

# Učební texty k státní bakalářské zkoušce Programování

študenti MFF

15. augusta 2008



*Vážený študent/čitateľ,*

toto je zbierka vypracovaných otázok pre bakalárske skúšky Informatikov. Otázky boli vypracované študentmi MFF počas prípravy na tieto skúšky, a teda zatiaľ neboli overené kvalifikovanými osobami (profesormi/dokotorandmi mff atď.) - preto nie je žiadna záruka ich správnosti alebo úplnosti.

Väčšina textov je vypracovaná v čestine resp. slovenčine, prosíme dodržujte túto konvenciu (a obmedzujte teda používanie napr. anglických textov). Ak nájdete nejakú chybu, nepresnosť alebo neúplnú informáciu - neváhajte kontaktovať administrátora alebo niektorého z prispievateľov, ktorý má write-prístup k svn stromu, s opravou :-). Podobne - ak nájdete v „texte“ veci ako ??? a TODO, znamená to že danú informáciu je potrebné skontrolovať, resp. doplniť...

Texty je možné ďalej používať a šíriť pod licenciou **GNU GFDL** (čo pre všetkých prispievajúcich znamená, že musia súhlasiť so zverejnením svojich úprav podľa tejto licencie).

Veríme, že Vám tieto texty pomôžu k úspešnému zloženiu skúšok.

---

**Hlavní writeři :-)** :

- *ajs*
- *andree* – <http://andree.matfyz.cz/>
- *Hydrant*
- *joshis* / *Petr Dvořák*
- *kostej*
- *nohis*
- *tuetschek* – <http://tuetschek.wz.cz/>

Úvodné verzie niektorých textov vznikli prepisom otázok vypracovaných „písomne na papier“, alebo inak ne- $\text{\TeX}$ -ovskými. Autormi týchto pôvodných verzií sú najmä nasledujúce osoby: *gASK*, *Grafí*, *Kate* (mat-15), *Nytram*, *Oscar*, *Stando*, *xStyler*. Časť je prebratá aj z pôvodných súborkových textov... Všetkým patrí naša/vaša vďaka.

# Obsah

<b>1</b>	<b>Základy teoretické informatiky</b>	<b>4</b>
1.1	Logika – jazyk, formule, sémantika, tautologie . . . . .	4
1.2	Rozhodnutelnost, splnitelnost, pravdivost a dokazatelnost . . . . .	6
1.3	Normální tvary výrokových formulí, prenexní tvary formulí predikátové logiky . . . . .	11
1.4	Automaty – Chomského hierarchie, třídy automatů a gramatik, determinismus a nedeterminismus. . . . .	12
<b>2</b>	<b>Algoritmy a datové struktury</b>	<b>18</b>
2.1	Časová složitost algoritmů, složitost v nejhorším a průměrném případě	19
2.2	Třídy složitosti P a NP, převoditelnost, NP-úplnost . . . . .	21
2.3	Binární vyhledávací stromy, vyvažování, haldy . . . . .	22
2.4	Hašování . . . . .	27
2.5	Sekvenční třídění, porovnávací algoritmy, přihrádkové třídění, třídící síť . . . . .	32
2.6	Grafové algoritmy . . . . .	38
2.7	Tranzitivní uzávěr . . . . .	47
2.8	Algoritmy vyhledávání v textu . . . . .	48
2.9	Algebraické algoritmy . . . . .	51
2.10	Základy kryptografie, RSA, DES . . . . .	53
<b>3</b>	<b>Databáze</b>	<b>57</b>
3.1	Podstata a architektury DB systémů . . . . .	57
3.2	Konceptuální, logická a fyzická úroveň pohledu na data . . . . .	61
3.3	Algoritmy návrhu schémat relací . . . . .	63
3.4	Transakční zpracování, vlastnosti transakcí, uzamykací protokoly, zablokování . . . . .	66
3.5	ER-diagramy, metody návrhu IS . . . . .	71
3.6	SQL . . . . .	73
3.7	Indexy, trigger, uložené procedury, uživatelé . . . . .	78
3.8	Vícevrstevné architektury . . . . .	81
3.9	Vazba databází na internetové technologie . . . . .	83
3.10	Organizace dat na vnější paměti, B-stromy a jejich varianty . . . . .	83
<b>4</b>	<b>Programovací jazyky a překladače</b>	<b>90</b>
4.1	Principy a základy implementace objektově orientovaných jazyků a jazyků s blokovou strukturou, běhová podpora vyšších programovacích jazyků . . . . .	90
4.2	Oddělený překlad, sestavení, řízení překladu . . . . .	98
4.3	Neprocedurální programování, logické programování . . . . .	100
4.4	Struktura překladače, lexikální, syntaktická analýza . . . . .	105
4.5	Interpretované jazyky, virtuální stroje . . . . .	112
4.6	Pojmy a principy objektového návrhu . . . . .	114
4.7	Generické programování a knihovny . . . . .	116
4.8	Návrhové vzory . . . . .	118

<b>5</b>	<b>Architektura počítačů a operačních systémů</b>	<b>121</b>
5.1	Architektury počítače . . . . .	121
5.2	Procesory, multiprocesory . . . . .	124
5.3	Sběrnice, protokoly . . . . .	129
5.4	Vstupní a výstupní zařízení . . . . .	130
5.5	Architektury OS . . . . .	131
5.6	Vztah OS a HW, obsluha přerušení . . . . .	132
5.7	Procesy, vlákna, plánování . . . . .	133
5.8	Synchronizační primitiva, vzájemné vyloučení . . . . .	134
5.9	Zablokování a zotavení z něj . . . . .	138
5.10	Organizace paměti, alokační algoritmy . . . . .	139
5.11	Principy virtuální paměti, stránkování, algoritmy pro výměnu stránek, výpadek stránky, stránkovací tabulky, segmentace . . . . .	141
5.12	Systémy souborů, adresářové struktury . . . . .	144
5.13	Bezpečnost, autentifikace, autorizace, přístupová práva . . . . .	147
5.14	Druhy útoků a obrana proti nim . . . . .	147
5.15	Kryptografické algoritmy a protokoly . . . . .	148
<b>6</b>	<b>Sítě a internetové technologie</b>	<b>155</b>
6.1	Architektura ISO/OSI . . . . .	156
6.2	Rodina protokolů TCP/IP (ARP, IPv4, IPv6, ICMP, UDP, TCP) – adresace, routing, fragmentace, spolehlivost, flow control, congestion control, NAT . . . . .	158
6.3	Rozhraní BSD Sockets . . . . .	169
6.4	Spolehlivost - spojované a nespojované protokoly, typy, detekce a oprava chyb . . . . .	171
6.5	Bezpečnost – IPsec, principy fungování AH, ESP, transport mode, tunnel mode, firewalls . . . . .	172
6.6	Internetové a intranetové protokoly a technologie – DNS, SMTP, FTP, HTTP, NFS, HTML, XML, XSLT a jejich použití . . . . .	174

# 1 Základy teoretické informatiky

## Požadavky

- Logika - jazyk, formule, sémantika, tautologie
- Rozhodnutelnost, splnitelnost, pravdivost, dokazatelnost
- Normální tvary výrokových formulí, prenexní tvary formulí predikátové logiky
- Automaty - Chomského hierarchie, třídy automatů a gramatik, determinismus a nedeterminismus.

## 1.1 Logika – jazyk, formule, sémantika, tautologie

### Jazyk

#### Logika prvního řádu

Formální systém logiky prvního řádu obsahuje jazyk, axiomy, odvozovací pravidla, věty a důkazy. Jazyk prvního řádu zahrnuje:

- neomezeně mnoho proměnných  $x_1, x_2, \dots$
- funkční symboly  $f_1, f_2 \dots$ , každý má aritu  $n \geq 0$
- predikátové symboly  $p_1, p_2 \dots$ , každý má aritu
- symboly pro logické spojky ( $\neg, \vee, \&, \rightarrow, \leftrightarrow$ )
- symboly pro kvantifikátory ( $\forall, \exists$ )
- může (ale nemusí) obsahovat binární predikát " $=$ ", který pak se pak ale musí chovat jako rovnost, tj. splňovat určité axiomy (někdy se potom proto řadí mezi logické symboly).

Proměnné, logické spojky a kvantifikátory jsou *logické symboly*, ostatní symboly se nazývají *speciální*.

#### Definice (Jazyk výrokové logiky)

Jazyk výrokové logiky je jazyk prvního řádu, obsahující *výrokové proměnné* („prvotní formule“), *logické spojky*  $\neg, \vee, \&, \rightarrow, \leftrightarrow$  a *pomocné symboly* (závorky).

#### Definice (Jazyk predikátové logiky)

Jazyk predikátové logiky je jazyk prvního řádu, obsahující proměnné, *predikátové symboly* (s nenulovou aritou), *funkční symboly* (mohou mít nulovou aritu), symboly pro logické spojky a symboly pro *kvantifikátory*.

### Formule

#### Definice (Formule výrokové logiky)

Pro jazyk výrokové logiky jsou následující výrazy formule:

1. každá výroková proměnná
2. pro formule  $A, B$  i výrazy  $\neg A, (A \vee B), (A \& B), (A \rightarrow B), A \leftrightarrow B$
3. každý výraz vzniknuvší konečným užitím pravidel 1. a 2.

Množina formulí se nazývá *teorie*.

**Definice (Term)**

V predikátové logice je *term*:

1. každá proměnná
2. výraz  $f(t_1, \dots, t_n)$  pro  $f$   $n$ -ární funkční symbol a  $t_1, \dots, t_n$  termy
3. každý výraz vzniknuvší konečným užitím pravidel 1. a 2.

Podslovo termu, které je samo o sobě term, se nazývá *podterm*.

**Definice (Formule predikátové logiky)**

V predikátové logice je formule každý výraz tvaru  $p(t_1, \dots, t_n)$  pro  $p$  predikátový symbol a  $t_1, \dots, t_n$  termy. Stejně jako ve výrokové logice je formule i (konečné) spojení jednodušších formulí log. spojkami. Formule jsou navíc i výrazy  $(\exists x)A$  a  $(\forall x)A$  pro formuli  $A$  a samozřejmě cokoliv, co vznikne konečným užitím těchto pravidel.

Podslovo formule, které je samo o sobě formule, se nazývá *podformule*.

**Definice (Volné a vázané proměnné)**

Výskyt proměnné  $x$  ve formuli je *vázaný*, je-li tato součástí nějaké podformule tvaru  $(\exists x)A$  nebo  $(\forall x)A$ . V opačném případě je *volný*. Formule je *otevřená*, pokud neobsahuje vázanou proměnnou, je *uzavřená*, když neobsahuje volnou proměnnou. Proměnná může být v téže formuli volná i vázaná (např.  $(x = z) \rightarrow (\exists x)(x = z)$ ).

