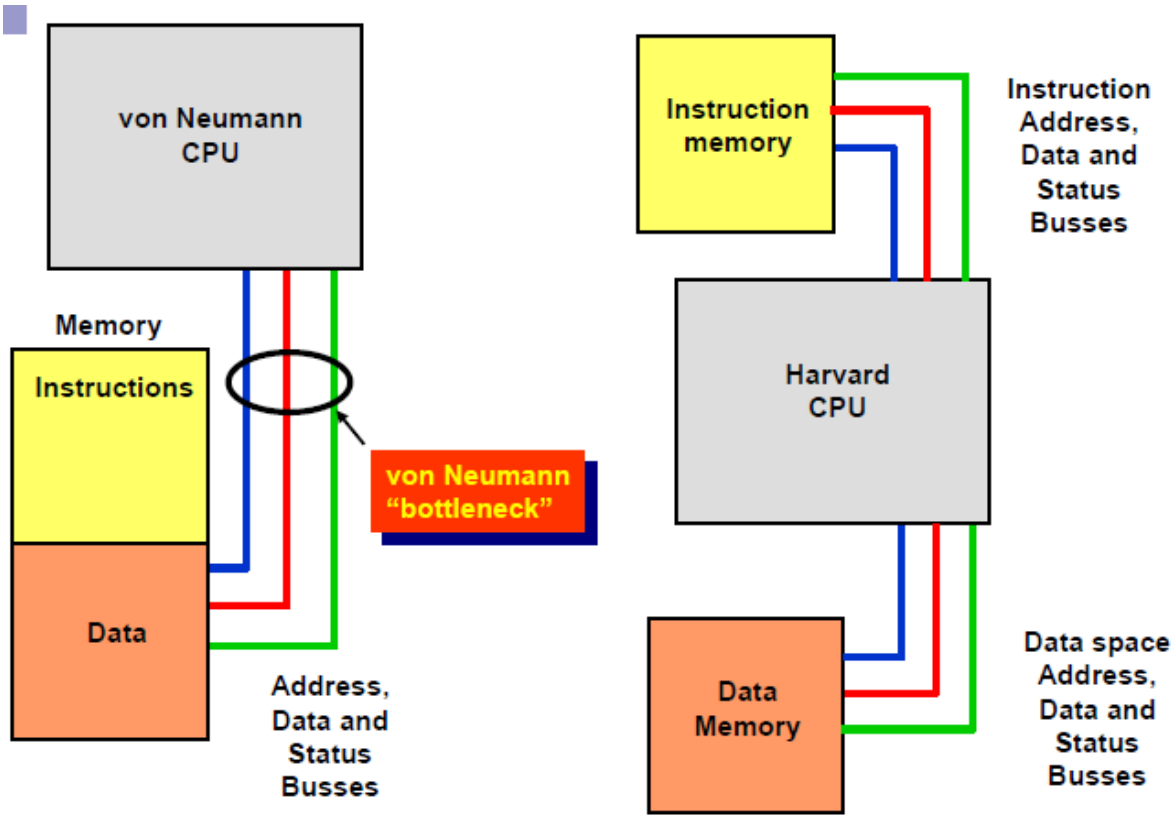


27.2 Mikroprogramovatelný automat

- řadič, který nefunguje s fixní konfigurací, ale používá tzv. mikrokód
- použití v CPU -> instrukce z instrukční sady se překládá na sekvenci mikroinstrukcí, nahrazuje rozsáhlou logiku pevného řadiče
- mikroinstrukce definují, které hardwarové části je potřeba propojit, aby byla vykonána samotná instrukce

- oproti fixní konfiguraci má výhodu v možnosti opravy chyb procesoru pomocí aktualizace tabulky překladu instrukce -> mikroprogram
- mikroinstrukce se provádějí velmi rychle a lze je paralelizovat
- příklad mikroprogramu jedné instrukce přičítání např.:
  - přiveď registr AX k ALU jako první operand
  - přiveď dočasný registr k ALU jako druhý operand
  - nastav ALU do režimu sčítání
  - nastav carry na 0
  - ulož výsledek do regstru AX
  - nastav příznaky
- Tato sekvence je částí vykonávání jedné instrukce. V kontrastu, pevný řadič by operandy ALU dekódoval a logickým součinem povoloval / zakazoval přímo z operačního znaku instrukce. Stejně tak by podle operačního kódu řídil režim činnosti ALU a umístění výsledku. Pro větší počet instrukcí narůstá velikost potřebné logiky.

27.3 Architektura počítače

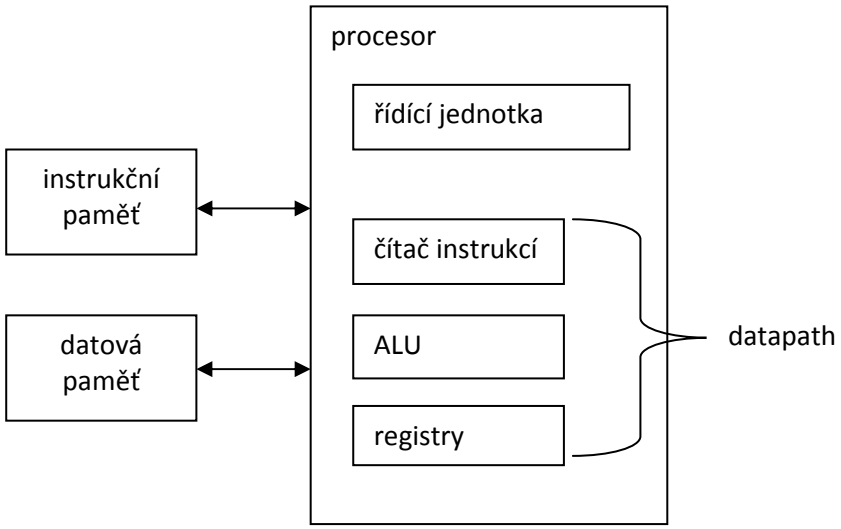


Von_Neumanova jednodušší, pomalejší, možnost rodělit paměť data/instrukce dle potřeby

Harvardská paměť pro instrukce a data fyzicky oddělena

Sběrnice propojující paměti se skládá ze tří sběrnic: adresní, datové, řídící

27.4 Struktura CPU



řídící_jednotka (řadič) zajišťuje součinost jednotlivých částí CPU

ALU aritmeticko-logická jednotka (může jich být i více) zajišťuje všechny aritmetické a logické výpočty

PC program counter - čítač instrukcí - uchovává stav paměti, procesor má vždy po resetu nastavenou určitou hodnotu

registry

- *datové* - ukládání hodnot
- *adresní* - uchovávají adresy odkud mají být dada načítána / kam ukládána
- procesor může vykonávat aritmetické/logické operace pouze nad daty v registrech

27.4.1 ukazatel zásobníku - stack pointer - SP

- uchováná adresu posledního záznamu uloženého na zásobníku
- obvykle "růst dolů": při push se SP zvětší, při pop se SP snižuje
- zásobník pracuje na principu LIFO (last in, first out)
- používá se pro uchování návratové adresy při volání funkcí, nebo zároveň pro ukládání parametrů pro volanou funkci
- sekundární využití je pro ukládání proměnných programu, ke kterým nepřistupujeme instrukcemi PUSH a POP, ale běžnými ukazateli

27.4.2 typy instrukcí

CISC complex instruction set computer

- instrukce se skládá z několika kroků ⟶zpomalování procesoru
- různě dlouhé , různě trvající instrukce

RISC reduced instruction set computing

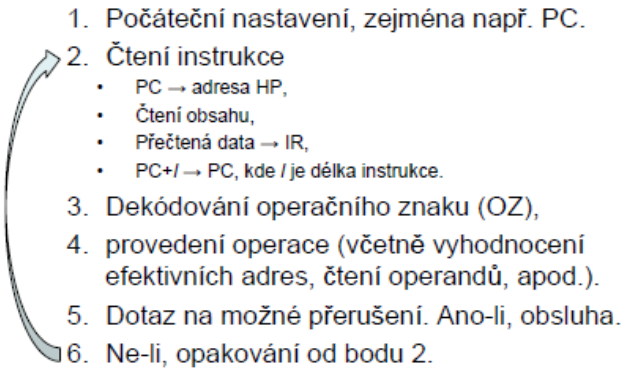
- snížení počtu instrukcí, snaha dosáhnout 1takt=1instrukce

27.4.3 jeden cyklus cpu - asi není nezbytné

nevíme co znamená HP.

Tak jako tak, prvním krokem je načtení obsahu paměti na adrese PC, tak jestli tím nemysleli adresu v hlavní paměti?? Tohle je detail.

.



28 Struktury a hierarchie pamětí. Způsoby adresace. Různá šíře adres generovaných CPU (logických adres) a fyzických adres paměti. Mapování, stránkování, segmentace. Přerušení a výjimky. Zdroje přerušení, přerušovací vektory. DMA přenosy. (A0B35SPS)

(pozn. autora: V předmětu SPS toto není. Čerpal jsem z přednášek OSS a APO.)

28.1 Struktury a hierarchie pamětí

- Rychlé a malé paměti jsou umístěny blízko CPU.
- Větší a pomalejší paměti jsou dále od CPU.
- Princip cachování nejčastěji používaných dat v rychlých pamětech - zrychlení přístupu k datům.
- od nejrychlejší a nejmenší po největší a nejpomalejší:
  - on-chip L1 caches
  - off-chip L2 caches (SRAM)
  - hlavní paměť (DRAM)
  - vedlejší paměť (pevný disk)

28.2 Způsoby adresace

Máme 2 typy adres. Logickou a fyzickou adresu. Logická adresa je adresa, se kterou pracuje CPU. Tato adresa je překládána na fyzickou adresu, což je konkrétní adresa v dané paměti. Výhody - CPU vidí unifikované adresy - pro přístup do hlavní paměti i k IO zařízením. Data v hlavní paměti nejsou poskládána tak jak jdou za sebou, mohou být

zpřeházena - optimalizace využití prostoru paměti a fragmentace. Přes logickou adresu se nechá přistupovat k datům sekvenčně, tak jak jdou za sebou.

Velikost logického adresního prostoru je dána architekturou procesoru - kolik adres je procesor schopen generovat. Např. 32 bitový procesor generuje2^{32}adres. Velikost fyyického adresního prostoru je dána nainstalovaným hardware počítače - fyzicky dostupnou pamětí.

Stránkování Souvislý LAP (Logický Adresní Prostor) není zobrazován jako jediná souvislá oblat FAP. FAP se dělí na úseky zvané rámce, LAP se dělí na úseky dané stránky.

Struktura logické adresy Logická adresa se zkládá ze dvou částí. První část je index v tabulce stránek - díky němu se v tabulce stránek vyhledá fyzická adresa stránky. Druhá část je offset - posunutí ve stránce.

Převod logické adresy na fyzickou

- K logické adrese je v tabulce stránek nalezena fyzická adresa dané stránky/rámce.
- K adrese rámce je přičten offset a výsledek je požadovaná fyzická adresa.

28.3 Různá šíře logických a fyzických adres

CPU může mít menší datovou sběrnici, než je šíře fyzické adresy. Např. fyzická adresa má 20 bitů, ale datová sběrnice CPU je jen 16 bitů. Zbývající 4 bity jsou offset segmentu a jsou uloženy ve 4 segmentových registrech CPU. FA = (segment << 4)+offset.

Opačný případ - CPU generuje více logických adres (např. 64 bitový procesor generuje 2^{64}adres) než je fyzických adres (LA je větší než FA). Tento problém se řeší virtualizací paměti. Virtualizace zpočívá v tom, že FAP se rozšiřuje o úseky na vnější paměti (např. na pevném disku).

28.4 Mapování, stránkování, segmentace

Stránkování LAP je rozdělen do úseků, které na sebe navazují - stránky. Logická adresa odkazuje na adresu stránky, která odkazuje na adresu rámce FAP. Rámce v FAP na sebe nemusí navazovat.

Segmentace LAP je rozdělena na segmenty. Segmenty jsou části programu - mají logický význam (hlavní program, procedura, funkce,...), jsou různě dlouhé - nízká vnitřní fragmentace. Výhody - lze určit přístup do nepovolené části paměti (segmentation fault), se segmenty v paměti lze libovolně hýbat, lze nastavovat práva k přístupu do segmentu. Nevýhoda - externí fragmentace.

Stránkování a segmentace najednou Výše uvedené metody lze kombinovat. Segmentace vybírá části LAP, stránkování zobrazuje LAP do FAP - LAP je dělena na segmenty, které jsou stránkovány.

28.5 Přerušení a výjimky

Přerušení

- cílem je zlepšení účinnosti systému
- je potřeba provést jinou posloupnost příkazů jako reakci na nějakou „neobvyklou" událost
- přerušující událost způsobí, že se pozastaví běh procesu v CPU takovým způsobem, aby ho bylo možné později znovu obnovit, aniž by to přerušený proces „poznal"
- využití např. při IO operacích
- testování, zda je voláno přerušení alespoň po dokončení každé instrukce
- přerušení bývá často voláno programem
- Maskovatelná, lze je zakazat v stavovem řídícím slovu CPU, případně řízení priorit (periferie, čítače, časovače)
- Nemaskovatelná - ošetření HW chyb, hlídací obvod (Watch Dog)

Výjimka

- Výjimka – ošetření zvláštních situací, které brání dalšímu vykonávání instrukcí (exception)
  - Matematické přetečení (výsledek instrukce s kontrolou saturace přetekl)
  - Načtena nedefinovaná instrukce (neznámý operační kód instrukce typu IR, nebo neznámá funkce instrukce typu R)
  - Systémové volání (instrukce syscall)

Zdroje přerušení, přerušovací vektory

Určení zdroje výjimky přerušení Softwarové vyhledání (polled exception handling)

- Veškerá přerušení a výjimky spouštějí rutinu od stejné adresy – např. standardní MIPS, adresa 0x00000004
- Rutina zjistí důvod ze stavového registru (MIPS: cause registr)

Vektorová obsluha přerušení

- Již hardware CPU zjistí příčinu/číslo zdroje
- V paměti se nachází na pevné/řídicím registrem specifikované (VBR) adrese tabulka vektorů přerušení
- Procesor převede číslo zdroje na index do tabulky
- Z daného indexu načte slovo a vloží ho do PC ●

Nevektorová obsluha více pevně určených adres podle priorit/důvodu

- Často jsou přístupy kombinované, např. výjimky mají oddělené cílové adresy skoků, využívá tabulku atd. ale veškerá vnější přerušení končí pouze na jednom z vektorů

28.6 DMA přenosy

Direct Memory Access - přímý přístup do paměti. Využívá se při přenosu velkého množství dat - data nemusí jít přes procesor a nevytěsňují tak data z cache.

- Program/OS nastaví parametry přenosu
- Procesor nastaví adresy do DMA řadiče, ten na konci přenosu vyvolá přerušení

29. A4B02FYZ

5. června 2012

1 Kinematika hmotného bodu

1.1 Vztažný systém

Pohyb je relativní, a proto je nutno zavést vztažný systém (vztažnou soustavu). Se vztažným systémem spojíme pohyb tělesa. Nejznámější vztažný systém je pravoúhlý souřadný systém (kartézský).

1.2 Polohový vektor

Polohový vektor určuje polohu bodu. Jeho počáteční bod leží v počátku souřadné soustavy a jeho koncový bod splývá s polohou, kterou určuje. Velikost polohového vektoru je: $|\vec{r}| = r = \sqrt{x^2 + y^2 + z^2}$

Jednotkový polohový vektor je definován poměrem: $\vec{r_0} = \frac{\vec{r}}{|\vec{r}|}$, jeho velikost je jedna a je bezrozměrný.

1.3 Trajektorie

Množina koncových bodů polohového vektoru $\vec{r} = \vec{r}(t)$ je trajektorie. Je to křivka, po které se hmotný bod pohybuje.

1.3.1 Parametrická rovnice trajektorie

Časová závislost polohového vektoru je vektorová rovnice popisující křivku v prostoru. $\vec{r} = f(t) = [x(t); y(t); z(t)]$. Každá souřadnice vektorové funkce představuje jednu parametrickou rovnici trajektorie.

1.4 Rychlost

Okamžitá rychlost je dána změnou polohy za jednotku času. Určuje ji rovnice $\vec{v} = \frac{d\vec{r}}{dt}$, jednotka rychlosti je $\mathrm{m.s}^{-1}$.

Tato rovnice představuje tři složkové rovnice:

$v_x = \frac{dx}{dt}$, $v_y = \frac{dy}{dt}$, $v_z = \frac{dz}{dt}$

Velikost rychlosti se zjišťuje jako velikost vektoru, tedy (kde s je délka dráhy):

$v = |\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2} = \frac{\sqrt{dx^2+dy^2+dz^2}}{dt} = \frac{ds}{dt}$

1.5 Zrychlení

Okamžité zrychlení je dáno změnou vektoru rychlosti za jednotku času. Určuje ho rovnice$\vec{a} = \frac{d\vec{v}}{dt}$ Jednotka zrychlení je m.s^{-2}.

1.5.1 Tečné a normálové zrychlení

Zrychlení často rozkládáme na tečnou $\vec{a_t}$a normálovou $\vec{a_n}$složku zrychlení. Platí, že $\vec{a} = \vec{a_t} + \vec{a_n}$. Tečná složka zrychlení má směr tečny a normálová směr normály (kolmice) k trajektorii. Velikost těchto složek je $a_t = \frac{dv}{dt}$ a $a_n = \frac{v^2}{R}$, kde v je velikost rychlosti a R je poloměr "křivosti" trajektorietrajektorie, obojí v místě rozkladu vektoru zrychlení. Velikost zrychlení se zjišťuje jako velikost vektoru nebo z tečné a normálové složky zrychlení, jako $a = |a| = \sqrt{a_x^2 + a_y^2 + \vec{a_z^2}} = \sqrt{a_t^2 + a_n^2}$

1.6 Klasifikace pohybů

1.6.1 Přímočarý

Vektor v má stále stejný směr, který splývá s přímkou, po níž se hmotný bod pohybuje.

Dráhou přímočarého pohybu je přímka. Proto stačí popis v souřadné soustavě s jedinou osou x. Pohyb tedy stačí popsat veličinami $x = s$, $v_x = v$, $a_x = a$. Rychlost přímočarého pohybu odvodíme z definiční rovnice zrychleni (7), stačí ji napsat pro směr x.

$a = \frac{dv}{dt} \Rightarrow dv = adt \Rightarrow_{v_0} \int^v dv =_0 \int^t adt \Rightarrow v - v_0 =_0 \int^t adt$

Pro pohyb rovnoměrně zrychlený, splňující podmínku a = konst dále platí: Rychlost přímočarého pohybu rovnoměrně zrychleného je tedy dána rovnicí:

$v =_0 \int^t adt + v_0 = a \,_0\int^t dt + v_0 = at + v_0$

Polohu přímočarého pohybu na ose x vypočítáme, jako:

$v = \frac{dx}{dt} \Rightarrow dx = vdt \Rightarrow_{x_0} \int^x dx =_0 \int^t vdt = x - x_0 =_0 \int^t vdt$

Pro pohyb rovnoměrně zrychlený, splňující podmínku a = konst dále platí:

$x =_0 \int^t (v_0 + at)dt + x_0 = v_0 \,_0\int^t dt +_0 \int^t tdt + x_0 = x_0 + v_0 t + \frac{1}{2}at^2$, tedy $s = x = x_0 + v_0 t + \frac{1}{2}at^2$

1.6.2 Křivočarý

Vektor v r mění svůj směr, který je vždy tečný ke křivce, po níž se hmotný bod pohybuje. Speciálními křivočarými pohyby jsou kruhový pohyb a vrhy.

Ke křivočarému pohybu už musíme obecně použít vektorový popis. Bez odvození napišme, že pro popis obecného křivočarého pohybu v prostoru platí analogické rovnice jako v předchozím odstavci s tím rozdílem, že k popisu použijeme vektory. Bude tedy platit následující sestava rovnic:

Pro pohyb rovnoměrně zrychlený s konstantním zrychlením:

$\vec{v} = \vec{v_0} + \vec{at}$

$\vec{r} = \vec{r_0} + \vec{v_0}t + \frac{1}{2}\vec{a}t^2$

2 Dynymika hmotného bodu

Dynamika je část mechaniky, která se zabývá příčinami pohybu a příčinami změn pohybu.

2.1 Základní veličiny dynamiky

Základní veličiny dynamiky jsou hmotnost, hybnost a síla.

2.1.1 Hmotnost

Hmotnost je skalární veličina, která vyjadřuje míru setrvačných a gravitačních vlastností tělesa. Je to jedna ze 7 základních veličin fyziky. Základní jednotkou hmotnosti je kg.

2.1.2 Hybnost

Hybnost $\vec{p}$ je vektorová veličina, která vyjadřuje míru setrvačných účinků a míru gravitačních účinků tělesa dané hmotnosti m Hybnost závisí na hmotnosti m a rychlosti $\vec{v}$ tělesa, směr hybnosti je stejný jako směr rychlosti. Hybnost se vypočítá jako$\vec{p} = m\vec{v}$. Jednotka hybnosti je kg.m.s^{-1}.

2.1.3 Síla

Síla je vektorová fyzikální veličina, která vyjadřuje míru vzájemného působení těles. Síla má za následek buďto změnu pohybového stavu těles nebo jejich deformaci. Pokud chceme sílu definovat obecně i pro relativistickou fyziku, musíme sílu definovat jako časovou derivaci hybnosti tělesa $\vec{p}$, tedy $\vec{F} = \frac{d\vec{p}}{dt}$. V klasické mechanice (v případech, kdy lze zanedbat změnu hmotnosti při pohybu) přejde rovnice $\vec{F} = \frac{d\vec{p}}{dt}$ na tvar $\vec{F} = \frac{d\vec{p}}{dt} = \frac{d(m\vec{v})}{dt} = m\vec{a}$. Tyto vztahy představují druhý Newtonův zákon, jednotka síly je Newton N=kg.m.s^{-2}.

2.2 Newtonovy zákony

Existují tři Newtonovy pohybové zákony, které umožňují určit pohyb tělesa v inerciální vztažné soustavě, jsou-li známé síly, které působí na dané těleso.

2.2.1 první Newtonův zákon

Jestliže na těleso nepůsobí žádné vnější síly nebo výslednice sil je nulová, pak těleso setrvává v klidu nebo v rovnoměrném přímočarém pohybu.

2.2.2 druhý Newtonův zákon

Jestliže na těleso působí síla, pak se těleso pohybuje se zrychlením, které je přímo úměrné působící síle a nepřímo úměrné hmotnosti tělesa.

Což je dáno vzorcem (kde F je vektor síly m hmotnost a a vektor zrychlení):

$\vec{F} = m\vec{a}$

Obecněji se tento vztah dá zapsat, jako (kde p je vektor hybnosti, je čas a v je vektor rychlosti):

$\vec{F} = \frac{d\vec{p}}{dt} = \frac{d(m\vec{v})}{dt}$

Tento zákon lze dále použít k sestavení pohybové rovnice (kde r jevektorová funkce, která určuje polohu hmotného bodu v závislosti na čase):

$\vec{F} = m\frac{d^2\vec{r}}{dt^2}$

2.2.3 třetí Newtonův zákon

Proti každé akci vždy působí stejná reakce; jinak: vzájemná působení dvou těles jsou vždy stejně velká a míří na opačné strany.

To se dá zapsat, jako: (F_1je síla akce a F_2je síla reakce)

$F_1 = -F_2$

3 Pohybové rovnice pro inerciální a neinerciální vztažné soustavy

3.1 Inerciální vztažná soustava

Za inerciální vztažnou soustavu budeme považovat takovou, která se vzhledem ke stálici (Slunci) buď nepohybuje (v = 0), nebo se všechny její pevné body pohybují rovnoměrně přímočaře ($\vec{v}$ = konst. r). Při takovém pohybu žádný pevný bod v této soustavě nebude zakřivovat svoji trajektorii. Platí, že každá další vztažná soustava, je-li vzhledem k inerciální soustavě v klidu nebo pohybu rovnoměrném přímočarém, je rovněž inerciální. Jako příklad můžeme uvést například stěny vagonu, který se pohybuje po přímé trati stálou rychlostí. V inerciálních vztažných soustavách platí 1. Newtonův pohybový zákon - zákon setrvačnosti.

3.2 Neinerciální vztažná soustava

Všechny ostatní vztažné soustavy jsou neinerciální. V neinerciálních vztažných soustavách neplatí 1. Newtonův pohybový zákon ani 3. Newtonův pohybový zákon, tzn. že těleso, ačkoliv na ně nepůsobí žádná síla nebo výslednice sil je nulová, mění svůj pohybový stav (rychlost), tzn. pohybuje se s nenulovým zrychlením.

3.3 Posouvající se neinerciální soustava

Je to soustava, která se vzhledem k inerciální pohybuje přímočaře nerovnoměrně. Zrychlení soustavy a všech bodů na osách je stejné a nenulové. Za reprezentující bod budeme považovat počátek takové neinerciální vztažné soustavy.

Zrychlení definujeme:

$a_u = \vec{\frac{dR}{dt}}$

Pak platí, že v takto definované neinerciální soustavě vzniká setrvačná zdánlivá síla:

$\vec{F_s} = -m\vec{a_u}$

kde m je hmotnost tělesa sledovaného v neinerciální soustavě. Pokud chceme řešit pohybovou úlohu v neinerciální soustavě, musíme ke všem skutečným silám připočítat síly zdánlivé.