**Sémantika****Definice (Pravdivostní ohodnocení ve výrokové logice)**

Výrokové proměnné samotné neanalyzujeme – jejich hodnoty máme dány už z vnějšku, máme pro ně množinu pravdivostních hodnot  $\{0, 1\}$ .

*Pravdivostní ohodnocení  $v$*  je zobrazení, které každé výrokové proměnné přiřadí právě jednu hodnotu z množiny pravdivostních hodnot. Je-li známo ohodnocení proměnných, lze určit *pravdivostní hodnotu  $\bar{v}$*  pro každou formuli (při daném ohodnocení) – indukcí podle její složitosti, podle tabulek pro logické spojky.

**Definice (Interpretace jazyka predikátové logiky)**

*Interpretace jazyka* je definována množinovou strukturou  $\mathcal{M}$ , která ke každému symbolu jazyka a množině proměnných přiřadí nějakou množinu individuí.  $\mathcal{M}$  obsahuje:

- neprázdnou množinu individuí  $M$ .
- zobrazení  $f_M : M^n \rightarrow M$  pro každý  $n$ -ární funkční symbol  $f$
- relaci  $p_M \subset M^n$  pro každý  $n$ -ární predikát  $p$

Interpretace termů se uvažuje pro daný jazyk  $L$  a jeho interpretaci  $\mathcal{M}$ . *Ohodnocení proměnných* je zobrazení  $e : X \rightarrow M$  (kde  $X$  je množina proměnných). *Interpretace termu  $t$*  při ohodnocení  $e$  -  $t[e]$  se definuje následovně:

- $t[e] = e(x)$  je-li  $t$  proměnná  $x$
- $t[e] = f_M(t_1[e], \dots, t_n[e])$  pro term tvaru  $f(t_1, \dots, t_n)$ .

Ohodnocení závisí na zvoleném  $\mathcal{M}$ , interpretace termů při daném ohodnocení pak jen na konečně mnoha hodnotách z něj. Pokud jsou  $x_1, \dots, x_n$  všechny proměnné termu  $t$  a  $e, e'$  dvě ohodnocení tak, že  $e(x_i) = e'(x_i) \forall i \in \{1, \dots, n\}$ , pak  $t[e] = t[e']$ .

*Pozměněné ohodnocení*  $y$  pro  $x = m \in M$  je definováno:

$$e(x/m)(y) = \begin{cases} m(\text{pro } y \equiv x) \\ e(y)(\text{jinak}) \end{cases}$$

**Definice** (*Substituce, instance, substituovatelnost*)

*Substituce* proměnné za podterm v termu  $(t_{x_1, \dots, x_n}[t_1, \dots, t_n])$  je současné nahrazení všech výskytů proměnných  $x_i$  termy  $t_i$ . Je to term.

*Instance* formule je současné nahrazení všech volných výskytů nějakých proměnných za termy. Je to taky formule, vyjadřuje speciálnější tvrzení – ne vždy ale lze provést substituci bez změny významu formule. Term je *substituovatelný* do formule  $A$  za proměnnou  $x$ , pokud pro  $\forall y$  vyskytující se v  $t$  žádná podformule formule  $A$  tvaru  $(\exists y)B$  ani  $(\forall y)B$  neobsahuje volný výskyt  $x$ .

**Definice** (*Uzávěr formule*)

Jsou-li  $x_1, \dots, x_n$  všechny proměnné s volným výskytem ve formuli  $A$ , potom  $(\forall x_1) \dots (\forall x_n)A$  je *uzávěr* formule  $A$ .

## Tautologie

**Definice** (*Tautologie ve výrokové logice*)

Formule je *tautologie*, jestliže je pravdivá při libovolném ohodnocení proměnných ( $\models A$ ).

**Definice** (*Tautologický důsledek*)

Teorie  $U$  je *tautologický důsledek* teorie  $T$ , jestliže každý model  $T$  je také modelem  $U$  ( $T \models U$ ). Model nějaké teorie ve výrokové logice je takové ohodnocení proměnných, že každá formule z této teorie je pravdivá. K modelům se ještě vrátíme v následující sekci.

## 1.2 Rozhodnutelnost, splnitelnost, pravdivost a dokazatelnost

Z těchto témat se rozhodnutelnosti budeme věnovat až jako poslední, protože k vyslovení některých vět budeme potřebovat pojmy, definované v částech o splnitelnosti, pravdivosti a dokazatelnosti.

### Splnitelnost

**Definice** (*Splnitelnost*)

Množina formulí  $T$  ve výrokové logice je *splnitelná*, jestliže existuje ohodnocení  $v$  takové, že každá formule  $\forall A \in T$  je pravdivá při  $v$ . Potom se  $v$  nazývá *model teorie*  $T$  ( $v \models T$ ).

## Pravdivost

### Definice (Pravdivá formule výrokové logiky)

Formule výrokové logiky  $A$  je *pravdivá* při ohodnocení  $v$ , je-li  $\bar{v}(A) = 1$ , jinak je *nepravdivá*. Je-li formule  $A$  pravdivá při ohodnocení  $v$ , pak říkáme, že  $v$  je *model*  $A$  ( $v \models A$ ).

### Definice (Tarského definice pravdy)

Pro daný (redukovaný, tj. jen se „základními“ log. spojky) jazyk predikátové logiky  $L$ ,  $\mathcal{M}$  jeho interpretaci, ohodnocení  $e$  a  $A$  formuli tohoto jazyka platí:

1.  $A$  je *splněna v ohodnocení  $e$*  ( $\mathcal{M} \models A[e]$ ), když:
  - $A$  je atomická tvaru  $p(t_1, \dots, t_n)$ , kde  $p$  není rovnost a  $(t_1[e], \dots, t_n[e]) \in p_{\mathcal{M}}$ .
  - $A$  je atomická tvaru  $t_1 = t_2$  a  $t_1[e] = t_2[e]$
  - $A$  je tvaru  $\neg B$  a  $\mathcal{M} \not\models B[e]$
  - $A$  je tvaru  $B \rightarrow C$  a  $\mathcal{M} \not\models B[e]$  nebo  $\mathcal{M} \models C[e]$
  - $A$  je tvaru  $(\forall x)B$  a  $\mathcal{M} \models B[e(x/m)]$  pro každé  $m \in M$
  - $A$  je tvaru  $(\exists x)B$  a  $\mathcal{M} \models B[e(x/m)]$  pro nějaké  $m \in M$
2.  $A$  je *pravdivá v interpretaci  $\mathcal{M}$*  ( $\mathcal{M} \models A$ ), jestliže je  $A$  splněna v  $M$  při každém ohodnocení proměnných (pro uzavřené formule stačí jedno ohodnocení, splnění je vždy stejné)

### Definice (Logicky pravdivá formule predikátové logiky)

Formule  $A$  je *validní (logicky pravdivá)* ( $\models A$ ), když je platná při každé interpretaci daného jazyka.

## Dokazatelnost

### Definice (Axiomy výrokové logiky)

Pro redukovaný jazyk výrokové logiky (po snížení počtu log. spojek na základní ( $\rightarrow$ ,  $\neg$ )) jsou *axiomy výrokové logiky* (schémata axiomů) všechny formule následujících tvarů:

- $(A \rightarrow (B \rightarrow A))$  (A1 - „implikace sebe sama“)
- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$  (A2 - „roznásobení“)
- $(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$  (A3 - „obrácená negace implikace“)

### Definice (Odvozovací pravidlo výrokové logiky)

Výroková logika má *jediné odvozovací pravidlo* – *modus ponens*:

$$\frac{A, A \rightarrow B}{B}$$

**Definice** (*Důkaz ve výrokové logice*)

Důkaz  $A$  je konečná posloupnost formulí  $A_1, \dots, A_n$ , jestliže  $A_n = A$  a pro každé  $i = 1, \dots, n$  je  $A_i$  buď axiom, nebo je odvozená z předchozích pravidlem modus ponens. Existuje-li důkaz formule  $A$ , pak je tato *dokazatelná* ve výrokové logice (je větou výrokové logiky -  $\vdash A$ ).

**Definice** (*Důkaz z předpokladů*)

Důkaz formule  $A$  z předpokladů je posloupnost formulí  $A_1, \dots, A_n$  taková, že  $A_n = A$  a  $\forall i \in \{1, \dots, n\}$  je  $A_i$  axiom, nebo prvek množiny předpokladů  $T$ , nebo je odvozena z přechozích pravidlem modus ponens. Existuje-li důkaz  $A$  z  $T$ , pak  $A$  je *dokazatelná z  $T$*  -  $T \vdash A$ .

**Věta** (*o dedukci ve výrokové logice*)

Pro  $T$  množinu formulí a formule  $A, B$  platí:

$$T \vdash A \rightarrow B \text{ právě když } T, A \vdash B$$

**Definice**

Množina formulí  $T$  je *sporná*, pokud je z předpokladů  $T$  dokazatelná libovolná formule, jinak je  $T$  *bezesporná*.  $T$  je *maximální bezesporná* množina, pokud je  $T$  bezesporná a navíc jediná její bezesporná nadmnožina je  $T$  samo. Množina všech formulí dokazatelných z  $T$  se značí  $Con(T)$ .

**Věta** (*Lindenbaumova*)

Každou bezespornou množinu formulí výrokové logiky  $T$  lze rozšířit na maximální bezespornou  $S$ ,  $T \subset S$ .

**Věta** (*o bezespornosti a splnitelnosti*)

Množina formulí výrokové logiky je bezesporná, právě když je splnitelná.

**Věta** (*Věta o úplnosti výrokové logiky*)

Je-li  $T$  množina formulí a  $A$  formule, pak platí:

1.  $T \vdash A$  právě když  $T \models A$
2.  $\vdash A$  právě když  $\models A$  ( $A$  je větou výrokové logiky, právě když je tautologie)

Tedy výroková logika je bezesporná a jsou v ní dokazatelné právě tautologie.

**Věta** (*o kompaktnosti*)

Množina formulí výrokové logiky je splnitelná, právě když je splnitelná každá její konečná podmnožina.

**Definice** (*Formální systém predikátové logiky*)

Pracujeme s redukovaným jazykem (jen s log. spojkami  $\neg, \rightarrow$  a jen s kvantifikátorem  $\forall$ ). *Schémata Axiomů predikátové logiky* vzniknou z těch ve výrokové logice prostým dosazením libovolných formulí predikátové logiky za výrokové proměnné. *Modus ponens* platí i v pred. logice. Další axiomy a pravidla:

- *schéma(axiom) specifikace*:  $(\forall x)A \rightarrow A_x[t]$

- *schéma přeskočku*:  $(\forall x)(A \rightarrow B) \rightarrow (A \rightarrow (\forall x)B)$ , pokud proměnná  $x$  nemá volný výskyt v  $A$ .
- *pravidlo generalizace*:  $\frac{A}{(\forall x)A}$

Toto je formální systém pred. logiky *bez rovnosti*. S rovností přibývá symbol  $=$  a další tři axiomy.

**Poznámka** (*Vlastnosti formulí predikátové logiky*)

1. Je-li  $A'$  instance formule  $A$ , pak jestliže platí  $\vdash A$ , platí i  $\vdash A'$ . (*Věta o instancích*)
2. Je-li  $A'$  uzávěr formule  $A$ , pak  $\vdash A$  platí právě když  $\vdash A'$ . (*Věta o uzávěru*)

**Věta** (*o dedukci v predikátové logice*)

Nechť  $T$  je množina formulí pred. logiky,  $A$  je uzavřená formule a  $B$  lib. formule, potom  $T \vdash A \rightarrow B$  právě když  $T, A \vdash B$ .

**Definice** (*Teorie, model*)

Pro nějaký jazyk  $L$  prvního řádu je množina  $T$  formulí tohoto jazyka *teorie prvního řádu*. Formule z  $T$  jsou *speciální axiomy* teorie  $T$ . Pro interpretaci  $\mathcal{M}$  jazyka  $L$  je  $\mathcal{M}$  *model teorie  $T$*  ( $\mathcal{M} \models T$ ), pokud jsou všechny speciální axiomy  $T$  pravdivé v  $\mathcal{M}$ . Formule  $A$  je *sémantickým důsledkem  $T$* :  $T \models A$ , jestliže je pravdivá v každém modelu teorie  $T$ .

**Věta** (*o korektnosti*)

Je-li  $T$  teorie prvního řádu a  $A$  formule, potom platí:

1. Jestliže  $T \vdash A$ , potom  $T \models A$ .
2. Speciálně jestliže  $\vdash A$ , potom  $\models A$ .

**Věta** (*o úplnosti v predikátové logice*)

Nechť  $T$  je teorie s jazykem prvního řádu  $L$ . Je-li  $A$  lib. formule jazyka  $L$ , pak platí:

1.  $T \vdash A$  právě když  $T \models A$
2.  $T$  je bezesporná, právě když má model.

**Definice** (*Úplná teorie*)

Teorie  $T$  s jazykem  $L$  prvního řádu je *úplná*, je-li bezesporná a pro libovolnou uzavřenou formuli  $A$  je jedna z formulí  $A, \neg A$  dokazatelná v  $T$ .

**Věta** (*o kompaktnosti*)

Teorie  $T$  s jazykem  $L$  prvního řádu má model, právě když každý její konečný fragment  $T' \subset T$  má model. Tj. pro libovolnou formuli  $A$  jazyka  $L$  platí:  $T \models A$  právě když  $T' \models A$  pro nějaký konečný fragment  $T' \subset T$ .

## Rozhodnutelnost

### Definice (Rekurzivní funkce a množina)

Rekurzivní funkce jsou všechny funkce popsatelné jako  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ , kde  $k \geq 1$ , tedy všechny „algoritmicky vyčíslitelné“ funkce. Množina přirozených čísel je rekurzivní množina (rozhodnutelná množina), pokud je rekurzivní její charakteristická funkce (to je funkce, která určí, které prvky do množiny patří).

### Definice (Spočetný jazyk, kód formule)

Spočetný jazyk je jazyk, který má nejvýš spočetně mnoho speciálních symbolů. Pro spočetný jazyk, kde lze efektivně (rekurzivní funkcí) očíslovat jeho speciální symboly, lze každé jeho formuli  $A$  přiřadit její kód formule - přír. číslo  $\#A$ .

### Věta (Churchova o nerozhodnutelnosti predikátové logiky)

Pokud spočetný jazyk  $L$  prvního řádu obsahuje alespoň jednu konstantu, alespoň jeden funkční symbol arity  $k > 0$  a pro každé přirozené číslo spočetně mnoho predikátových symbolů, potom množina  $\{\#A \mid A \text{ je uzavřená formule a } L \models A\}$  není rozhodnutelná.

### Věta (o nerozhodnosti predikátové logiky)

Nechť  $L$  je jazyk prvního řádu bez rovnosti a obsahuje alespoň 2 binární predikáty. Potom je predikátová logika (jako teorie) s jazykem  $L$  nerozhodnutelná.

### Definice (Tři popisy aritmetiky)

Je dán jazyk  $L = \{0, S, +, \cdot, \leq\}$ .

- Robinsonova aritmetika - "Q" s jazykem  $L$  má 8 následujících axiomů:

1.  $S(x) \neq 0$
2.  $S(x) = S(y) \rightarrow x = y$
3.  $x \neq 0 \rightarrow (\exists y)(x = S(y))$
4.  $x + 0 = x$
5.  $x + S(y) = S(x + y)$
6.  $x \cdot 0 = 0$
7.  $x \cdot S(y) = (x \cdot y) + x$
8.  $x \leq y \leftrightarrow (\exists z)(z + x = y)$

Poznámka: Někdy, pokud není potřeba definovat uspořádání, se poslední axiom spolu se symbolem „ $\leq$ “ vypouští.

- Peanova aritmetika - "P" má všechny axiomy Robinsonovy kromě třetího, navíc má Schéma(axiomů) indukce - pro formuli  $A$  a proměnnou  $x$  platí:  $A_x[0] \rightarrow \{(\forall x)(A \rightarrow A_x[S(x)]) \rightarrow (\forall x)A\}$ .
- Úplná aritmetika má za axiomy všechny uzavřené formule pravdivé v  $\mathbb{N}$ , je-li  $\mathbb{N}$  standardní model aritmetiky - „pravdivá aritmetika“. Teorie modelu  $\mathbb{N}$  je množina  $Th(\mathbb{N}) = \{A \mid A \text{ je uzavřená formule a } \mathbb{N} \models A\}$ .

Platí:  $Q \subseteq P \subseteq Th(\mathbb{N})$ .  $Q$  má konečně mnoho axiomů, je tedy rekurzivně axiomatizovatelná.  $P$  má spočetně mnoho axiomů, kódy axiomů schématu indukce tvoří rekurzivní množinu.  $Th(\mathbb{N})$  není rekurzivně axiomatizovatelná.

**Definice** (*Množina kódů vět teorie*)

Pro  $T$  teorii s jazykem aritmetiky definujeme množinu kódů vět teorie  $T$  jako  $Thm(T) = \{\#A \mid A \text{ je uzavřená formule a } T \vdash A\}$ .

**Definice** (*Rozhodnutelná teorie*)

Teorie  $T$  s jazykem aritmetiky je *rozhodnutelná*, pokud je množina  $Thm(T)$  rekurzivní. V opačném případě je  $T$  *nerozhodnutelná*.

**Věta** (*Churchova o nerozhodnutelnosti aritmetiky*)

Každé bezesporné rozšíření Robinsonovy aritmetiky  $Q$  je nerozhodnutelná teorie.

**Věta** (*Gödel-Rosserova o neúplnosti aritmetiky*)

Žádné bezesporné a rekurzivně axiomatizovatelné rozšíření Robinsonovy aritmetiky  $Q$  není úplná teorie.

### 1.3 Normální tvary výrokových formulí, prenexní tvary formulí predikátové logiky

**Poznámka** (*Vlastnosti log. spojek*)

Platí:

1.  $A \wedge B \vdash A$ ;  $A, B \vdash A \wedge B$
2.  $A \leftrightarrow B \vdash A \rightarrow B$ ;  $A \rightarrow B, B \rightarrow A \vdash A \leftrightarrow B$
3.  $\wedge$  je idempotentní, komutativní a asociativní.
4.  $\vdash (A_1 \rightarrow \dots (A_n \rightarrow B) \dots) \leftrightarrow ((A_1 \wedge \dots \wedge A_n) \rightarrow B)$
5. DeMorganovy zákony:  $\vdash \neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$ ;  $\vdash \neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$
6.  $\vee$  je monotonní ( $\vdash A \rightarrow A \vee B$ ), idempotentní, komutativní a asociativní.
7.  $\vee$  a  $\wedge$  jsou navzájem distributivní.

**Věta** (*o ekvivalenci ve výrokové logice*)

Jestliže jsou podformule  $A_1 \dots A_n$  formule  $A$  ekvivalentní s  $A'_1 \dots A'_n$  ( $\vdash A'_i \leftrightarrow A_i$ ) a  $A'$  vytvořím nahrazením  $A'_i$  místo  $A_i$ , je i  $A$  ekvivalentní s  $A'$ . (Důkaz indukci podle složitosti formule, rozbořem případů  $A_i$  tvaru  $\neg B, B \rightarrow C$ )

**Lemma** (*o důkazu rozbořem případů*)

Je-li  $T$  množina formulí a  $A, B, C$  formule, pak  $T, (A \vee B) \vdash C$  platí právě když  $T, A \vdash C$  a  $T, B \vdash C$ .

**Definice** (*Normální tvary*)

Výrokovou proměnnou nebo její negaci nazveme *literál*. *Klauzule* budiž disjunkce několika literálů. *Formule v normálním konjunktivním tvaru (CNF)* je konjunkce klauzulí. *Formule v disjunktivním tvaru (DNF)* je disjunkce konjunktí literálů.

**Věta** (*o normálních tvarech*)

Pro každou formuli  $A$  lze sestavit formule  $A_k, A_d$  v konjunktivním, resp. disjunktivním tvaru tak, že  $\vdash A \leftrightarrow A_d, \vdash A \leftrightarrow A_k$ . (Důkaz z DeMorganových formulí a distributivity, indukci podle složitosti formule)

## Prenexní tvary formulí predikátové logiky

**Věta** (o ekvivalenci v predikátové logice)

Nechť formule  $A'$  vznikne z  $A$  nahrazením některých výskytů podformulí  $B_1, \dots, B_n$  po řadě formulí  $B'_1, \dots, B'_n$ . Je-li  $\vdash B_1 \leftrightarrow B'_1, \dots, \vdash B_n \leftrightarrow B'_n$ , potom platí i  $\vdash A \leftrightarrow A'$ .

**Definice** (Prenexní tvar)

Formule predikátové logiky  $A$  je v *prenexním tvaru*, je-li

$$A \equiv (Q_1x_1)(Q_2x_2) \dots (Q_nx_n)B,$$

kde  $n \geq 0$  a  $\forall i \in \{1, \dots, n\}$  je  $Q_i \equiv \forall$  nebo  $\exists$ ,  $B$  je otevřená formule a kvantifikované proměnné jsou navzájem různé.  $B$  je *otevřené jádro*  $A$ , část s kvantifikátory je *prefix*  $A$ .