3.4 Rotující neinerciální soustava

Je to soustava, která vzhledem k inerciální rotuje. Je výhodné zvolit si osu rotace za osu z obou soustav, inerciální i neinerciální, jak ukazuje obr. 1. V neinerciální soustavě takového typu pak vzniknou tři zdánlivé síly.

3.4.1 Síla Eulerova

Eulerova síla je zdánlivá síla působící v rotující neinerciální soustavě, která rotuje s proměnnou úhlovou rychlostí, e $\neq 0$. Její vektorový výpočet je:

$\vec{F_E} = -m\vec{\varepsilon} \times \vec{r}$

kde $\vec{r}$ je polohový vektor tělesa o hmotnosti m, nacházejícího se v rotující neinerciální soustavě, která rotuje s úhlovým zrychlením $\vec{\varepsilon}$.

3.4.2 Síla Coriolisova

Coriolisova síla je zdánlivá síla působící na tělesa pohybující se v rotující neinerciální vztažné soustavě tak, že se mění jejich vzdálenost od osy otáčení. Coriolisova síla má směr kolmý ke spojnici těleso - osa otáčení a způsobuje stáčení trajektorie tělesa proti směru otáčení soustavy.

Její vektorový výpočet je dán rovnicí:

$\vec{F_c} = -2m\omega \times \vec{v'}$

kde m je hmotnost tělesa, $\vec{v}$ je rychlost tělesa v neinerciální vztažné soustavě, $\vec{\omega}$ je vektor úhlové rychlosti otáčení neinerciální soustavy.

Velikost Coriolisovy síly spočteme jako:

$F_c = -2m\omega v' sin\alpha$

αje úhel sevřený mezi vektorem úhlové rychlosti a vektorem rychlosti.

3.4.3 Síla odstředivá

Odstředivá síla je jedna ze zdánlivých sil, které působí na těleso v otáčející se neinerciální vztažné soustavě. V inerciálních vztažných soustavách odstředivé síly nepůsobí. Důsledkem odstředivé síly je odstředivé zrychlení. Odstředivá síla je dána rovnicí:

$\vec{F_o} = -m\vec{\omega} \times \vec{\omega} \times \vec{r}$

Kde m je hmotnost tělesa, $\vec{\omega}$ je vektor úhlové rychlosti otáčející se neinerciální soustavy a $\vec{r}$ je polohový vektor tělesa, jehož počátek leží na ose rotace.

Velikost odstředivé síly je:

$F_o = m\omega^2 r$

4 Práce a Energie

4.1 Energie

Je schopnost vykonat práci. Abychom mohli vykonat práci, musíme mít energii. Celková mechanická energie objektu je součet jeho kinetické a potenciální energie. Jednotkou energie v soustavě SI je joule (J) = $\mathrm{kgm^2\ s^{-2}}$ (1 Joule je definován jako práce, kterou koná síla 1 N působící po dráze 1 m.)

4.1.1 Kinetická energie

Kinetická energie je energie pohybová. Vyjadřuje skutečnost, že pohybující se těleso je schopné konat práci jako důsledek svého pohybu, např. nárazem na okolní objekt. Kinetická energie hmotného bodu, těles zanedbatelných rozměrů nebo těles pohybujících se bez rotace (takový pohyb se nazývá translační nebo posuvný) je definována vztahem:

$E_k = \frac{1}{2}mv^2$

4.1.2 Potenciální energie

Potenciální energie má svoji podstatu v poloze nebo konfiguraci. Ne každý objekt je však schopen vykonat práci v důsledku své polohy. V gravitačním poli Země se potencionální energie spočítá, jako:

$E_p = mgh$

4.1.3 Zákon zachování mechanické energie

Jestliže těleso nebo hmotný systém nepodléhají účinkům okolí, pak součet kinetické a potenciální energie částic, z nichž se skládá, zůstává stálý. To znamená, že v soustavě se může měnit jeden druh energie v druhý.

$E = E_p + E_k = konst.$

4.2 Práce

Působení síly na fyzikální těleso nebo na silové pole, při kterém dochází k posouvání nebo deformaci tohoto tělesa resp. ke změně rozložení potenciální energie v silovém poli.

4.2.1 Posuvný pohyb

Práce je definována následujícím vztahem:

$W = \vec{F} \cdot \vec{s} = F \cdot s \cdot cos\alpha$

Kde α je úhel mezi silou a trajektorií pohybu.

Pokud je dráha zakřivena nebo je síla proměnná, použijeme pro výpočet integrál tzv. elementárních prací:

$dW = \vec{F} \cdot d\vec{s}$, tedy $W =_0 \int^s \vec{F} \cdot d\vec{s} =_0 \int^s (F \cdot cos\alpha) d\vec{s}$

4.2.2 Otáčivý pohyb

Mechanická práce závisí na momentu síly, který na těleso působí, na úhlu, o který se těleso otočí, a na úhlu, který svírá vektor momentu síly a osa otáčení tělesa.

Otočí-li se těleso kolem neměnné osy otáčení působením konstantního momentu síly M rovnoběžného s osou otáčení tělesa o úhel α, pak lze velikost práce zapsat ve tvaru:

$W = M \cdot \alpha$

Pokud je moment síly proměnný, použijeme pro výpočet integrál tzv. elementárních prací:

$dW = \vec{M} \cdot d\vec{\alpha}$, tedy $W =_0 \int^s \vec{M} \cdot d\vec{\alpha}$

4.2.3 Moment síly

Moment síly je vektorová fyzikální veličina, která vyjadřuje míru otáčivého účinku síly. Otáčivý účinek síly se vztahuje vzhledem k danému bodu nebo přímce. Bod, ke kterému se moment síly určuje, se nazývá momentovým bodem. Kolmá vzdálenost p síly od její osy k bodu je tzv. rameno síly.

Moment síly je definován jako součin síly a kolmé vzdálenosti osy síly od daného bodu. Velikost momentu síly tedy závisí na velikosti síly a na vzdálenosti od osy otáčení (čím dále, tím větší moment síly).

4.2.4 Konzervativní silové pole

Konzervativní silové pole je silové pole, které může konat práci, ale v izolovaném systému na uzavřené křivce je celková vykonaná práce nulová. Konzervativní síly lze vyjádřit jako záporný gradient potenciální energie: F = -∇ V, proto se též nazývají potenciálové. Mezi konzervativní síly patří např. gravitační síla a elektrostatická síla.

4.2.5 Moment hybnosti

Moment hybnosti je vektorová fyzikální veličina, která popisuje rotační pohyb tělesa. Moment hybnosti má při rotačním pohybu stejný význam jako hybnost při pohybu přímočarém. Pojem momentu hybnosti je analogický pojmu hybnosti: tak jako je hybnost součinem hmotnosti a rychlosti v případě translačního pohybu, tak je moment hybnosti součinem momentu setrvačnosti a úhlové rychlosti v případě rotačního pohybu.

Moment hybnosti L je určen vektorovým součinem jako

L = r * p;

kde r je polohový vektor a p je hybnost.

Jednotka SI: kilogram krát metr na druhou za sekundu, značka jednotky: kg.m2.s-1

4.2.6 I. impulsová věta

Časová změna celkového momentu hybnosti je rovna výslednici všech vnějších sil, které na soustavu působí.

$$\dot{\vec{p}} = \sum_{k=1}^{n} \vec{F}_k$$

To také znamená, že vnitřní síly nemohou změnit pohybový stav soustavy jako celku. Je-li výslednice všech vnějších sil nulová, nemění se výsledná hybnost soustavy. V tomto případě se výsledná hybnost zachovává.

4.2.7 II. impulsová věta

Časová změna výsledného momentu hybnosti je rovna celkovému momentu vnějších sil.

$$\frac{d\vec{L}}{dt} = \sum_{k=1}^{n} \vec{r}_k \times \vec{F}_k$$

Je-li výsledný moment vnějších sil nulový, zachovává se celkový moment hybnosti soustavy.

5 Otáčivý pohyb tuhého tělesa

Rotace je takový pohyb tuhého tělesa, při kterém se všechny body tělesa otáčejí kolem jedné společné osy otáčení se stejnou úhlovou rychlostí. Trajektoriemi jednotlivých bodů tělesa jsou soustředné kružnice. Úhlové rychlosti a úhlová zrychlení jednotlivých bodů tělesa jsou při otáčivém pohybu stejné.

5.1 Osa otáčení

Osa otáčení je přímka, kolem které se těleso při otáčivém pohybu otáčí. Body tělesa, které na ose leží, zůstávají na svých místech, jejich rychlost je nulová.

5.2 Moment setrvačnosti

Moment setrvačnosti je fyzikální veličina, která vyjadřuje míru setrvačnosti tělesa při otáčivém pohybu. Její velikost závisí na rozložení hmoty v tělese vzhledem k ose otáčení. Body (části) tělesa s větší hmotností a umístěné dál od osy mají větší moment setrvačnosti.

Moment setrvačnosti je definován vztahem:

$J =_v \int \rho r^2 dV$

Jeho jednotka je kg.m^2

Je-li tuhé těles homogení, tedy je-li jeho hustota $\rho = konst.$, pak lze moment definovat následovně:

$J = \rho_v \int r^2 dV$

5.2.1 Steinerova věta (Steinerův doplněk)

Pro moment setrvačnosti tuhého tělesa vzhledem k ose otáčení jdoucí mimo těžiště tělesa platí Steinerova věta:

$J = J_T + md^2$

Kde J_T je moment setrvačnosti vzhledem k ose jdoucí těžištěm tělesa, m je hmotnost tělesa a d je kolmá vzdálenost těžiště od osy otáčení.

5.3 Kinetická energie při otáčivém pohybu tělesa

Kinetická energie E_k tuhého tělesa při otáčivém pohybu je rovna $E_k = \frac{1}{2}J\omega^2$

5.4 Pohybová rovnice při otáčení tuhého tělesa

Při otáčení tuhého tělesa lze odvodit jeho pohybovou rovnici ve tvaru:

$\vec{M} = J \cdot \vec{\varepsilon}$

6 Gravitační pole a příklady jeho působení

6.1 Newtonův Gravitační zákon

Newtonův gravitační zákon je zákon, který je použitelný pouze pro slabá gravitační pole. Je formulován tak, že každá dvě tělesa o hmotnostech m_1a $m_2$na sebe působí silou přímo úměrnou hmotnostem obou těles a silou nepřímo úměrnou čtverci jejich vzdáleností, tedy:

$F_g = G\frac{m_1 m_2}{r^2}$

Kde G je gravitační konstanta G=6,67·$10^{-11} m^3 kg^{-1} s^{-2}$

6.2 Intenzita Gravitačního pole

V okolí každého tělesa existuje gravitační pole, které působí na jiná tělesa. Pro porovnání silového působení v různých místech gravitačního pole je zavedena intenzita grav. pole, která je definována následujícím vztahem:

$\vec{K} = \frac{\vec{F_g}}{m}$

Její jednotkou je N.kg^{-1} (kde \mathbf{F}_g je gravitační síla a m je hmotnost hmotného bodu, na nějž těleso s intenzitou gravitačního pole \mathbf{K} působí.)

Velikost intenzity gravitačního pole $\vec{K}$ v daném místě pole určíme ze vztahu pro velikost gravitační síly vyjádřenou v gravitačním zákonu.

$K = G\frac{M}{r^2}$

6.3 Potenciál Gravitačního pole

Gravitační potenciál hmotného bodu je v newtonovské fyzice vyjádřen vzorcem:
$V = -G\frac{M}{r}$

7 Mechanické kmitavé soustavy

Těleso připevněné k pružině, která je upevněná k boční pevné stěně, těleso se pohybuje ve vodorovném směru bez tření, zvolený směr pohybu např. ve směru souřadnicové osy x. Pokud je těleso vychýleno z rovnovážné polohy, pak na něj působí síla F.

F = -k x

Kde k je tuhost pružiny.

Na základě druhého Newtonova zákona je možné sestavit pohybovou rovnici:

ma = -k x

$\frac{d^2x}{dt^2} + \frac{k}{m}x = 0$

Je možné zavédst $\omega_0^2 = \frac{k}{m}$, následně můžeme přepsat rovnici do tvaru:

$\frac{d^2x}{dt^2} + \omega_0^2 x = 0$

8 Lineární harmonický oscilátor

Přes diferenciální rovnice se dojde ke vzorečku:

$u(t) = u_0 sin(\omega_0 t + \psi)$

$u_0$je amplituda harmonických kmitů, $\omega_0$je úhlová rychlost ψje fázová konstanta.