**Definice** (Varianta formule predikátové logiky)

Formule  $A'$  je *varianta*  $A$ , jestliže vznikla z  $A$  postupným nahrazením podformulí  $(Qx)B$  (kde  $Q$  je  $\forall$  nebo  $\exists$ ) formulí  $(Qy)B_x[y]$  a  $y$  není volná v  $B$ . Podle *věty o variantách* je varianta s původní formulí ekvivalentní.

**Lemma** (o prenexních operacích)

Pro převod formulí do prenexního tvaru se používají tyto operace (výsledná formule je s původní ekvivalentní). Pro podformule  $B, C$ , kvantifikátor  $Q$  a proměnnou  $x$ :

1. podformuli lze nahradit nějakou její variantou
2.  $\vdash \neg(Qx)B \leftrightarrow (\overline{Q}x)\neg B$
3.  $\vdash (B \rightarrow (Qx)C) \leftrightarrow (Qx)(B \rightarrow C)$ , pokud  $x$  není volná v  $B$
4.  $\vdash ((Qx)B \rightarrow C) \leftrightarrow (\overline{Q}x)(B \rightarrow C)$ , pokud  $x$  není volná v  $C$
5.  $\vdash ((Qx)B \wedge C) \leftrightarrow (Qx)(B \wedge C)$ , pokud  $x$  není volná v  $C$
6.  $\vdash ((Qx)B \vee C) \leftrightarrow (Qx)(B \vee C)$ , pokud  $x$  není volná v  $C$

**Věta** (o prenexních tvarech)

Ke každé formuli  $A$  predikátové logiky lze sestrojít ekvivalentní formuli  $A'$ , která je v prenexním tvaru. (Důkaz: indukcí podle složitosti formule a z prenexních operací, někdy je nutné přejmenovat volné proměnné)

## 1.4 Automaty – Chomského hierarchie, třídy automatů a gramatik, determinismus a nedeterminismus.

TODO: přesunout věty a popisy gramatik do sekce o Chomského hierarchii, za definici gramatik (abychom vedeli s čím operujeme ;))

## Třídy automatů a gramatik

### Definice (Konečný automat)

Konečný automat je pětice  $A = (Q, X, \delta, q_0, F)$ , kde  $Q$  je stavový prostor (množina všech možných stavů),  $X$  je abeceda (množina symbolů),  $\delta$  je přechodová funkce  $\delta : Q \times X \rightarrow Q$ ,  $q_0 \in Q$  je poč. stav a  $F \subseteq Q$  množina koncových stavů.

### Definice

Slovo  $w$  je posloupnost symbolů v abecedě  $X$ . Jazyk  $L$  je množina slov, tedy  $L \subseteq X^*$ , kde  $X^*$  je množina všech posloupností symbolů abecedy  $X$ .  $\lambda$  je prázdná posloupnost symbolů. Rozšířená přechodová funkce je  $\delta^* : Q \times X^* \rightarrow Q$  - tranzitivní uzávěr  $\delta$ . Jazyk rozpoznávaný konečným automatem - regulární jazyk je  $L(A) = \{w | w \in X^*, \delta^*(q_0, w) \in F\}$ . Pravá kongruence je taková relace ekvivalence na  $X^*$ , že  $\forall u, v, w \in X^* : u \sim v \Rightarrow uw \sim vw$ . Je konečného indexu, jestliže  $X^*/\sim$  má konečný počet tříd.

### Věta (Nerodova)

Jazyk  $L$  nad konečnou abecedou  $X$  je rozpoznatelný kon. automatem, právě když existuje pravá kongruence konečného indexu na  $X^*$  tak, že  $L$  je sjednocením jistých tříd rozkladu  $X^*/\sim$ .

### Věta (Pumping (iterační) lemma)

Pro jazyk rozpoznatelný kon. automatem  $L$  existuje  $n \in \mathbb{N}$  tak, že libovolné slovo  $z \in L$ ,  $|z| \geq n$  lze psát jako  $uvw$ , kde  $|uv| \leq n$ ,  $|v| \geq 1$  a  $\forall i \geq 0 : uv^i w \in L$ .

### Definice

Dva automaty jsou ekvivalentní, jestliže přijímají stejný jazyk. Homomorfismus (isomorfismus) automatů je zobrazení, zachovávající poč. stav, přech. funkci i konc. stavy (+ prosté a na). Pokud existuje isomorfismus automatů  $A \rightarrow B$ , pak jsou tyto dva ekvivalentní (jen 1 implikace!). Dosažitelný stav  $q$  -  $\exists w \in X^* : \delta^*(q_0, w) = q$ . Relace ekvivalence je automatovou kongruencí, pokud zachovává konc. stavy a přech. funkci. Ke každému automatu existuje redukt - ekvivalentní automat bez nedosažitelných a navzájem ekvivalentních stavů. Ten je určen jednoznačně pro daný jazyk (až na isomorfismus), proto lze zavést normovaný tvar.

### Poznámka (Operace s jazyky)

S jazyky lze provádět množinové operace ( $\cup, \cap$ ), rozdíl ( $\{w | w \in L_1 \& w \notin L_2\}$ ), doplněk ( $\{w | w \notin L\}$ ), dále zřetězení ( $L_1 \cdot L_2 = \{uv | u \in L_1, v \in L_2\}$ ), mocniny ( $L^0 = \lambda, L^{i+1} = L^i \cdot L$ ), iterace ( $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$ ), otočení  $L^R$ , levý (i pravý) kvocient ( $K \setminus L = \{v | uv \in L, u \in K\}$ ) a derivace (kvocienty podle jednoslovného jazyka). Třída jazyků rozpoznatelných konečnými automaty je na tyto operace uzavřená.

### Definice (Regulární jazyky)

Třída regulárních jazyků nad abecedou  $X$  je nejmenší třída, která obsahuje  $\emptyset, \forall x \in X$  obsahuje  $x$  a je uzavřená na sjednocení, iteraci a zřetězení.

### Věta (Kleenova)

Jazyk je regulární, právě když je rozpoznatelný konečným automatem.

### Definice (Regulární výrazy)

Regulární výrazy nad abecedou  $X = x_1, \dots, x_n$  jsou nejmenší množina slov v abecedě  $x_1, \dots, x_n, \emptyset, \lambda, +, \cdot, *, (, )$ , která obsahuje výrazy  $\emptyset$  a  $\lambda$  a  $\forall i$  obsahuje  $x_i$  a je uzavřená na sjednocení (+), zřetězení ( $\cdot$ ) a iterace (\*). Hodnota reg. výrazu  $a$  je reg. jazyk  $[a]$ , lze takto reprezentovat každý reg. jazyk.

### Definice (Dvoucestné konečné automaty)

Dvoucestný konečný automat je pětice  $(Q, X, \delta, q_0, F)$ , kde oproti kon. automatu je  $\delta : Q \times X \rightarrow Q \times \{-1, 0, 1\}$  (tj. pohyb čtecí hlavy). Přijímá slovo, pokud výpočet začal vlevo v poč. stavu a čtecí hlava opustila slovo  $w$  vpravo v konc. stavu (mimo slovo končí výpočet okamžitě).

### Poznámka

Jazyky přijímané dvoucestnými automaty jsou regulární - každý dvoucestný automat lze převést na (nedeterministický) konečný automat.

### Definice (Zásobníkové automaty)

Zásobníkový automat je sedmice  $M = (Q, X, Y, \delta, q_0, Z_0, F)$ , kde proti konečným automatům je  $Y$  abeceda pro symboly na zásobníku,  $Z_0$  počáteční symbol na zásobníku a funkce instrukcí  $\delta : Q \times (X \cup \{\lambda\}) \times Y \rightarrow \mathcal{P}(Q \times Y^*)$ . Je z principu nedeterministický; vždy se nahrazuje vrchol zásobníku, nechte ale pokaždé vstupní symboly. Instrukci  $(p, a, Z) \rightarrow (q, w)$  lze vykonat, pokud je automat ve stavu  $p$ , na zásobníku je  $Z$  a na vstupu  $a$ . Vykonání instrukce znamená změnu stavu, pokud  $a \neq \lambda$ , tak i posun čtecí hlavy a odebrání  $Z$  ze zásobníku, kam se vloží  $w$  (prvním písmenem nahoru). Výpočet končí buď přečtením slova, nebo v případě, že pro danou situaci není definována instrukce (Situace zás. automatu je trojice  $(p, u, v)$ , kde  $p \in Q$ ,  $u$  je nepřečtený zbytek slova a  $v$  celý zásobník).

Přijímat slovo je možné buď koncovým stavem (slovo je přečteno a automat v konc. stavu), nebo zásobníkem (slovo je přečteno a zásobník prázdný - konc. stavy jsou v takovém případě nezájímavé -  $F = \emptyset$ ).

### Poznámka

Pro zás. automat přijímající konc. stavem vždy existuje ekvivalentní automat ( $L(A_1) = L(A_2)$ ) přijímající zásobníkem a naopak.

### Věta

Každý bezkontextový jazyk je rozpoznáván zásobníkovým automatem, přijímajícím prázdným zásobníkem. Stejně pro každý zásobníkový automat existuje bezkontextová gramatika, která generuje jazyk jím přijímaný.

### Poznámka (Vlastnosti bezkontextových gramatik)

Bezkontextová gramatika je redukovaná, pokud  $\forall X \in V_N$  existuje terminální slovo  $w \in V_T^*$  tak, že  $X \Rightarrow^* w$  a navíc  $\forall X \in V_N, X \neq S$  existují slova  $u, v$  tak, že  $S \Rightarrow^* uXv$ . Ke každé bezkontextové gramatice lze sestavit ekvivalentní redukovanou.

Pro každé terminální slovo v bezkontextové gramatice existují derivace, které se liší jen pořadím použití pravidel (a prohozením některých pravidel dostanu stejné terminální slovo), proto lze zavést *levé (pravé) derivace* - tj. *kanonické* derivace. Pokud  $X \Rightarrow^* w$ , pak existuje i levá (pravá) derivace. Znázornění průběhu derivací je možné určit *derivačním stromem* - určuje jednoznačně pravou/levou derivaci.

Bezkontextová gramatika je *víceznačná* (nejednoznačná), pokud v ní existuje slovo, které má dvě různé levé derivace; jinak je *jednoznačná*. Jazyk je jednoznačný, pokud k němu existuje generující jednoznačná gramatika. Pokud je každá gramatika nějakého jazyka nejednoznačná, je tento *podstatně nejednoznačný*.

**Definice** (*Greibachové normální forma*)

Gramatika je v *Greibachové normální formě*, jsou-li všechna její pravidla ve tvaru  $A \rightarrow au$ , kde  $a \in V_T$  a  $u \in V_N^*$ . Ke každému bezkontextovému jazyku existuje gramatika v G. normální formě tak, že  $L(G) = L \setminus \{\lambda\}$ . Každou bezkontextovou gramatiku lze převést do G. normální formy.