Rychlost a zrychlení kmitajícího hmotného bodu (tělesa) určíme na základě následujících vztahů:

$v(t) = \frac{du(t)}{dt} = u_0\omega_0 cos(\omega_0 t + \psi)$

$a(t) = \frac{dv(t)}{dt} = \frac{d^2u}{dt^2} = -u_0\omega_0^2 sin(\omega_0 t + \psi)$

8.1 Tlumené kmitání

Řešení je přes diferenciální rovnice

8.1.1 Chování systému při tlumených kmitech

V závilosti na velikosti tlumení ζlze rozlišit 3 situace:

Proζ<1 systém bude oscilovat okolo rovnovážné polohy, ale amplituda bude s časem klesat. Pro úhlovou frekvenci kmitů platí vztah:

$\omega = \omega_0\sqrt{1-\zeta^2}$

Pro ζ=1 nastane kritické tlumení, průběh oscilací je popsán rovnicí:

$x(t) = \frac{1}{2}((x(0) + \frac{\dot{x_0}}{\omega_0}e^{\omega_0 t} + (x(0) - \frac{\dot{x_0}}{\omega_0})e^{-\omega_0 t})$

Pro ζ>1 Komplikovaný průběh, jde o velmi velké tlumení a oscilace proto nelze pozorovat

Otázka č.30 - A4B02FYZ - Ondřej Smrž

Dynamický systém

- je systém, který se v čase mění v závislosti na množině pravidel, která určuje jak se jeden stav systému změní do druhého stavu.
- skládá se ze dvou částí
  - *stavového vektoru*, který popisuje stav ve kterém se právě systém nachází
  - *funkce*, která nám určuje jak se systém dostane z jednoho stavu do druhého

Klasifikace dynamického systému

- Lineární - funkce popisující lineární systém musí splňovat aditivitu a homogenitu $f(x+y)=f(x)+f(y)$ a $f(\alpha x)=\alpha f(x)$
- Nelineární - funkce nesplňují předchozí pravidla
- Autonomní - systém běžných diferenciálních rovnic, které nezávisí na nezávislé proměnné. Pokud je tato proměnná čas, systém se nazývá *time-invariant system*

$$x_d(t) = x(t+\delta)$$
$$y(t) = t\,x_d(t)$$
$$y_1(t) = t\,x_d(t) = t\,x(t+\delta)$$

$$y(t) = t\,x_d(t)$$
$$y_2(t) = y(t+\delta) = (t+\delta)x(t+\delta)$$

$y_1(t) \neq y_2(t)$ tzn. systém je neautonomní nebo není time-invariantní

- Neautonomní
- Diskrétní - popsán rovnicí nebo množinou rovnic. Pokud popsán pouze jednou rovnicí mluvíme o jednodimenzionální mapě. Typicky je systém popsán následovně, kde k je čas

$$x(0) = x_0$$
$$x(k+1) = f(x(k))$$
$$x(k) = f^k(x_0)$$

$$x(0) = x_0$$
$$x' = f(x)$$

- Continuous system - popsán jednou nebo více diferenciálníma rovnicema

$$x(k+1) = ax(k) + b$$
$$x'(t) = ax(t) + b$$

- jednodimenzionální - popsán jednou funkcí

$$\vec{x}(k+1) = \mathbf{A}\vec{x}(k) + \vec{\mathbf{B}}$$
$$\vec{x}'(t) = \mathbf{A}\vec{x}(t) + \vec{\mathbf{B}}$$

- vícedimenzionální - popsán vektorem funkcí

  Pojmy: diagonální matice,počítání vlastních čísel, vektorů, jakobian(matice parciálních derivací funkcí), stopa matice(součet prvků na hlavní diagonále),...

Fázový portrét

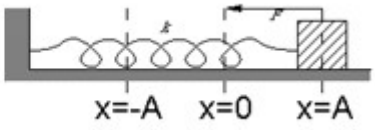
phase space(fázový prostor) - je prostor ve kterém jsou reprezentovány všechny možné stavy systému, kde každy stav odpovídá unikatnímu bodu fázového prostoru

- fázový prostor dvou dimenzionálního systému se nazývá phase plane(fázová rovina)

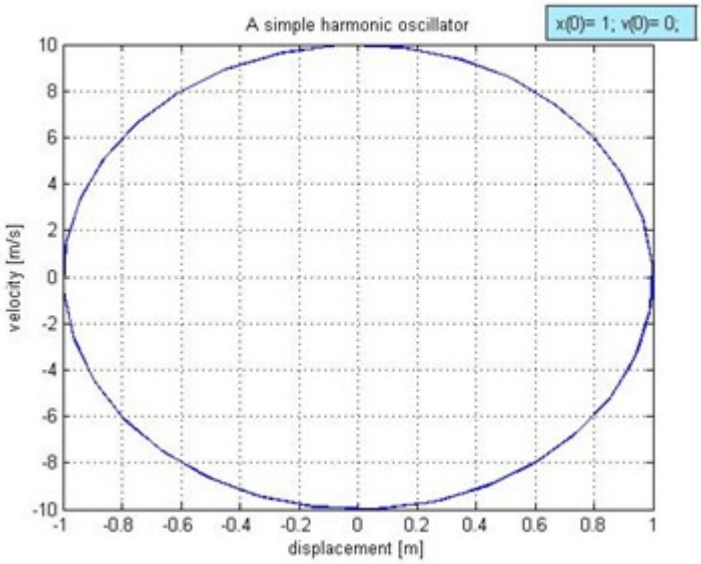
- v mechanických systémech se většinou fázový prostor skládá z proměnných vyjadřující rychlost a polohu

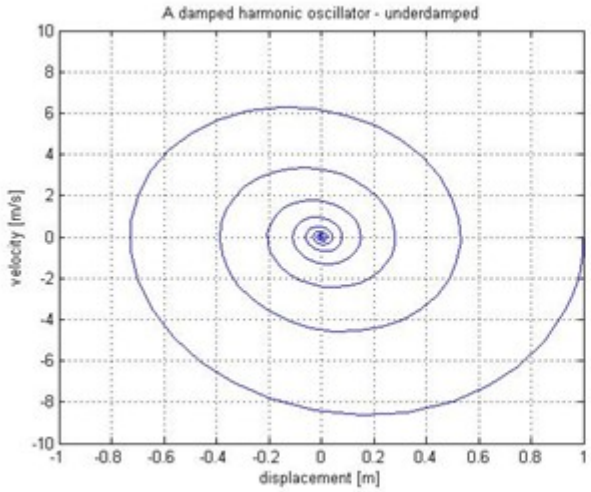
fázový portrét - graf jedné fázové křivky(nebo více) odpovídající rozdílným počátečním podmínkám dané fázové roviny

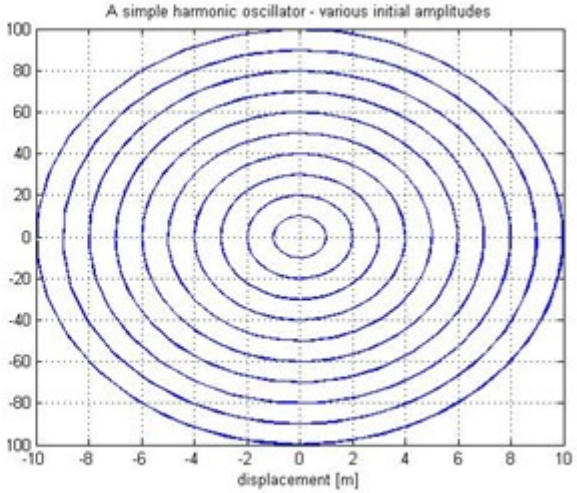
$$\ddot{x} + \omega^2 x = 0$$



počáteční vychýlení x = 1 metr a rychlost 0







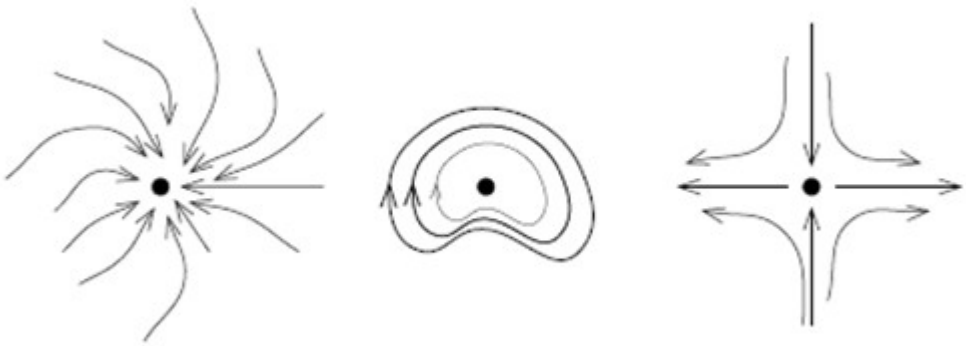
underdamped oscilátor vypadá jako spirála a jednoduchý harmonický oscilátor jako soustředné kruhy ;)

Stabilita a stacionární body

- stacionární bod - speciální bod dynamického systému, který se nemění v závislosti na čase (pro hledání položíme první derivaci rovnou nule)
- atraktor - je množina(bod, křivka,..) ke které se systém v čase vyvíjí
- stabilní stacionární bod - pro každé počáteční hodnoty systém konverguje k tomuto bodu

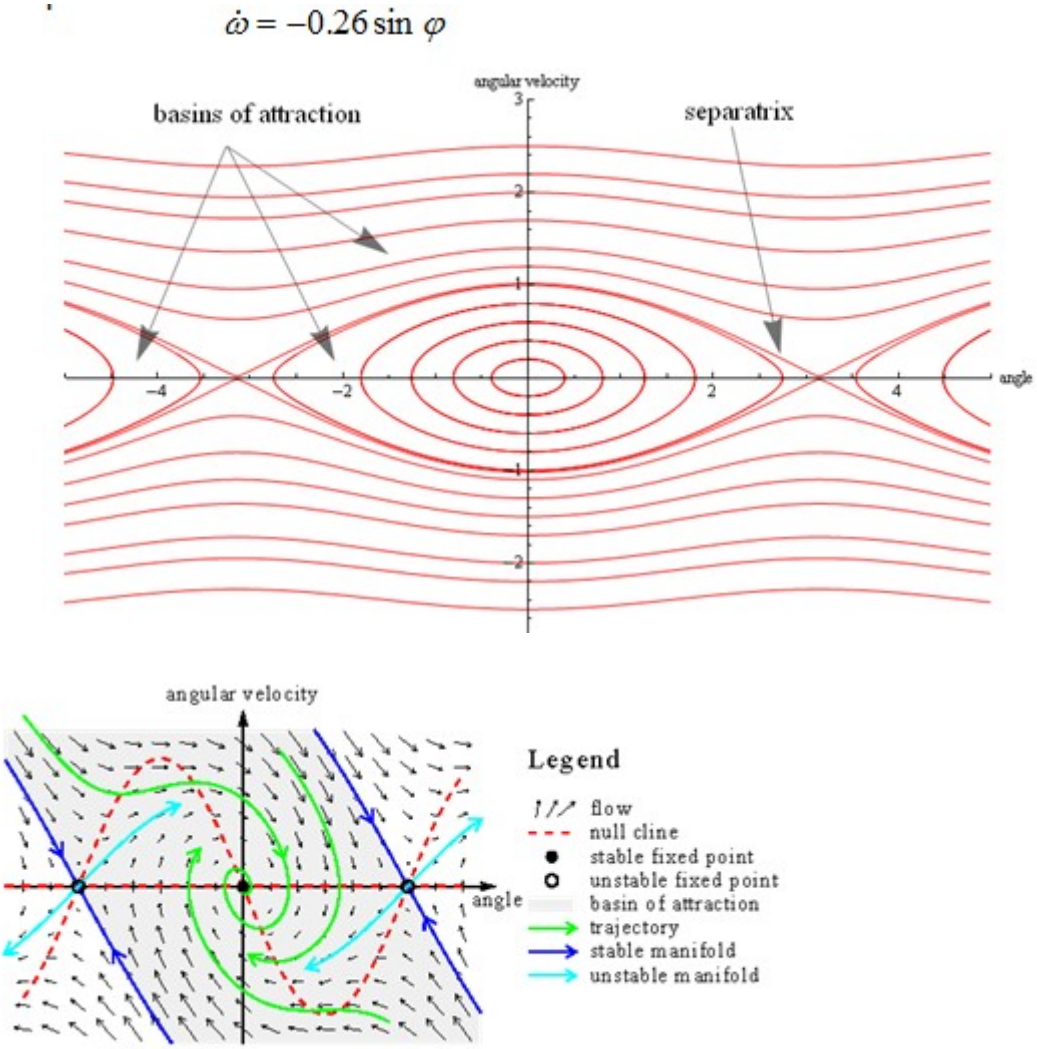
- částečně stabilní stacionární bod - pro počáteční hodnoty dostatečně blízko zůstává systém v okolí část.stab.bodu ale nekonverguje k němu
- nestabilní bod - pro počáteční hodnoty dostatečně blízko se systém od nestabilního bodu vzdaluje

stabilní bod, částečně stabilní a nestabilní bod



Topologická klasifikace

- basin of attraction - oblast v fázovém prostoru všech počátečních podmínek, které tíhnou k částečným řešením jako mezní cyklus, stabilní bod a jiné atraktory
- trajektorie - je řešení rovnic pohybu. Křivka ve fázovém prostoru parametrizovaná časem
- flow - výraz pro trajektorii nebo více trajektorií v dynamickém systému. Např. pohyb proměnných v čase
- nullclines - přímky kde derivace podle času jedné komponenty je rovna 0
- separatrix - hranice která odděluje dvě různá chování dynamického systému. V 2D například křivka oddělující dvě basin of attraction