**Poznámka** (*Úpravy bezkontextových gramatik*)

Spojením více pravidel ( $A \rightarrow uBv, B \rightarrow w_1, \dots, B \rightarrow w_k$  se převede na  $A \rightarrow uw_1v | \dots | uw_kv$ ) dostanu ekvivalentní gramatiku. Stejně tak odstraněním levé rekurze (převod přes nový neterminál).

**Definice** (*Chomského normální forma*)

Pro gramatiku v *Chomského normální formě* jsou všechna pravidla tvaru  $X \rightarrow YZ$  nebo  $X \rightarrow a$ , kde  $X, Y, Z \in V_N$ ,  $a \in V_T$ . Ke každému bezkontextovému jazyku  $L$  existuje gramatika  $G$  v Chomského normální formě tak, že  $L(G) = L \setminus \{\lambda\}$

**Poznámka** (*Vlastnosti třídy bezkontextových jazyků*)

Třída bezkontextových jazyků je uzavřená na sjednocení, zrcadlení, řetězení, iteraci a pozitivní iteraci, substituci a homomorfismus, inverzní homomorfismus a kvocient s regulárním jazykem. Není uzavřená na průnik a doplněk.

**Definice** (*Dyckův jazyk*)

*Dyckův jazyk* je definován nad abecedou  $a_1, a'_1, \dots, a_n, a'_n$  gramatikou

$$S \rightarrow \lambda | SS | a_1 S a'_1 | \dots | a_n S a'_n$$

Je bezkontextový, popisuje správná uzávorkování a lze jím popisovat výpočty zásobníkových automatů, tedy i bezkontextové jazyky.

**Definice** (*Turingův stroj*)

*Turingův stroj* je pětice  $T = (Q, X, \delta, q_0, F)$ , kde  $X$  je abeceda, obsahující symbol  $\varepsilon$  pro prázdné políčko, přechodová funkce  $\delta : (Q \setminus F) \times X \rightarrow Q \times X \times \{-1, 0, 1\}$  popisuje změnu stavu, zápis na pásku a posun hlavy. Výpočet končí, není-li definována žádná instrukce (spec. platí pro  $q \in F$ ). *Konfigurace* Turingova stroje jsou údaje, popisující stav výpočtu - nejmenší souvislá část pásky, obsahující všechny neprázdné buňky a čtenou buňku, vnitřní stav a poloha hlavy. *Krok výpočtu* je  $uqv \vdash wpz$  pro  $u$  část slova vlevo od akt. pozice na pásce,  $v$  od čteného písmena dál a  $q$  stav stroje. *Výpočet* je posloupnost kroků, slovo  $w$  je přijímáno, pokud

$q_0w \vdash^* upv$ ,  $p \in F$ . Jazyky (množiny slov bez  $\varepsilon$ ) přijímané Turingovými stroji jsou *rekurzivně spočetné*.

### Věta

Každý jazyk typu 0 (s gramatikou s obecnými pravidly) je rekurzivně spočetný.

## Chomského hierarchie

### Definice (Přepisovací systém)

*Přepisovací (produkční) systém* je dvojice  $R = (V, P)$ , kde  $V$  je konečná abeceda a  $P$  množina přepisovacích pravidel (uspořádaných dvojic prvků z  $V^*$ ). Slovo  $w$  se *přímo přepíše* na  $z$  ( $w \Rightarrow z$ ), pokud  $\exists u, v, x, y \in V : w = xuy, z = xvy, (u, v) \in P$ . Derivace (*odvození*) je zřetězení několika přímých přepsání.

### Definice (Generativní gramatika)

*Generativní gramatika* je čtveřice  $G = (V_N, V_T, S, P)$ , kde  $V_N$  je množina neterminálních symbolů,  $V_T$  množina terminálních symbolů,  $S$  startovací symbol ( $S \in V_N$ ) a  $P$  množina pravidel. *Jazyk generovaný gramatikou* je  $L(G) = \{w \in V_T^*, S \Rightarrow^* w\}$ .

### Definice (Chomského hierarchie)

*Chomského hierarchie* je rozdělení gramatik do 4 tříd podle omezení na pravidla:

- $G_0$  (*Rekurzivně spočetné jazyky*) mohou mít obecná pravidla.
- $G_1$  (*Kontextové jazyky/gramatiky*) - jen pravidla tvaru  $\alpha X \beta \rightarrow \alpha \omega \beta$ , kde  $X \in V_N$  a  $\alpha, \beta, \omega \in (V_N \cup V_T)^*$ , navíc  $|\omega| > 0$ . Může obsahovat i pravidlo  $S \rightarrow \lambda$ , ale pak se  $S$  nesmí vyskytovat na pravé straně žádného pravidla.
- $G_2$  (*Bezkontextové jazyky/gramatiky*) - jen pravidla tvaru  $X \rightarrow \omega$ , kde  $X \in V_N$ ,  $\omega \in (V_N \cup V_T)^*$ .
- $G_3$  (*Regulární jazyky/pravé lineární gramatiky*) - jen pravidla typu  $X \rightarrow \omega Y$  a  $X \rightarrow \omega$ , kde  $\omega \in V_T^*$  a  $X, Y \in V_N$ .

Definuje uspořádání tříd jazyků podle inkluze, tedy  $\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$ .

### Poznámka

S  $\mathcal{L}_1 \supset \mathcal{L}_2$  nastává problém, protože bezkontextové gramatiky umožňují pravidla tvaru  $X \rightarrow \lambda$ . Řešením je převod na *nevypouštějící bezkontextové gramatiky* - takové bezkontextové gramatiky, které nemají pravidla typu  $X \rightarrow \lambda$ .

### Věta (o nevypouštějících bezkontextových gramatikách)

Ke každé bezkontextové  $G$  existuje nevypouštějící bezkontextová  $G_0$  tak, že

$$L(G_0) = L(G) \setminus \{\lambda\}$$

Je-li  $\lambda \in L(G)$ , pak  $\exists G_1$ , t.ž.  $L(G_1) = L(G)$  a jediné pravidlo v  $G_1$  s  $\lambda$  na pravé straně je  $S \rightarrow \lambda$  a  $S$  není v  $G_1$  na pravé straně žádného pravidla.

**Poznámka** (*Lineární gramatiky*)

Pro každou gramatiku typu G3 lze sestavit konečný automat, který přijímá právě jazyk jí generovaný, stejně tak pro každý konečný automat lze sestavit gramatiku G3. Levé lineární gramatiky také generují regulární jazyky, díky uzavřenosti na reverzi. *Lineární gramatiky*, s pravidly typu  $X \rightarrow uYv, X \rightarrow w$ , kde  $X, Y \in V_N, u, v, w \in V_T^*$ , generují lineární jazyky - silnější než regulární jazyky.

**Definice** (*Separovaná a nevypouštějící gramatika*)

*Separovaná gramatika* je gramatika (obecně libovolné třídy), obsahující pouze pravidla tvaru  $\alpha \rightarrow \beta$ , kde buď  $\alpha, \beta \in V_N^+$ , nebo  $\alpha \in V_N$  a  $\beta \in V_T \cup \{\lambda\}$ . *Nevypouštějící (monotónní) gramatika* (také se neomezuje na konkrétní třídu) je taková, že pro každé pravidlo  $u \rightarrow v$  platí  $|u| \leq |v|$ .

**Poznámka** (*Kontextové gramatiky*)

Ke každé kontextové gramatice lze sestavit ekvivalentní separovanou. Ke každé monotónní gramatice lze nalézt ekvivalentní kontextovou.

**Determinismus a nedeterminismus****Definice** (*Nedeterministický konečný automat*)

*Nedeterministický konečný automat* je pětice  $(Q, X, \delta, S, F)$ , kde  $Q$  je mn. stavů,  $X$  abeceda,  $F$  mn. konc. stavů,  $S$  množina počátečních stavů a  $\delta : Q \times X \rightarrow \mathcal{P}(Q)$  je přechodová funkce. Slovo  $w$  je takovým automatem přijímáno, pokud existuje posloupnost stavů  $\{q_i\}_{i=1}^n$  tak, že  $q_1 \in S, q_{i+1} \in \delta(q_i, x_i), q_{n+1} \in F$ .

**Poznámka**

Pro každý nedeterministický konečný automat  $A$  lze sestavit deterministický kon. automat  $B$  tak, že jimi přijímané jazyky jsou ekvivalentní (může to znamenat exponenciální nárůst počtu stavů).

**Definice** (*Deterministický zásobníkový automat*)

*Deterministický zásobníkový automat* je  $M = (Q, X, Y, \delta, q_0, Z_0, F)$  takové, že  $\forall p \in Q, \forall a \in (X \cup \{\lambda\}), \forall Z \in Y$  platí  $|\delta(p, a, Z)| \leq 1$  a navíc pokud pro nějaké  $p, Z$  je  $\delta(p, \lambda, Z) \neq \emptyset$ , pak  $\forall a \in X$  je  $\delta(p, a, Z) = \emptyset$ .

**Poznámka**

Deterministický zásobníkový automat je „slabší“ než nedeterministický, rozpoznává *deterministické bezkontextové jazyky* koncovým stavem a *bezprefixové bezkontextové jazyky* prázdným zásobníkem (takové jazyky, kde  $u \in L(M) \Rightarrow \forall w \in X^* : uw \notin L(M)$ ) - když se poprvé zásobník automatu vyprázdní, výpočet určitě končí.

Bezprefixové bezkontextové jazyky jsou vždy deterministické, opačně to neplatí. Deterministický bezkontextový jazyk lze na bezprefixový převést zřetězením s dalším symbolem, který není v původní abecedě.

Regulární jazyky a bezprefixové bezkontextové jazyky jsou neporovnatelné množiny.

**Definice** (*Nedeterministický Turingův stroj*)

*Nedet. Turingův stroj* je pětice  $T = (Q, X, \delta, q_0, F)$ , kde oproti deterministickým je  $\delta : (Q \setminus F) \times X \rightarrow \mathcal{P}(Q \times X \times \{-1, 0, 1\})$ . Přijímá slovo  $w$ , pokud existuje nějaký výpočet  $q_0 w \vdash^* upv$  tak, že  $p \in F$ .

**Poznámka**

Nedeterministické Turingovy stroje přijímají právě rekurzivně spočetné jazyky, tj. nejsou silnější než deterministické. Výpočty nedet. stroje lze totiž díky nekonečnosti pásky simulovat deterministickým (např. prohledáváním do šířky).