Klasifikace - jednodimenzionální linearání systém

Time	Derivative at x~	Fixed point is
Continous	f'(x~)<0	Stable
	f'(x~)>0	Unstable
	f'(x~)=0	Cannot decide
Discrete	|f'(x~)|<1	Stable
	|f'(x~)|>1	Unstable
	|f'(x~)|=1	Cannot decide

Bacteria equation $\frac{dx}{dt} = bx - px^2; \quad f(x) = bx - px^2$

Derivative $f'(x) = b - 2px$

1st fixed point - unstable $\tilde{x}_1 = 0; \quad f'(\tilde{x}_1) = b > 0$

2nd fixed point - stable $\tilde{x}_2 = \frac{b}{p}; \quad f'(\tilde{x}_2) = b - 2p\frac{b}{p} = -b < 0$

Klasifikace - dvoudimenzionální linearání systém

Set of equations for 2D system

$$x_1 = f_1(x_1, x_2) = ax_1 + bx_2$$
$$x_2 = f_2(x_1, x_2) = cx_1 + dx_2$$

Jacobian matrix for 2D system

$$\mathbf{J} = \mathbf{A} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Calculation of eigenvalues

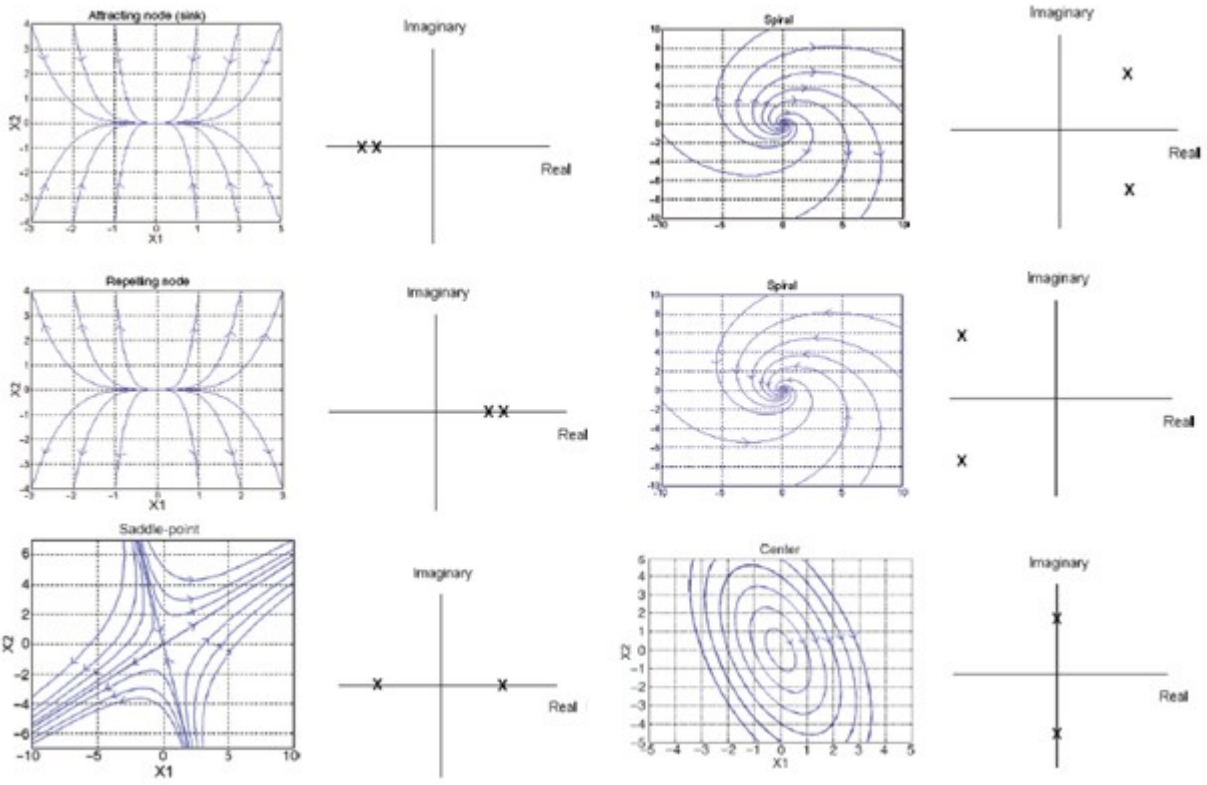
$$\det(\mathbf{A} - \lambda\mathbf{E}) = 0 \quad \Rightarrow \quad \det\begin{bmatrix} a-\lambda & b \\ c & d-\lambda \end{bmatrix} = 0$$
$$(a-\lambda)(d-\lambda) - bc = 0 \quad \Rightarrow \quad \lambda^2 - (a+d)\lambda + ad - bc = 0$$

Formulation using trace and determinant

$$Tr(\mathbf{A}) = a + d; \quad Det(\mathbf{A}) = ad - bc$$
$$\lambda^2 - Tr(\mathbf{A})\lambda + Det(\mathbf{A}) = 0$$
$$\lambda_{12} = \frac{Tr(\mathbf{A}) \pm \sqrt{Tr(\mathbf{A})^2 - 4Det(\mathbf{A})}}{2}$$

STRUČNÁ KLASIFIKACE DVOUDIMENZIONÁLNÍCH SYSTÉMŮ PODLE VLASTNÍCH ČÍSEL

např. atracting node má obě vlastní čísla reálná záporná



pro zvídavé(=šprty) další speciální případy lze najít ve slidech(2.) na stránce 19-21. Klasifikace vícedimenzionálních systémů je poslední slide.

STABILITA NELINEÁRNÍHO SYSTÉMU

metoda linearizace = první Lyapunova metoda - hlavní myšlenka je rozepsat pravou stranu funkce v rovnici pohybu do Taylerova polynomu a zanedbat tak vyšší členy

pokud metoda linearizace nedokáže rozhoudnout musíme použít d*ruhou Lyapunovu metodu =*

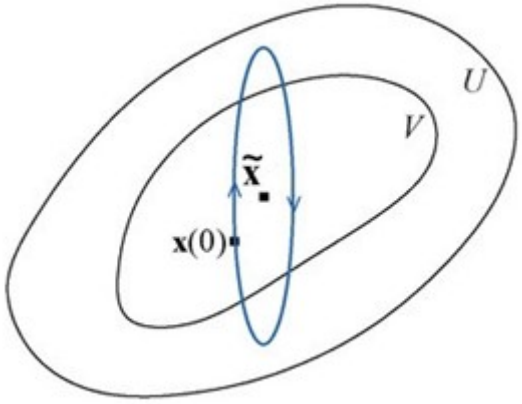
Lyapunovy funkce - příklad harmonického oscilátoru. Pokud se na harmonický tlumený oscilátor podíváme z pohledu energie. Je jasné, že v určitou chvíli se musí ustáli v ekvilibriu (bod 0,0). Princip druhé Lyapunovy metody je najít takovou funkci $V(\overrightarrow{x})$, která představuje energii a splně následující podmínky:

1. funkce je souvisle diferencovatelné v okolí stacionárního bodu
2. *pozitivně definitní* V - $V(\overrightarrow{x}) > 0$ pro všechna $\overrightarrow{x} \neq \overrightarrow{\widetilde{x}}$ a $V(\overrightarrow{\widetilde{x}}) = 0$
3. *negativně definitní* dV/dt - $\frac{dV(\overrightarrow{x})}{dt} < 0$ ve všech stavech $\overrightarrow{x}$; $\frac{dV(\overrightarrow{\widetilde{x}})}{dt} = 0$

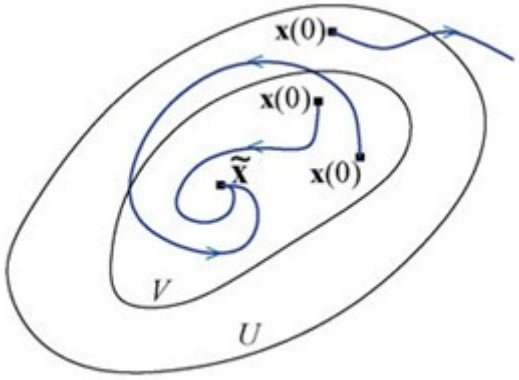
pokud lze najít takovou funkci pak je stacionární bod $\widetilde{x}$ stabilní

VYŠETŘOVÁNÍ STABILITY NELINEÁRNÍHO SYSTÉMU

- Lyapunova stabilita - stabilní bod $\widetilde{x}$je stabilní ekvilibrium pokud pro každé okolí U bodu $\widetilde{x}$ existuje takové okolí V podmnožina U, že každé řešení x(t) začínající ve V zůstane v U po celou dobu.



- Asymptoticky stabilní- stabilní bod $\widetilde{x}$ je asypmtoticky stavilní pokud je Lyapunově stabilní a další V může být vybráno tak, že $|x(t) - \widetilde{x}| \to 0$ pro $t \to \infty$ pro všechna $x(0) \in V$



- Exponencionální stabilita - pokud existuje okolí V bodu $\widetilde{x}$a a>0 že

$$\left|x(t)-\widetilde{x}\right| < e^{-at} \quad as \quad t \to \infty \quad for\ all \quad x(0) \in V$$

BIFURCATION(=?ROZDĚLENÍ?)

Rozlišujeme dvě základní:

- *global bifurcation* - occurs when larger invariant sets of the system 'collide' with each other, or with equilibria of the system. They cannot be detected purely by a stability analysis of the equilibria (fixed points). The global bifurcation includes, for example, collision of a limit cycle with a saddle point, collision of a limit cycle with a node etc

- *local bifurcation* - is a sudden change in the number or nature of the fixed and periodic points caused by a parameter change in the system. Fixed points may appear or disappear, change their stability, or even break apart into periodic points. Such bifurcation can be analysed entirely through changes in the local stability properties of equilibria, periodic orbits or other invariant sets.

Existuje 5 typů bifurcations - sadle node bif., periodic b., pitchfork b., transcritical b., hopf b.

pro zájemce více slidy 3 strana 24 a dál(popřípadě můžu doplnit, all přijde mi to zbytečné..)

#31 APO-1

Petr Svec

Architektura pocitace

Klasickty von Neumanovsky pocitac:

- Procesor
  - Radic
  - ALU (Arithetic-Logical Unit)
- Pamet
- I/O System

Radic se dal sestava z casti datove a vlastni ridici casti. Datovou lze navic rozdelit na registry a dalsi obvody.
Dulezite registry:

- Program Counter (PC)
- Instruction Register (IR)
- Stack Pointer (SP)

Zakladni cyklus pocitace (predchazi mu inicializace registru apod.):

1. Cteni instrukce
   - PC → adresa Hlavni Pameti (HP)
   - Cteni obsahu
   - Prectena data se ulozi do IR
   - Inkrementace PC o delku intrukce
2. precteni opcode(kod instrukce - typicky nekolik prvnich bitu v zavilsti na velikosti instrukcni sady)
3. provedeni instrukce - zahrnuje i dalsi veci jako je cteni operandu
4. Pokud doslo k preruseni, tak se zpracuje.
5. skok na bod 1.

Preruseni jsou dva druhy a u obou dochazi k preruseni sekvence vykonavaneho kodu a prechazi se na obsluhu.

- **Vnejsi preruseni** - asynchroni obsluha vnejsi udalosti, napr. reakce OS na event ze vstupni periferie.
- **Vnitrni preruseni**(Vyjimka) - primo od CPU, ktere takto reaguje na problem pri zpracovani instrukce napr. deleni nulou
- **Synchronni soft. preruseni** - zamerne preruseni vznikle vlozenim patricne instrukce na misto v kodu.

Kazda instrukce se sklada z opcode a operandu. Operandy muzou byt registry(resp. jejich "indexy"), adresa (napr. pro skoky)

RISC/CISC

- **RISC**(Reduced Instruction Set Computer) je architektura CPU, ve ktere je velice omezena intrukcni sada (IS) napr. jen na scitani, bitove posuny, nejake cteni/ukladani atd. Zbytek se realizuje softwarove. Vsechny instrukce maji pevny format a delku a take stejnou dobu vykonani. Velikost IS taky implikuje potrebu mnohem mensiho poctu tranzistoru, coz je mimo jine duvod proc treba ARMy moc nezerou. Zastupci: ARM, MIPS, PowerPC
- **CISC**(Complex Instruction Set Computer) je architektura (prozatim) vetsiny uzivatelskych CPU. Instrukcni sada byva casto pomerne velka, instrukce nemusi byt stejne dlouhe a jejich vykonani muze trvat ruzne dlouho. Zastupci: x86, amd64

Site procesoru a paralelni architektury

existuji 4 typy struktur procesoru:

1. **SISD** - Klasicky procesor tj. jednoduchy tok instrukci i dat
2. **SIMD** - Procesorove pole - jed. tok instrukci, vicenasobny tok dat. napr. GPU
3. **MISD** - Piplene
4. **MIMD** - Multiprocesorove pole

Propojovaci site
Zajistuji propojeni a komunikaci mezi procesory, deli se na staticke a dynamicke, pricemz ve statickych jsou spoje nemenne a v dynamickych naopak. U dynamickych jsou spojovaci spinace rizeny bud lokalne, kdy ma kazda skupina svuj radic, nebo centralne, kdy existuje jen jeden.
Staticke prop. site jsou: linearni, stromova, kruhova, mrizova, hvezdicova, krychlova a polygonalni(uplny graf)

- procesory mohou byt skalarni nebo vektorove
- skalarni procesory jsou bezne procesory
- vektorove p. provadi jednu operaci nad nekolika operandy → odstranuje se tim rezije spojena indexovanim jednolivych prvku vektoru.

Paralelizovane SISD

- samotne SISD je vetsina pocitacu zalozena na von Neumanove architekture
- palarelizovane SISD jsou systemy postavene na architekture VLIW(= very long instruction word), zalohovane systemy a systemy pouzivajici pipelining

Architektura VLIW

- velice dlouhe instrukce, ktere jsou rozdelene na nekolik casti/useku, ktere jsou zpracovavany paralelne
- jednotlive exekucni jednotky(EJ) jsou propojeny
- operacni pamet je rozdelena podle usporadani EJ tj. kazdy blok pameti odpovida jedne EJ

Zalohovane systemy - systemy, ve kterych bezi paralelne nekolik SISD tj. vsechny provadi stejny vypocet nad stejnou sadou dat ,pricemz vysledek ze vsech jednotlivych procesoru/pocitacu je porovnavan v komparatoru. Cilem je zvyseni spolehlivosti a bezpecnosti.

Paralelni systemy SIMD
SIMD systemy jsou ty, ve kterych se zpracovavaji dobre rozdelitelna data. Idealnim prikladem

jsou matice, ktere jsou takovym gró tohoto oboru, jelikoz se s nimi numericky pocitaji diferencialni rovnice, pouzivaji se k reprezentaci obrazu apod. Konkretni aplikace z toho plynouci: pocitani aerodynamiky, meteorologie, temer cokoliv co se tyka zpracovani obrazu.

Princip prace spociva v soucasnem zpracovani nekolika prvku napr. nekolik prvku z pole, kdy procesory z procesoroveho pole(PP) provadeji synchronne stejnou operaci(instrukci). Procesory jsou rizene spolecnym radicem.

1. SIMD s lokalni pameti
   - PP je rizeno univerzalnim pocitacem/procesorem, ktery zpracovava nadrizeny program → rozhoduje o maticovych ulohach a zabezpecuje prenos dat na procesory v poli
   - radic PP sam zpracovava skalarni a ridici instrukce, zatimco vektorove nechava zpracovat PP
   - kazdy procesor ma svou vlastni pamet operandu
   - procesory si mezi sebou posilaji data pres propojovaci sit
2. SIMD se sdilenou pameti
   - narozdil od SIMD s lok. pameti je v tomto pripade pamet od procesoru oddelena a komunikace probiha pres propojovaci sit
   - pridelovani pameti do procesoru zajistuje radic
   - pocet pametovych modulu muze byt jiny nez pocet procesoru

Paralelni systemy MIMD
- kazdy procesor zpracovava instrukce a data sveho vlasntiho programu
- deli se na tesne vazane a volne vazane

1. tesne vazane
   - kazdy procesor ma malou vyrovnavaci pamet pro data
   - procesory maji sdilenou operacni pamet, ktera je oddelena od procesoru a pripojena spolu s periferiemi na propojovaci sit, pres kterou komunikuji s CPU
   - periferie maji malou autonomii
   - propojovaci sit umoznuje lib. propojeni
2. volne vazane
   - procesory maji vlastni (lokalni)pamet a vlastni periferie, pricemz lokalni pamet obsahuje jak program, tak data
   - propojovaci sit byva staticka
     - Hierarchicka organizace sbernic - procesory nejnizsi urovne spolu s pametmi seskupeny do clusteru, ktere jsou pripojeny komunikacnimi moduly na sbernice vyssi hierarchie
     - Organizace do n-rozmerne krychle(nebo mrize) - kazdy proc. modul ma ma 8(4) komunikacnich procesoru pro pripojeni. Casti krychle lze dynamicky pridelovat pro ruzne ulohy. Je vyzadovan nadrizeny pocitac.
   - rizeni je slozitejsi nez u volne vazanych, ale na druhou stranu jsou tyto systemy odolnejsi vuci porucham a vypocty lze navic znekolikanasobit podle potreby. Jsou pouzivany tam, kde je treba vysoka spolehlivost

NUMA(Non-Uniform Memory Access)
- pouziva se u MIMD systemu, kde ma zajistit kratsi cekani na zapis nebo cteni do pameti tim, ze kazdemu procesoru je poskytnuta samostatna pamet
- pouziva tesnejsi vazbu vice CPU v uzlu, ktere jsou dale propojene do vetsich celku

Hierarchie pameti

- duvod hierarchizace - rychlejsi pamet je drazsi, hierarchizaci dostaneme system rychlosti se blizici tomu, ktery by pouzival vyhradne tu nejrychlejsi z pameti z hierarchie.
- hlavni myslenka - bezici programy pouzivaji v dany okamzik ke svemu behu jen cast adresoveho prostoru Kterou cast bude program pouzivat se rozhoduje podle nasledujicich dvou faktoru:

- Casova lokalita - co se pouzilo nedavno se brzo pouzije znova (soft. cykly, promenne)
- Prostorova lokalita - polozky blizko aktualne pouzivanych budou brzo treba (sekvenci provadeni kodu)

- pametova hierarchie se bezne sklada (smerem od te nejblize procesoru) z nekolika cache (L1,L2 a cim dal casteji i L3), operacni pameti a pevneho disku.
- data v pameti se hledaji smerem od CPU

Virtualizace pameti

- virtualni pamet (VP) je uroven abstrakce postavena nad vsemi zdroji dostupne pameti
- umoznuje procesu/programu dotazovat se pouze na logicke adresy (LA) a nezabyvat se jestli to je napr. v RAM nebo na HDD. VP prostor tedy muze byt mnohem vetsi nez je velikost fyzicke pameti.
- prevod mezi VA a fyzickou adresou casto zajistuje procesor (hardwarove)
- bezne implementovano pomoci strankovani, kde je strankovana jak operacni pamet, tak misto na HDD
- jednotka LA prostoru je stranka u fyzickeho adr. prostoru (FAP) to je ramec -kazdemu procesu je prideleno urcite mnozstvi stranek. ty si kazdy proces drzi ve sve Tabulce stranek
- nekolik prvnich bitu logicke adresy je indexem do tabulky stranek, na kterem lezi hledana fyzicka adresa (FA). Zbytek tvori offset jak ve strance, tak v ramci (za predpokladu, ze jsou stejne velke). Spolu s FA je na stejnem indexu take nekolik priznaku - validity bit (pritomnost stranky ve FAP), dirty(obsah modifikovan), access rights
- v pripade, ze je ramec prazdny, tak se hledana stranka pomoci DMA nacte do onoho ramce.
- pokud je nedostatek pameti, tak se odebere nejaky (nejake) ramec (ramce) na zaklade nektereho z algoritmu na vyber "obeti" (nejspise LRU - Last recently used). V pripade, ze byt aktivni dirty bit, tak se puvodni data nejprve zapisi na disk. Nakonec se aktualizuje page table.

Prerusovaci a I/O podsystem

I/O podsystem
Druhy periferii:

1. vystupni
2. vstupni
3. obousmerne - napr. HDD

Metody prenosu z/na periferie:

1. Programovy kanal - pooling(= cekani ve smycce), nejhloupejsi reseni
2. Programovy kanal s prerusenim. IO operace je zahajena na zadost programu pomoci OS. Existuji dve varianty:
   - synchronni - program ceka na dokonceni IO operace
   - asynchronni - p. neceka na dokonceni IO a muze bezet soubezne s ni. Jakmile jsou data dostupna perif. vyvola preruseni a dojde k jejich. precteni

3. Direct memory access (DMA)
4. Autonomni kanal

DMA
- prenos je realizovany specialni jednotkou bez prime ucasti OS. Ten jen nastavi parametry prenosu = kolik bytu a do jakeho bufferu a na jake adrese. Nasledne radic periferie inicializuje DMA prenos, ktery probiha dokud neni precten pozadovany pocet bytu definovany OS. Po ukonceni prenosu je vygenerovano preruseni.
- vyhodou je, ze velke datove prenosy nevytesnuji data z cache
- DMA radic nemusi mit jen disky, ale napr. i sitove rozhrani

Autonomni kanal(Bus Master DMA) (Pozn. to co je ve slidech k BM DMA je temer totozny s beznym DMA, presto to uvadim. na wiki jsem nasel ale neco navic.)
- inteligence presunuta do zarizeni
- periferie je doplnena o vlastni radic
- prubeh prenosu

1. nadrizeny procesor vlozi sekvenci datovych bloku do pameti
2. nakonfiguruje nebo primo naprogramuje radic periferie. ta provede sekveni prenosu
3. po uplnem nebo castecnem preruseni je informovan procesor

- - - - - -
Dle wikipedie se Bus Master DMA od klasickeho lisi tim, ze periferie muze dostat kontrolu nad pametovou sbernici a zapsat tak primo do pameti bez jakehokoliv zapojeni ze strany CPU.

Vyjimky a preruseni

- Vyjimky
  - pro MIPS napr. mat. preteceni, nactena neznama instr., systemove volani
  - nedostupnost dat nebo selhani zapisu
- Preruseni
  - maskovatelna - lze zakazat ve stavovem slovu CPU
  - nemaskovatelna - casto osetreni HW chyb, hlidaci obvod

Vyjimky jsou prijaty temer vzdy, preruseni jen pokud jsou nemaskovatelna nebo povolena.

Zpracovani vyjimky/preruseni:

1. stavove slovo(PSW) a PC se ulozi bud na zasobnik nebo do spec. registru
2. PC se nastavi na adresu obsluzne rutiny pripadajici dane vyjimce pripadne i cislu zdroje preruseni, ktera je nasledne vykonana
3. V zavislosti na typu vyjimky/preruseni doje ke specifickemu zpracovani
4. provede se instrukce CPU pro uvedeni do stavu pred zpracovanim vyjimky/preruseni (instrukce navratu z preruseni) a obnovi se puvodni registry (PC a PSW)

Pri spravnem obslouzeni vyjimky by puvodni program nemel primo poznat, ze k preruseni doslo.

Urceni zdroje vyjimky/preruseni

- soft. hledani
  - veskera preruseni a vyj. spousteji rutinu od stejne adresy. rutina zjisti duvod preruseni ze stavoveho registru (!= PSW)
- vektorova obsluha preruseni
  - cislo zdroje zjisti CPU
  - v pameti se nachazi na pevnem miste (specifikovane ridicim registrem VBR) tabulka vektoru preruseni, CPU prevede cislo zdroje na index a z nej nacte v poli slovo, kt. vlozi do PC
- nevektorova obsluha vice pevne urcenych adres podle priorit
- casto i kombinovane

Async vs. sync preruseni: async. nejsou vazane na instrukci, zatimco sync ano

#32 APO-2

Petr Svec

Mnohaurovnova struktura pocitacu a virtualizace

Struktura

Viceurovnova struktura ma za cil usnadnit praci programatorum, pridanim dodatecnych urovni abstrakce - predevsim jazyku vyssi urovne.

- druhou nejnizsi urovni (znaceno L2) je strojovy jazyk tj. nuly a jednicky (prvni nema ani moc smysl uvazovat, jelikoz se k ni normalni smrtelnik ani nedostane viz. dale).
- pro lidi samozrejme necitelny, coz je duvodem existence vyssich urovni, ale na druhou stranu to nejrychlejsi co na danem hardwaru dostaneme, takze kazdy program ve vysledku bezi na tehle urovni.

Virtualni pocitac
- na kazde vyssi urovni nez je strojova zavadime virtualni pocitac M_i s jazykem L_i. Program napsany v L_i se preklada do L_{i-1} nebo je interpretovan v M_{i-1} pro $i > 2$ (pri i = 1 jsme na strojovem jazyce).
- pro pripomenuti:

  - interpretace = program v L_{i-1} zpracovava L_i jako data
  - kompilace = prevod instrukci z L_i na instrukce v L_{i-1}

Bezna struktura soucasneho pocitace
Serazeno od nejnizsi uroven po nejvysi se soucasny pocitac sklada z nasledujicich urovni:

1. L1 - mikrokod (mikroinstrukce) - preklada strojovy kod na instrukce pro praci s obvody - interpretovany, ale velice rychly. Byl pridan az pozdeji tj. po strojovem kodu.
2. L2 - strojovy kod
3. L3 - uroven operacniho systemu (OS)
4. L4 - assembly lang.(ASM)
5. L5 - high lvl lang.