**Definice** (*Lineárně omezený automat*)

*Lineárně omezený automat* je nedeterministický Turingův stroj s omezenou páskou (např. symboly  $l$  a  $r$ , které nelze přepsat ani se dostat mimo jejich rozmezí). Slovo je přijímáno, pokud  $q_0 lwr \vdash^* upv$ , kde  $p \in F$ . Prostor výpočtu je omezen délkou vstupního slova. Lineárně omezené automaty přijímají právě kontextové jazyky.

**Poznámka** (*Rozhodnutelnost*)

Turingův stroj může nepřijmout slovo buď skončením výpočtu v nekonečném stavu, nebo pokud výpočet nikdy neskončí. Turingův stroj *rozhoduje jazyk*  $L$ , když přijímá právě slova tohoto jazyka a pro libovolné slovo je jeho výpočet konečný. Takové jazyky se nazývají *rekurzivní*.

Problém zastavení výpočtu Turingova stroje je algoritmicky nerozhodnutelný (kvůli možnosti jeho simulace jiným Turingovým strojem). Pro bezkontextové jazyky je algoritmicky rozhodnutelné, zda dané slovo patří do jazyka. Pro bezkontextovou gramatiku nelze algoritmicky rozhodnout, zda  $L(G) = X^*$ . Pro dvě kontextové gramatiky je nerozhodnutelné, zda jejich jazyky mají neprázdný průnik.

## 2 Algoritmy a datové struktury

**Požadavky**

- Časová složitost algoritmů, složitost v nejhorším a průměrném případě
- Třídy složitosti P a NP, převoditelnost, NP-úplnost
- Binární vyhledávací stromy, vyvažování, haldy
- Hašování
- Sekvenční třídění, porovnávací algoritmy, přihrádkové třídění, třídící sítě
- Grafové algoritmy - prohledávání do hloubky a do šířky, souvislost, topologické třídění, nejkratší cesta, kostra grafu
- Tranzitivní uzávěr
- Algoritmy vyhledávání v textu
- Algebraické algoritmy - DFT, Euklidův algoritmus
- Základy kryptografie, RSA, DES

## 2.1 Časová složitost algoritmů, složitost v nejhorším a průměrném případě

**Definice** (*časová složitost*)

**Časovou složitostí** algoritmu rozumíme závislost jeho časových nároků na velikosti konkrétních vstupních dat. Analogicky se definuje i **paměťová složitost**. Dobu zpracování úlohy o velikosti  $n$  značíme  $T(n)$

Časovou složitost často zkoumáme z několik hledisek:

- **v nejhorším případě** – maximální počet operací pro nějaká data,
- **v nejlepším případě** – minimální počet operací pro nějaká data,
- **v průměrném (očekávaném) případě** – průměr pro všechna možná vstupní data (někdy též střední hodnota náhodné veličiny  $T(n)$ ).

**Poznámka**

Jako jednu „operaci“, nebo-li *krok algoritmu* rozumíme jednu elementární operaci nějakého abstraktního stroje (např. Turingova stroje), proveditelnou v konstantním čase. Intuitivně je možné chápat to jako několik operací počítače, které dohromady netrvají více, než nějakou pevně danou dobu.

**Poznámka**

Časová složitost problému je rovna složitosti nejlepšího algoritmu řešícího daný problém.

## Asymptotická složitost

### Definice

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

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

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

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

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

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

### Poznámka

Asymptotická složitost zkoumá chování algoritmů na „velkých“ datech a dle jejich složitosti je zařazuje do skupin (polynomiální, exponenciální... ). Při zkoumání se zanedbávají aditivní a multiplikativní konstanty.

## Amortizovaná složitost

### Definice (*Amortizovaná složitost*)

*Amortizovaná časová složitost* počítá průměrný čas na jednu operaci při provedení posloupnosti operací. Používá se typicky pro počítání časové složitosti operací nad datovými strukturami. Dává realističtější horní odhad složitosti posloupnosti operací, než počítání s nejhorsím případem u každé operace.

### Agregační metoda

Spočítáme (nejhorší možný) čas  $T(n)$  pro posloupnost operací. Amortizovaná cena jedné operace je potom  $\frac{T(n)}{n}$ .

### Účetní metoda

Od každé operace „vybereme“ určitý „obnos“, ze kterého „zaplatíme“ za danou operaci a pokud něco zbude, dáme to na účet. Pokud je operace dražší než kolik je její obnos, tak potřebný rozdíl vybereme z účtu. Zůstatek na účtu musí být stále nezáporný. Pokud uspějeme, tak „obnos“ = amortizovaná cena jedné operace.

### Poznámka

Jde o to, že některá operace může trvat krátce, ale „rozháže“ datovou strukturu, takže následující operace potřebují víc času. Nebo naopak trvá dlouho a datovou strukturu „uspořádá“, takže ostatní operace jsou kratší.

## 2.2 Třídy složitosti P a NP, převoditelnost, NP-úplnost

TODO: doplnit, tohle asi nestaci na pohodovou zkousku (aspon podle toho co jsem vnimal, jsa svedek zkouseni u Dr. Yaghoba) – chce to napr. nejaky prikklad prevedeni jako delal Kucera, mozna jeste neco

### Definice

- **Úloha** – Pro dané zadání (vstup, *instanci úlohy*) najít výstup s danými vlastnostmi.
- **Optimalizační úloha** – Pro dané zadání najít optimální (většinou nejmenší nebo největší) výstup s danými vlastnostmi.
- **Rozhodovací problém** – Pro dané zadání odpovědět ANO/NE.

### Definice (třída P)

**Třída složitosti P** (někdy též PTIME) tvoří problémy řešitelné sekvenčními deterministickými algoritmy v polynomiálním čase, tj. jejich časová složitost je  $O(n^k)$ . O algoritmech ve třídě P také říkáme, že jsou **efektivně řešitelné**.

### Definice (třída NP)

**Třída NP** (NPTIME) je třída problémů řešitelných v polynomiálním čase sekvenčními nedeterministickými algoritmy.

### Poznámka

Ví se, že  $P \subseteq NP$ . Neví se však, zda  $P \neq NP$ . Předpokládá se to, ale ještě to nikdo nedokázal.

### Příklady problémů ze třídy NP

- **KLIKA** (úplný podgraf) – Je dán neorientovaný graf  $G$  a číslo  $k$ . Existuje v  $G$  úplný podgraf velikosti aspoň  $k$ ?
- **HK** (Hamiltonovská kružnice) – Je dán neorientovaný graf  $G$ . Existuje v  $G$  Hamiltonovská kružnice?
- **SP** (Součet podmnožiny) – Jsou dána přirozená čísla  $a_1, \dots, a_n, b$ . Existuje podmnožina čísel  $a_1, \dots, a_n$ , jejíž součet je přesně  $b$ ?

### Definice (převody mezi rozhodovacími problémy)

Nechť  $A, B$  jsou dva rozhodovací problémy. Říkáme, že  $A$  je **polynomiálně redukovatelný (převoditelný)** na  $B$ , pokud existuje zobrazení  $f$  z množiny zadání problému  $A$  do množiny zadání problému  $B$  s následujícími vlastnostmi:

- Nechť  $X$  je zadání problému  $A$  a  $Y$  zadání problému  $B$ , takové, že  $f(X) = Y$ . Potom je  $X$  kladné zadání problému  $A$  právě tehdy, když  $Y$  je kladné zadání problému  $B$ .
- Nechť  $X$  je zadání problému  $A$ . Potom je zadání  $f(X)$  problému  $B$  (deterministicky sekvenčně) zkonstruovatelné v polynomiálním čase vzhledem k velikosti  $X$ .

**Definice** (*NP-těžký problém*)

Problém  $B$  je **NP-těžký**, pokud pro libovolný problém  $A$  ze třídy NP platí, že  $A$  je polynomiálně redukovatelný na  $B$ .

**Definice** (*NP-úplný problém*)

Problém je **NP-úplný**, pokud patří do třídy NP a je NP-těžký.

**Důsledky**

- Pokud je  $A$  NP-těžký a navíc je  $A$  polynomiálně redukovatelný na  $B$ , tak je  $B$  taky NP-těžký.
- Pokud existuje polynomiální algoritmus pro nějaký NP-těžký problém, pak existují polynomiální algoritmy pro všechny problémy ve třídě NP.

**Věta** (*Cook-Levin 1971*)

Existuje NP-úplný problém. (Dokázáno pro SAT)

## 2.3 Binární vyhledávací stromy, vyvažování, haldy

**Binární strom****Definice**

*Dynamická množina* je množina prvků (datová struktura), mění se v čase. Každý její prvek je přístupný přes ukazatel a obsahuje:

- *klíč* (jednu položku, typicky hodnotu z lin. uspořádané množiny),
- *ukazatel(e)* na další prvky,
- případně *další data*.

Na takové množině jsou definovány tyto operace:

- *find* - nalezení prvku podle klíče
- *insert* - přidání dalšího prvku
- *delete* - odstranění prvku
- *min, max* - nalezení největšího / nejmenšího prvku
- *succ, pred* - nalezení následujícího / předcházejícího prvku k nějakému předem danému

**Definice**

*Binární strom* je dynamická množina, kde každý prvek (uzel, node) má kromě klíče a příp. dalších dat tři ukazatele na *levého* a *pravého* syna a rodiče. Speciální uzel je *kořen*, který má NULLový ukazatel na rodiče. Ten je v binárním stromě jeden. Uzly, které mají NULLové ukazatele na pravého i levého syna, se nazývají *listy*.

*Podstrom* je část stromu (vybrané prvky), která je sama stromem - např. pokud se jako kořen určí jeden z prvků. *Levý(pravý)* podstrom nějakého prvku je strom,

ve kterém je kořenem levý(pravý) syn tohoto prvku. *Výška stromu* je délka nejdelší cesty od kořenu k listu.

Binární strom je *vyvážený*, pokud max. 1 uzel má jednoho syna (tj. všechny vnitřní uzly kromě až na jeden mají oba syny, listy z definice nemají žádného). Výška vyváženého stromu roste logaritmičsky vzhledem k počtu uzlů. Výška nevyváženého stromu může růst až lineárně vzhledem k počtu prvků (i „spojový seznam“ je platný bin. strom).

## Binární vyhledávací strom

### Definice

*Binární vyhledávací strom* je takový binární strom, ve kterém je jeho struktura určena podle klíče jeho uzlů: pro každý uzel s klíčem hodnoty  $k$  platí, že jeho levý podstrom obsahuje jen uzly s menší hodnotou klíče než  $k$  a jeho pravý podstrom jen uzly s hodnotou klíče větší nebo rovnou  $k$ .

### Algoritmus (*Vyhledávání v bin. stromě*)

```
Find( x - kořen, k - hledaná hodnota klíče ){
  while( x != NULL && k != x->klíč ){
    if ( k < x->klíč )
      x = x->levý_syn;
    else
      x = x->pravý_syn;
  }
  return x;
}
```

Složitost je  $O(h)$  v nejhorsím případě, kde  $h$  je výška stromu (tj. pro nevyvážené stromy až  $O(n)$  kde  $n$  je počet prvků). Asymptotická časová složitost ostatních operací je stejná.