Pozn. ve slidech je uveden OS jako nizsi nez assembly lang. Osobne mi to prijde podivny, jelikoz napr. u RISCovych procesoru bez instrukce nasobeni je nejprve treba funkci OS, ktere nasobeni realizuji abychom dostali kod v ASM. Dalsim prikladem je keyword "new" v C++ na alokaci pameti, ktere se take musi preves na systemove volani.

Strojova uroven - M_2
- definovana instrukcni sadou
- instrukcni format se sklada z OPcode (kod instrukce) a 0-3 adres/registru (pri 0 je adresa/registr implicitni)
- u RISC je nejcastejsi 3 adresovy format u CISC 2 adresovy

Virtualizace

Principem virtualizace je skryti nizsich vrstev a implementace vrstvy, ktera se tvari jako samostatny HW.

V. se deli na nekolik skupin:

- Na urovni jazyka a byte kodu - virtualni stroje (Java, .NOT)
- Emulace jine pocit. architektury
- Nativni virtualizace - izolovane prostredi poskytujici shodny typ architektury pro nemodifikovany OS
- Virtualizace s plnou podporou HW (napriklad vetsina novych CISCovych procesoru ji v sobe ma)
- Castecna virtualizace - typicky jen adresni prostory
- Paravirtualizace
- Virtualizace na urovni OS - oddelena uzivatelska rozhrani

Hypervizor (Virtual machine monitor - VMM)
- program, ktery zajistuje zakladni praci s hostovanymi OS, mezi ktere patri:

- monitorovani hostovanych OS, osetrovani vyjimek a emulace privilegovanych intrukci
- emulace periferii, preruseni, ktere mohou generovat, predavani dat do fyzickych zarizeni na hostujicim OS/systemu.
- rozdeleni zdroju mezi hostovane OS

Hypervizor muze byt implementovany jako program/proces uzivatelskeho prostoru v hostitelskem OS (QEMU), s pouzitim HW akcelerace a podpory v jadre OS (KVM) nebo jako samostatny system vyuzivajici system v jedne domene (jednom virtualizovanem prostoru vyhrazenem pro jeden hostovany OS) pro komunikaci s HW a z nej pak posila data ostatnim (XEN).

Xen (to je virtualizacni "program"/system)
V paravirtualizaci (XEN) systemova volani aplikace bezici v hostovanem OS nevolaji primo hostujici kernel, ale hypervizor. Hypervizor prenasi cast sve prace na hostujici sys. misto aby provadel syscall ve virtualizovane domene. Nevyhodou je, ze hostovany sys. musi byt upraveny na miru hypervizoru.

Klasicka architektura

Pozn. nasledujici text je z vetsi casti z wiki, jelikoz ve slidech neni temer nic uzitecnyho krome obrazku.

Klasickou architekturou je myslena CISC a (dle toho mala co je ve slidech) konkretne architektura m68k tj. Motorola rada 68xxx.

- m68k ma jak 16-bit tak 32-bit modely
- big endian
- ma 8 datovych a 8 adresovych registru, pricemz 8. adresovy je User Stack Pointer (USP) - jeho chovani je ovlivneno nasledujicim registrem
- status register (= PSW - Program Status Word) je 16-bitovy. Spodnich 8-bitu je pro tzv. Condition Code Register (CCR), hornich 8 pro system (sys. byte).

CCR (cisla jsou indexy PSW)

0 - carry bit - nastavi se na 1 pri prenosu z nejvice vyznam. bitu pri arit. operaci nebo kdyz je treba vypujcka pri odecitani

1 - overflow bit - pri preteceni na 1

2 - zero bit - pokud jsou vsechny bity operandu nebo vysledku nulove

3 - negative bit - pokud je nejvice vyznamny bit operandu nebo vysledku roven 1 (negativni vysledek v doplnkovem kodu)

4 - extended carry - pouziva se pri rozsireni operaci na praci s vicenasobnou presnosti, ma stejnou hodnotu jako carry

Pozn. zbyle 3 bity nejsou ve slidech primo zmineny, ale na obrazku v nich uvedenem jsou nastaveny na 0. vic jsem nehledal
System byte

- 8,9,10 - interrupt mask - definuje uroven preruseni pro kterou je prijeti zadosti blokovane
- 14,15 - trace bits - pokud je nektery z nich nastaveny, tak dojde ke generovani vyjimky po provedeni kazde instrukce nebo po instrukcich ovlivnujicich tok kodu

11. a 12. jsou 0

Systemove a IO sbernice

Pozn. Ve slidech je vse ukazovano na prikladu PC, takze nedostatek obecnosti je zamerny a neni to chyba.

sbernice(bus)

- spousta dratu, na ktery se pripojuji zarizeni v ramci pocitace, zjednodussuje komunikaci oproti point-to-point spojeni vsech komponent
- v jednom pocitaci neni typicky jen jedna sbernice, ale hned nekolik - procesorova, systemova, lokalni a IO sbernice
- podle typu "obsahu", ktery obsluhuje existuji 3 typy - adresova, datova, ridici. Muze byt i kombinovana

Nejblize CPU jsou Front-side Bus (FSB) a Back-side Bus (BSB). FSB je pripojeni na North Bridge - systemovy cip, ktery obsluhuje pameti a dnes jiz prehistoricke AGP (na to se pripojovalo GPU). V soucasny dobe je North b. vetsinou jiz soucasti CPU (AMD a Intel). BSB jsou zadni vratka do pameti pro L2 cache, ktera z ni cte primo (BSB je na cteni z pameti zoptimalizovana, a tak pomerne rychla).
S North b. je spojeny South bridge, na ktery se uz pripojuji ostatni sbernice a zarizeni (PCI, PCIe, USB etc.).

dvoubodove spojeni (point-to-point)

- pri malem poctu zarizeni je nejjednodussi, pri vetsim ale znacne komplikovane
- poskytuje vyssi vykon nez sbernice, jelikoz se zadne zarizeni nemusi o sve spojeni delit s jinym

Multiplexovana sbernice(time division multiplexing) - sbernice, kde jsou data nebo adresy vetsi velikosti nez jsou vodice schopne prenest, a tak se deli na casti. Priklad: sbernice prenese najednou 16-bit, ale nase data maji 32-bit → rozdeli se a poslou na dvakrat. Jsou take pripady,

kdy se posilaji adresy i data po stejnych dratech.

PCI
- PCI je sbernice z dob kdy dinosauri chodili po Zemi
- komunikace je paralelni. To mimo jine vyzaduje, aby byl pritomen arbitr sbernice - komponenta urcujici, ktere z PCI zarizeni posila po sbernici data (a adresy) v dany okamzik.
- vzdy, kdyz chce lib. PCI zarizeni posilat data po sbernici musi nejprve pozadat arbitra o prideleni prava
- kazdeho prenosu/transakce se ucasni 2 zarizeni - initiator(Bus Master) a target. Arbitr take resi pripad vice bus masteru (multimaster sbernice)
- prenos po sbernici ma 2 faze - adresovou a datovou
- zarizeni na sbernici maji spolecny 4-bit prerusovaci podsystem (narozdil od ostatnich dratu nejsou paralelni)
- sbernice je multiplexovana, adresy i data se posilaji pres stejne piny PCI zarizeni

PCI vs. PCI-Express (PCIe)
- PCIe ma narozdil od PCI prepinac, na ktere jsou pripojena jednotliva PCIe zarizeni. Prepinac samostny je je spojeny se sbernici. Komunikace je timpadem, narozdil od PCI, seriova
- rychlost PCIe zarizeni se udava poctem linek (x1, x16 atd.). Kazda linka se sklada z jednoho nebo vice proudu a ten je tvoren 2 vodici (jeden v kazdem smeru)
- komunikace je full duplex - 1 bit za takt v obou smerech

Otázka číslo 33 (A4B33OPT)

Martin Stránský

10. června 2012

"Snažili jsme se udělat to co nejlépe, ale dopadlo to jako vždycky."

1 Lineární programování

Linieární programování znamená řešení úlohy minimalizace lineární funkce $f = c^T x$ $(+d)$ za podmínek affinních funkcí g_i, h_i (affinita viz níže).

Slouží například k řešení úloh (viz skripta):

- optimální výrobní program (z různých druhů surovin vyrábíme různé druhy zboží s různou cenou)
- směšovací problém (kuchařka má uvařit oběd, aby v něm bylo b_1 vitamínů, b_2 bílkovin a b_3 tuků)
- distribuční problém
- dopravní problém
- hledání rovnovážných stavů u lineárních systémů a pod.

Obecná formulace úlohy LP:

$$\begin{array}{lll} \min\,(\max) & c_1x_1 + c_2x_2 + \ldots c_1x_n & \\ z.p. & a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i & i \in I_+ \\ & a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i & i \in I_- \\ & a_{i1}x_1 + \ldots + a_{in}x_n = b_i & i \in I_0 \\ & x_j \geq 0 & j \in J_+ \\ & x_j \leq 0 & j \in J_- \\ & x_j \in \mathbb{R} & j \in J_0 \end{array}$$

Obecnou formulaci převádíme na standardní tvar

$$\begin{array}{lll} \min & c_1x_1 + c_2x_2 + \ldots c_1x_n & \\ z.p. & a_{i1}x_1 + \ldots + a_{in}x_n = b_i & i = 1, \ldots, m \\ & x_j \geq 0 & j = 1, \ldots, n \end{array}$$

zkráceně

$$\min\{c^T x \mid Ax = b,\, x > 0\}$$

a zpět následovně:

- Maximalizaci nahradíme minimalizací podle $\min_{x \in X} f(x) = -\max_{x \in X}(-f(x))$.
- ($a_{i1}x_1 + \ldots + a_{in}x_n = b_i$ nahradíme $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ a $a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i$.)
- $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ doplníme slackovou proměnnou $u_i \geq 0$, $a_{i1}x_1 + \ldots + a_{in}x_n + u_i = b_i$.
- $a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i$ vynásobíme -1 a doplníme slackovou proměnnou $u_i \geq 0$, $-a_{i1}x_1 - \ldots - a_{in}x_n + u_i = -b_i$.
- $x_i \in \mathbb{R}$ převedeme na $x_i^+ \geq 0$, $x_i^- \geq 0$, $x_i = x_i^+ - x_i^-$.

Příklad (vlastní): převedení z obecného tvaru na standardní.

$$\begin{array}{ll} \max & c^T x \\ z.p. & a_{11}x_1 + \ldots + a_{2n}x_n \geq b_i \\ & a_{21}x_1 + \ldots + a_{2n}x_n \leq b_i \\ & a_{i1}x_1 + \ldots + a_{in}x_n = b_i \\ & x_1, \ldots, x_n \geq 0 \end{array}$$

se převede na

$$\begin{array}{lll} -\min & -c_1x_1 \ldots - c_nx_n & \\ z.p. & -a_{11}x_1 - \ldots - a_{2n}x_n + u_1 = -b_i & \\ & a_{21}x_1 + \ldots + a_{2n}x_n + u_2 = b_i & \\ & a_{i1}x_1 + \ldots + a_{in}x_n = b_i & i \in [3, m] \\ & x_1, \ldots, x_n, u_1, u_2 \geq 0 & \end{array}$$

Ještě poznámka k řešení přeurčených rovnic $Ax = b$, $A \in \mathbb{R}^{m \times n}$ $(m > n)$. Řeší se taková úloha:

$$\min\{||Ax - b||_p \mid x \in \mathbb{R}\}.$$

Dualita

Každá úloha LP má svojí duální úlohu, kterou lze dostat následující konstrukcí. Z duální úlohy lze poté tím samým postupem dostat primární. Popíšu jenom konstukci ze speciálního tvaru, více ve skriptech (s. 95):

$$\begin{array}{llll} \min & \sum_j c_j x_j & \max & \sum_i y_i b_i \\ z.p. & \sum_j a_{ij} x_j \geq b_i & z.p. & y_i \geq 0 \\ & x_j \geq 0 & & \sum_i y_i a_{ij} \leq c_j \end{array}$$

přehledněji:

$$\begin{array}{llll} \min & c^T x & \max & y^T b \\ z.p. & Ax \geq b & z.p. & y_i \geq 0 \\ & x \geq 0 & & y^T A \leq c^T \end{array}$$

U vět budeme postupovat přesně opačně, než jak je tomu ve skriptech:

Důležitá je zejména silná věta o dualitě která říká, že když se $c^T x = y^T b$ (kritéria) pro přípustná x, y ($Ax \geq b$ atd.), pak jsou x i y optimálními řešeními obou úloh.