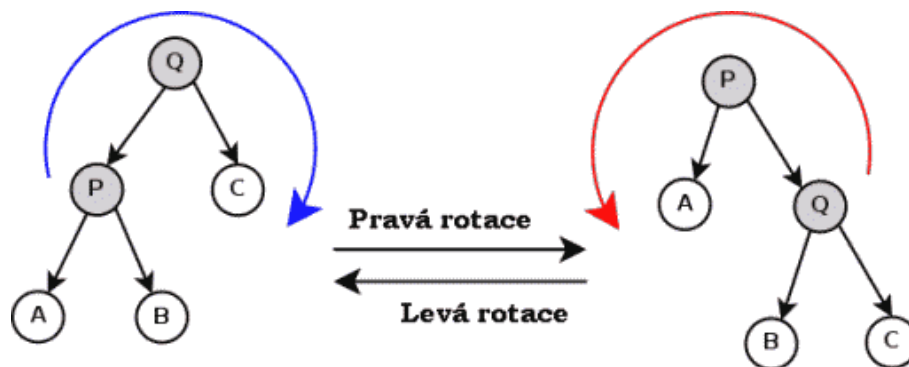
Vložení a vymazání prvku se provádí prostým nalezením místa, kam by se prvek měl vložit (nebo kde už je), a přepojením pointerů.

## Vyvažované vyhledávací stromy

Kvůli zajištění větší rychlosti (menší asymptotické časové složitosti) operací byly vytvořeny speciální druhy binárních vyhledávacích stromů, které jsou průběžně vyvažovány, aby měly max. výšku menší než  $c \cdot \log n$ , kde  $n$  je počet uzlů a  $c$  nějaká konstanta.

### Definice (*Pomocné operace na stromech*)

Pro vyvažování stromů při vkládání a odebírání uzlů se definují pomocné operace: *pravá* a *levá rotace*. Zachovávají vlastnosti bin. vyhledávacích stromů a jsou proveditelné v konstatním čase - jde jen o přepojení uzlů násl. způsobem (pro pravou rotaci na uzlu  $Q$  a levou na  $P$ ):



(Zdroj obrázku: Wikipedia)

### Definice (Červeno-černé stromy)

Červeno-černé stromy jsou binární vyhledávací stromy s garantovanou max. výškou  $O(\log n)$ , kde  $n$  je počet uzlů, tj. operace na nich mohou mít asymptotickou časovou složitost  $O(\log n)$ . Pro jejich popis je nutné definovat *interní uzly* - všechny uzly stromu a *externí uzly* - na (interních) listech (a uzlech s jedním potomkem) uměle přidané NULLové ukazatele (de facto „listy“ červeno-černého stromu). Externí uzly slouží jenom jako abstrakce pro popis stromů, při implementaci se s nimi neoperuje. Červeno-černý strom má tyto čtyři povinné vlastnosti:

1. Každý uzel (externí i interní) má definovanou barvu, a to černou nebo červenou.
2. Každý externí uzel je černý.
3. Každý červený vrchol musí mít oba syny černé.
4. Každá cesta od libovolného vrcholu k listům v jeho podstromě musí obsahovat stejný počet černých uzlů.

Pro červeno-černé stromy se definuje *výška uzlu*  $x$  ( $\mathbf{h}(x)$ ) jako počet uzlů na nedelší možné cestě k listu v jeho podstromě. *Černá výška uzlu* ( $\mathbf{bh}(x)$ ) je počet černých uzlů na takové cestě.

### Věta (Vlastnosti červeno-černých stromů)

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

### Důsledek

Operace hledání (minima, maxima, následníka, ...), které jsou stejné jako u obecných binárních vyhledávacích stromů, mají garantovanou časovou složitost  $O(\log n)$ .

### Algoritmus (Vkládání a odebírání uzlů v červeno-černých stromech)

Obě operace mají podle garantované max. výšky garantovanou čas. složitost  $O(\log n)$  pro  $n$  počet uzlů. Protože bez porušení vlastností červeno-černých stromů lze kořen vždy přebarvit načerno, můžeme pro ně předpokládat, že *kořen stromu je vždy černý*.

Vkládání vypadá následovně:

- Nalezení místa pro vložení a přidání nového prvku jako v obecných bin. vyhl. stromech, nový prvek se přebarví načerveno.
- Pokud je jeho otec černý, můžeme skončit – vlastnosti stromů jsou splněné. Pokud je červený, musíme strom upravovat (tady předpokládám, že otec přidávaného uzlu je levým synem, opačný případ je symetrický):
- Je-li i strýc červený, přebarvit otce a strýce načerno a přenést chybu o patro výš (je-li děd černý, končím, jinak můžu pokračovat až do kořene, který už lze přebarvovat beztréstně).
- Je-li strýc černý a přidávaný uzel je levým synem, udělat pravou rotaci na dědovi a přebarvit uzly tak, aby odpovídaly vlastnostem stromů.
- Je-li strýc černý a přidávaný uzel je pravým synem, udělat levou rotaci na otci a převést tak na předchozí případ.

*Odebírání* se provádí takto:

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

### **Definice** (*AVL stromy (Adelson-Velsky & Landis)*)

*AVL stromy* jsou, podobně jako červeno-černé stromy, bin. vyhledávací stromy, které zaručují max. logaritmický nárůst výšky vzhledem k počtu prvků. Pro každý uzel  $x$  se v AVL stromu definuje *faktor vyváženosti* jako rozdíl výšky jeho levého a pravého podstromu:  $\mathbf{bf}(x) = h(x \rightarrow \text{levý}) - h(x \rightarrow \text{pravý})$ . Pro všechny uzly v AVL stromu platí, že  $|\mathbf{bf}(x)| \leq 1$ .

### **Věta** (*Zaručení výšky AVL stromů*)

Výška AVL stromu s  $n$  vrcholy je  $O(\log n)$ . (Důkaz: buď  $T_n$  AVL strom výšky  $n$  s minimálním počtem uzlů. Ten má podstromy  $T_{n-1}$  a  $T_{n-2}$  atd., tj. velikost mini-

málního AVL stromu roste jako Fibonacciho posloupnost, tedy  $|T_n| \geq (\frac{1+\sqrt{5}}{2})^{n-1}$ .  
Důkaz tohoto indukci.)

### Algoritmus (*Operace na AVL stromech*)

Vyhledávací operace se provádí stejně jako na obecných bin. vyhledávacích stromech, vkládání a odebírání prvků taky, ale pokud tyto operace poruší zákl. vlastnost AVL stromů ( $|\mathbf{bf}(x) = 2|$ ), je nutné provést vyvažování – pomocí rotací (které mohou být propagovány až ke kořeni). Při vkládání a odebírání je navíc nutné průběžně (nejhůře až ke kořeni) upravovat indikaci faktoru vyváženosti jednotlivých uzlů.

## Halda

### Definice

*Halda (heap)* je dynamická množina se stromovou strukturou (binární halda je binární strom), pro kterou platí tzv. „vlastnost haldy“:

$$\text{Je-li } x \text{ potomek } y, \text{ pak } x \rightarrow \text{klíč} \geq y \rightarrow \text{klíč}$$

Haldy s touto nerovností jsou tzv. *min-heap*, pokud je nerovnost opačná, jde o *max-heap*.

### (Binární) haldy

Binární haldy jsou nejčastějším typem haldy. Zajišťují nalezení minimálního prvku v konstantním čase a odebrání a přidání minima v čase  $O(\log n)$ . V každé hladině od první až do předposlední je max. možný počet uzlů, v poslední jsou uzly co nejvíce „vlevo“ – tedy max. výška haldy s  $n$  prvky je  $(\log n) + 1$ . Proto je pro binární haldy jednoduše proveditelná jejich datová reprezentace polem (bez pointerů), kde při indexování od 0 má uzel na indexu  $k$ :

- Levého a pravého syna na indexu  $2k + 1$ , resp.  $2k + 2$  (pokud to není víc než celk. počet prvků, potom syny nemá).
- Rodiče na indexu  $\lceil \frac{k}{2} \rceil - 1$ .

*Přidání uzlu* do haldy znamená přidání prvku na konec haldy a dokud má jeho rodič větší klíč, jeho prohazování s rodičem (tedy posouvání o vrstvu výš). Při *odebírání uzlu* z haldy tento nahradím posledním prvkem v haldě a potom dokud neplatí vlastnost haldy (nejméně jeden z potomků má menší klíč), prohazují ho s potomkem s menším klíčem (a posouvám o vrstvu níž).

Vytvoření haldy je možné v čase  $O(n)$ , kde  $n$  je počet prvků v haldě – přidání 1 prvku do haldy trvá  $O(h)$ , kde  $h$  je aktuální výška (a  $h$  roste od 0 až k  $\lceil \log n \rceil$ , počet prvků ve výšce  $k$  je  $\frac{n}{2^{k+1}}$ , bereme-li výšku listů rovnou nule) - v součtu za všechny prvky jde o  $O(n \cdot \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h})$ .

Binární halda se používá např. k *třídění haldou* (heapsortu), kdy se z dat, která je potřeba utřídit, nejdříve postaví halda, a potom se opakuje operace odebrání kořene (tj. minimálního prvku).

## Fibonacciho haldy

Fibonacciho haldy mají nízkou časovou složitost běžných operací – amortizovaně  $O(1)$  pro vložení, hledání minima apod.; odebrání prvku a odebrání minima má složitost  $O(\log n)$  pro  $n$  prvků v haldě. Tvoří ji skupina stromů, vyhovujících „vlastnosti haldy“. Každý uzel haldy s  $n$  prvky má max.  $\log n$  potomků a ve svém podstromě minimálně  $F_{k+2}$  uzlů, kde  $F_k$  je  $k$ -té Fibonacciho číslo. To je zajištěno pravidlem, že při odebírání prvků lze z nekořenového uzlu oddělit max. 1 syna, jinak je nutné oddělit i tento uzel a ten se pak stane kořenem dalšího stromu. Počet stromů se snižuje při odebírání minima, kdy jsou spojovány dohromady.

Fibonacciho haldy se používají pro efektivní implementaci složitějších operací, jako např. Jarníkova nebo Dijkstrova algoritmu.

## 2.4 Hašování

**Definice** (*slovníkový problém*)

Dáno univerzum  $U$ , máme reprezentovat  $S \subseteq U$  a navrhnout algoritmy pro operace

- MEMBER( $x$ ) - zjistí zda  $x \in S$  a pokud ano nalezne kde,
- INSERT( $x$ ) - pokud  $x \notin S$ , vloží  $x$  do struktury reprezentující  $S$ ,
- DELETE( $x$ ) - když  $x \in S$ , smaže  $x$  ze struktury reprezentující  $S$ .