Přitom platí (věta o slabé dualitě), že $c^T x \geq y^T b$ pro jakékoli přípustné x, y; protože $y^T A \leq c^T$ vynásobeno zprava $x \geq 0$ rovná se $y^T Ax \leq c^T x$ (a stejně tak pro x), takže napsáno v řadě $c^T x \geq y^T Ax \geq y^T b$.

Věta o komplementaritě udává podmínky, kdy ($\Leftrightarrow$) se $c^T x = y^T b$, více ve skriptech na s. 96.

Dualita je dobrá k:

- lepšímu pochopení problému
- ověření optimality
- někdy lze jednodušeji spočítat duální úlohu a z ní řešení primární, než rovnou primární.

Takhle shrnuto to snad stačí.

2 Simplexový algoritmus

Řeší úlohu lineárnho programování ve standatdním tvaru $\min\{c^T x - d \mid Ax = b,\ x > 0\}$.

2.1 Části algoritmu

2.1.1 Přechod k sousední bázi

Co nejjednodušeji: máme soustavu $[A \mid b]$ tak, že v m *bázových* sloupcích jedna jednička a samé nuly a na každém řádku je alepoň jedna jednička.

$$\begin{array}{ccccc|c} 5 & 0 & 0 & -1 & 1 & 3 \\ 3 & 0 & 1 & 2 & 0 & 5 \\ -1 & 1 & 0 & 2 & 0 & -2 \end{array}$$

Chceme přejít k sousední bázi, která bude mít jeden jiný bázový sloupec, to znamená, že jestliže jsou teď báze sloupce 2, 3, 5, pak příště to bude například 1, 3, 5 nebo 2, 3, 4.

Zvolme pivot a_{ij} (níže) a vydělme řádek i a_{ij} (dostaneme na místo pivotu jedničku). Dále pomocí násobků ostatních řádků vynulujeme prvky ve sloupcích s bázemi (jako v GEM). Zbylé báze mají ve sloupci jedničku, takže to jde dobře.

2.1.2 Přípustné bázové řešení

Protože nenulové složky bázového řešení jsou rovny složkám vektoru b, je bázové řešení přípustné tehdy, když $b \geq 0$. Pakliže teď máme přípustné řešení, k tomuto nedojde když $a_{ij} \geq 0$ kde a_{ij} je pivot a pro každé $i' \neq i$ platí $a_{i'j} \leq 0$ nebo $b_i / a_{ij} \leq b_{i'} / a_{i'j}$

2.1.3 Nekladný sloupec

Pakliže jeden z nebázových sloupců obsahuje jen nekladné prvky, leží optimum v $-\infty$ - úloha je neomezená.

2.1.4 Úpravy účelového řádku

Simplexová tabulka:

$$\begin{bmatrix} c & d \\ A & b \end{bmatrix}$$

Přičtení jakékoli lineární kombinace z $[A\,d]$ k $[c\,d]$ zachová hodnotu účelové funkce.

2.2 Algoritmus

1. Vyber pivot (kde výběrem $j \in \operatorname{argmin} c_j$ se zajistí, že účelová funkce nestoupne). i se vybírá například $\operatorname{argmin} b_i/a_{ij}$.
2. Udělej ekvivalentní úpravu.
3. Vynuluj c_j pro bázová j.
4. Končíme, když všechny koeficienty c_j jsou nezáporné (optimum) nebo když máme nekladný sloupec (neomezená úloha).

3 Něco o tom zbytku

Konvexní množinu tvoří konvexní kombinace vektorů (je definována tak, že je uzavřená na konvexní kombinace). Konvexní kombinace je současně affinní a nezáporná kombinace: $\sum_k \alpha \geq 1 \wedge \forall k[\alpha_k \geq 0]$.

Konvexní polyedr je definován jako průnik konečně mnoha poloprostorů (H-reprezentace (half-space)). Extrémní body jsou vrcholy polyedru a jejich specifikem je to, že je lze dostat právě jednou konvexní kombinací. Reprezentovat jde ještě jako konvexní obal konečně mnoha vektorů (V-reprezentace (vertex)).

Funkce je konvexní právě tehdy, když $f(\alpha x+(1-\alpha y)) = \alpha f(x)-(1-\alpha)f(y)$.

Konvexní optimalizační úlohu řešíme tehdy, když je účelová funkce konvexní. Nejdůležitější je, že nalezené (lokální) minimum je vždy zároveň globální. Například u simplexové metody postupujeme po vrcholech konvexního polyedru k tomu nejnižšímu. K konvexním optimalizačním úlohám patří LP a QP (qudratic, také vytváří konvexní množiny).

34 (A4B33OPT)

1 Metoda nejmenších čtverců

Metoda nejmenších čtverců je matematicko-statistická metoda používaná zejména při zpracování nepřesných dat (typicky experimentálních empirických dat získaných například měřením). Metoda je v základní podobě určená pro řešení nekompatibilních soustav lineárních rovnic (v obecnější podobě hovoříme o nelineární metodě nejmenších čtverců), díky čemuž je fakticky ekvivalentní tzv. lineární regresi.

Řešme nehomogenn soustavu linearnch rovnic Ax = b

A je matice o rozměrech m×n a soustava má řešení právě tehdy, když b∈rngA, jinak je soustava přeurčená (typicky, když m>n, víc rovnic než neznámých)

V našem případě řešíme přeurčenou soustavu, tedy hledáme takové x, že vzdálenost mezi body Ax a b je co nejmenší, tedy:

$min\{||Ax - b|| \mid x \in R^n\}$

Místo normy klidně můžeme minimalizovat její čtverec:

$||Ax - b||^2$

1.1 Použití na regresi:

Regrese je modelování zavislosti proměnné y∈R na proměnné t∈T regresn funkcí y = f(t, x), která je známá, až na parametry $x \in R^n$ Je dán soubor dvojic (t_i, y_i), i = 1, ... ,m, kde měření $y_i \in R$ jsou zatížena chybou. Úkolem je najít parametry x, aby y_i f(t_i, x). Podle metody nejmenších čtverců tedy řešme.

$$\min_{x \in R^n} \sum_{i=1}^{m} (f(t_i, x) - y_i)^2$$

2 Analytické podmínky na lokální extrémy

2.1 Volné extrémy

V tomto případě hledáme lokální extrémy funkce.

Máme dva důležité typy bodů:

1. stacionární bod - bod, kde je funkce diferencovatelná a všechny parciální derivace jsou nulové.
2. kritický bod - bod, který je buď stacionární nebo v něm není funkce diferencovatelná.

2.1.1 Podmínka prvního řádu

Všechny kritické body jsou podezřelé z volného lokálního extrému.[1]

[1] Když počítáme příklad nejdříve uděláme všechny parciální derivace, následně je položíme rovny nule a vyřešíme soustavu rovnic => kritické body

2.1.2 Podmínky druhého řádu

- f má v bodě x ostré lokální minimum [maximum] na X právě tehdy, když Hessova matice druhých derivací f"(x) je pozitivně[2] [negativně[3]] definitní.
- Je-li f"(x) indefinitní[4], nemá f v x lokální minimum ani lokální maximum na X.
- Je-li f"(x) pozitivně [negativně] semidefinitní, nevíme o tomto bodě jestli je minimem, maximem nebo ani jedno z toho.

2.2 Vázané extrémy

V tomto případě hledáme lokální extrémy funkce za určité podmínky dané nejčastěji jinou funkcí nebo funkcemi.

2.2.1 postup

Extrémy hledáme za pomoci lagrangeových multiplikátorů λ, řešíme rovnici:
$f'(x) + \lambda g'(x) = 0^T$ [5]

3 Numerické metody pro optimalizaci bez omezení

U všech dále zmíněných případů se jedná o iterační numerické metody pro nalezení volného lokálního minima diferencovatelných funkcí na množině R^n

3.1 Gradientní metoda

Metoda volí směr sestupu jako záporný gradient funkce f v bodě x_k[6].
Tedy $x_{k+1} = x_k - \alpha_k (A^T A)^{-1} f'(x_k)^T$
Rychlost konvergence bývá často pomalá, kvůli cik-cak chování.

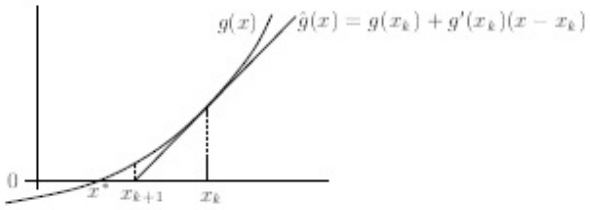
3.2 Newtonova metoda

Newtonova metoda je iteracn algoritmus na resen soustav nelinearnch rovnic. Lze ho pouzt i na minimalizaci funkce tak, ze hledame nulovy gradient.

[2] V příkladech hledáme vlastní čísla matice, když jsou všechna > 0, pak je poz. def.
[3] V příkladech hledáme vlastní čísla matice, když jsou všechna < 0, pak je neg. def.
[4] V příkladech hledáme vlastní čísla matice, když existuje vlastní číslo, které < 0 a zároveň existuje vlastní číslo, které >0, pak je matice indefinitní
[5] Tedy v praxi si napíšeme zadání a podmínky si převedeme do tvaru, kdy je na jedné straně rovnice nula a roznásobíme je λ_1 *až* λ_n, tento výraz se nazývá Lagrangeova funkce, ze které spočítáme první derivace a vyřešíme soustavu rovnic, z níchž dostaneme body podezřelé z extrémů. (zjištění druhu extrému je podle skript i wernera složité a v OPT vůbec nebylo)
[6] tj. vždy jdeme nejstrmějším směrem dolů

3.2.1 Použití na soustavy nelineárních rovnic

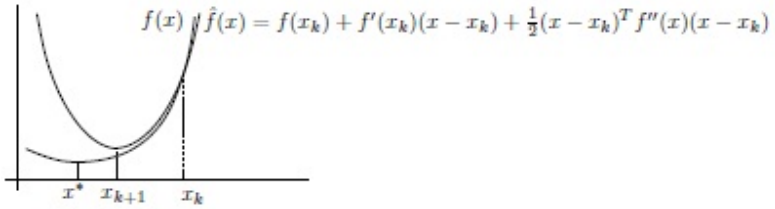
Zobrazení g aproximujeme v okolí bodu x_k taylorovým polynomem prvního řádu $g(x) = g(x_k) + g'(x_k)(x - x_k)$



Při řešení soustavy rovnic najdeme další krok, jako $x_{k+1} = x_k - g'(x_k)^{-1}g(x_k)$

3.2.2 Použití na hledání lokálního minima

V tomto případě aproximujeme funkci f taylorovým polynomem druhého řádu.



Tedy newtonovu metodu lze použít pro hledání lokálního extremu dvakrát diferencovatelné funkce, když položíme $g(x) = f'(x)^T$, z toho dostaneme, že iterace:

$x_{k+1} = x_k - f''(x_k)^{-1}f'(x_k)^T$

3.3 Gaussova-Newtonova metoda

Snažíme se najít přibližné řešení ve smyslu nejmenších čtverců, což vede na minimalizaci funkce:

$f(x) = ||g(x)||^2 = g(x)^T g(x)$

Další krok hledáme následovně:

$x_{k+1} = x_k - (g'(x_k)^T g'(x_k))^{-1}g'(x_k)^T g(x_k)$, což se dá v případě, že $g'(x_k)$ má plnou hodnost napsat, jako $x_{k+1} = x_k - g'(x_k)^+ g(x_k)$, kde $g'(x_k)^+$je pseudoinverze.

Výhody: Vyhneme se počítání druhých derivací (hesiánů)

Nevýhody: Metoda má horší konvergenční chování než Newtonova metoda

3.4 Levenberg-Marquardtova metoda

Levenbergova-Marquardtova metoda je široce používané vylepšení Gaussovy-Newtonovy metody, které matici $g'(x_k)^T g'(x_k)$ nahrazuje maticí $g'(x_k)^T g'(x_k)+$

$\mu_k I$, kde $\mu_k > 0$

Tedy další krok hledáme následovně:

$x_{k+1} = x_k - (g'(x_k)^T g'(x_k) + \mu_k I)^{-1} g'(x_k)^T g(x_k)$

Zajímavosti:

- Pro malé μ_k se Levenbergova-Marquardtova metoda blíží Gauss-Newtonově metodě.
- Pro velké μ_k se Levenbergova-Marquardtova metoda blíží Gradientní metodě s délkou kroku ${\mu_k}^{-1}$.

Parametr μ_k měníme po každé iteraci.

- Pokud iterace snížila účelovou funkci, pak iteraci přijmeme a μ_k zmenšíme
- Pokud iterace nesnížila účelovou funkci iteraci zamítneme a μ_k zvětšíme

Zmenšování azvětšování μ_k děláme násobením a dělením konstantou.