Například pomocí pole můžeme tyto operace implementovat rychle, ale nevýhodou je prostorová náročnost. Pro velké množiny je to někdy dokonce nemožné. *Hašování* se snaží zachovat rychlost operací a odstranit prostorovou náročnost.

Podívejme se nyní na *základní ideu* hašování. Mějme univerzum  $U$  a množinu  $S \subseteq U$  takovou, že  $|S| \ll |U|$ . Dále mějme funkci  $h : U \rightarrow \{0, 1, \dots, m-1\}$ . Množinu  $S$  potom reprezentujeme tabulkou (polem) o velikosti  $m$  tak, že prvek  $x \in S$  je uložen na řádku  $h(x)$ .

**Definice** (*Hašovací funkce, kolize*)

Funkci  $h : U \rightarrow \{0, 1, \dots, m-1\}$  potom nazýváme **hašovací funkcí**. Situaci  $h(s) = h(t)$ , pro  $s \neq t; s, t \in S$  nazveme **kolize**.

Jelikož mohutnost univerza  $U$  je větší než velikost hašovací tabulky, nelze se kolizím úplně vyhnout. Existuje spousta různých metod, jak kolize řešit. Podívejme se tedy na některé podrobněji.

**Definice**

Ještě si zavedeme některé značení. Velikost  $S$  (hašované množiny) označme  $\mathbf{n}$ , velikost tabulky (pole) označme  $\mathbf{m}$ , a faktor naplnění  $\alpha = \frac{\mathbf{n}}{\mathbf{m}}$ .

### Hašování se separovanými řetězci

Použijeme pole velikosti  $m$ , jehož  $i$ -tá položka bude spojový seznam  $S_i$  takový, že  $s \in S_i \Leftrightarrow h(s) = i$ , pro  $s \in S$ . Čili každý řádek pole obsahuje spojový seznam všech (kolidujících) prvků, které jsou hašovány na tento řádek. Seznamy nemusí být uspořádané, vznikají tak, jak jsou vkládány jednotlivé prvky do struktury.

### Algoritmus (*Hašování se separovanými řetězci*)

- **MEMBER** – Spočteme hodnotu hašovací funkce  $h(x)$ , prohledáme řetězec začínající na pozici  $h(x)$  a zjistíme zda se prvek nachází, či nenachází ve struktuře. Pokud se prvek v databázi nachází, tak musí nutně ležet v tomto řetězci.
- **INSERT** – Zjistíme zda  $x$  je v řetězci  $h(x)$ , pokud ne, přidáme ho nakonec, v opačném případě neděláme nic.
- **DELETE** – Vyhledá  $x$  v řetězci  $h(x)$  a smaže ho. Pokud se tam  $x$  nenachází, neudělá nic.

**Očekávaný počet testů** v neúspěšném případě je přibližně  $e^{-\alpha} + \alpha$  a při úspěšném vyhledávání přibližně  $1 + \frac{\alpha}{2}$ .

### Hašování s uspořádanými řetězci

Jak již je zřejmé z názvu je tato metoda téměř stejná jako předchozí. Jediný rozdíl je, že jednotlivé seznamy jsou uspořádány vzestupně dle velikosti prvků.

#### Algoritmus (*Hašování s uspořádanými řetězci*)

Rozdíly jsou pouze pro operaci **MEMBER**, kde skončíme prohledávání, když dojdeme na konec, nebo když nalezneme prvek, který je větší než hledaný a operaci **INSERT**, které vkládá prvek na místo kde jsme ukončili vyhledávání (před prvek, který ho ukončil).

**Očekávaný počet testů** v neúspěšném případě je přibližně roven  $e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$  a v úspěšném případě je přibližně  $1 + \frac{\alpha}{2}$ .

Nevýhodou předchozích dvou metod je nerovnoměrné využití paměti. Zatímco některé seznamy mohou být dlouhé, v některých není prvek žádný. Řešením je najít způsob, jak kolidující prvky ukládat na jiné (prázdné) řádky tabulky. Potom je ale nutné každý prvek tabulky rozšířit a položky pro práci s tabulkou.

Čím použijeme sofistikovanější metodu ukládání dat do tabulky, tím více budeme potřebovat položek pro práci s tabulkou a tedy vzroste paměťová náročnost. Naším cílem je tedy najít rozumný kompromis mezi sofistikovaností (rychlostí) strategie a její paměťovou náročností. Podívejme se na další algoritmy, které se o to pokoušejí.

### Hašování s přemísťováním

Seznamy jsou tentokrát ukládány do tabulky a implementovány jako dvousměrné. Potřebujeme tedy dvě položky pro práci s tabulkou: *next* – číslo řádku obsahující další prvek seznamu a *previous* – číslo řádku obsahující předchozí prvek seznamu. Když dojde ke kolizi, tj. chceme vložit prvek a jeho místo je obsazené prvkem z jiného řetězce, pak tento prvek z jiného řetězce přemístíme na jiný prázdný řádek v tabulce (proto hašování s přemísťováním).

### Algoritmus (Hašování s přemísťováním)

Algoritmus **MEMBER** funguje stejně jako u hašování se separovanými řetězci, jen místo ukazatele na další prvek použije hodnotu *next* z tabulky. Při operaci **INSERT** vložíme prvek kam patří pokud je tam místo, pokud již je místo obsazeno prvkem který tam patří, čili zde začíná seznam kolidujících prvků (*previous* = prázdné), pak postupujeme po položkách *next* až na konec seznamu, vložíme prvek na některý volný řádek tabulky a vyplníme správně hodnoty *next* a *previous*. Pokud je místo obsazeno prvkem z jiného seznamu (*previous* ≠ prázdné), tak tento prvek přemístíme na některý volný řádek, správně přepíšeme položky *next* a *previous* v měněném seznamu a vkládaný prvek uložíme na jeho místo. Operace **DELETE** je vcelku přímočará, jenom je třeba, pokud mažeme první prvek seznamu na jeho místo přesunout ten druhý v pořadí (pokud existuje).

**Očekávaný počet testů** je v neúspěšném případě roven přibližně  $(1 - \frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$  a v úspěšném je stejný jako pro hašování se separovanými řetězci a tedy  $\frac{n-1}{2m} + 1 \approx 1 + \frac{1}{\alpha}$ .

### Hašování se dvěma ukazateli

Hašování s přemísťováním má tu nevýhodu, že díky přemísťování prvků jsou operace **INSERT** a **DELETE** časově náročné. Tato metoda tedy implementuje řetězce jako jednosměrné seznamy, ale takové které nemusejí začínat na svém místě, tj. řetězec  $S_j$  obsahující prvky  $s \in S$  takové, že  $h(s) = j$ , nemusí začínat na  $j$ -tém řádku. Místo ukazatele na předchozí prvek tak do položek pro práci s tabulkou přidáme ukazatel na místo, kde začíná řetězec příslušný danému řádku. Položky pro práci s tabulkou tedy budou: *next* – číslo řádku tabulky kde je další prvek seznamu, *begin* – číslo řádku tabulky obsahující první prvek seznamu příslušného tomuto místu.

#### Algoritmus (Hašování se dvěma ukazateli)

Položka *begin* v  $j$ -tém řádku je vyplněna právě tehdy, když reprezentovaná množina  $S$  obsahuje prvek  $s \in S$  takový, že  $h(s) = j$ . Algoritmy jsou potom podobné těm u hašování s přemísťováním, ale přemísťování prvků je nahrazeno odpovídajícími změnami v položce *begin* daných řádků.

Díky práci s položkami jsou operace **INSERT** a **DELETE** rychlejší než při hašování s přemísťováním, ale začátek řetězce v jiném řádku tabulky přidá navíc jeden test, což změní složitost operace **MEMBER**.

**Očekávaný počet testů** v neúspěšném případě je přibližně  $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2 + \alpha) - 2$  a při úspěšném vyhledávání je roven  $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$

### Srůstající hašování

Nyní se podíváme na několik verzí metody, která se nazývá srůstající hašování. Budeme potřebovat jedinou položku pro práci s tabulkou a to ukazatel jednosměrného spojového seznamu. Na rozdíl od předchozích metod zde nejsou řetězce separované, v jednom řetězci mohou být prvky s různou hodnotou hašovací funkce. Když máme

přidat prvek  $s$ , tak ho zařadíme do řetězce, který se nachází na  $h(s)$ -tém řádku tabulky. Řetězce tedy v této metodě *srůstají*. Různé verze této metody se liší tím, kam přidáváme nový prvek a podle práce s pamětí. Dělí se na *standardní srůstající hašování* bez pomocné paměti a na hašování používající pomocnou paměť, kterému se říká jen *srůstající hašování*.

Nejdříve se budeme věnovat metodám standardního srůstajícího hašování (bez pomocné paměti):

- **LISCH** – late-insertion standard coalesced hashing – vkládá se za poslední prvek řetězce,
- **EISCH** – early-insertion standard coalesced hashing – vkládá se za první prvek řetězce.

Přirozená efektivní operace DELETE pro standardní srůstající hašování není známa. Na druhou stranu i primitivní algoritmy mají rozumnou očekávanou časovou složitost.

Další otázka zní, proč používat metodu EISCH, když programy pro metodu LISCH jsou jednodušší. Odpověď je na první pohled dost překvapující. Při úspěšném vyhledávání je metoda EISCH rychlejší než metoda LISCH. Je to proto, že je o něco pravděpodobnější, že se bude pracovat s novým prvkem. V neúspěšném případě jsou samozřejmě obě metody stejné, neboť řetězce jsou u obou stejně dlouhé.

Metody srůstajícího hašování (s pomocnou pamětí) mají použitou paměť rozdělenou na dvě části. Na tu přímo adresovatelnou hašovací funkcí a na pomocnou část. Adresovací část má  $m$  řádků, pokud hašovací funkce má hodnoty z oboru  $\{0, 1, \dots, m-1\}$ , v pomocné části jsou řádky ke kterým nemáme přístup přes hašovací funkci. Když při přidávání nového prvku vznikne kolize, tak se nejprve vybere volný řádek z pomocné části a teprve když je pomocné část zaplněna použijí se k ukládání kolidujících prvků řádky z adresovatelné části tabulky. Tato strategie oddaluje srůstání řetězců. Srůstající hašování se tedy, aspoň dokud není zaplněna pomocná část tabulky, podobá hašování se separovanými řetězci. Existují základní tři varianty:

- **LICH** – late-insertion coalesced hashing – vkládá prvek na konec řetězce,
- **VICH** – early-insertion coalesced hashing – vkládá prvek na řádek  $h(x)$  pokud je prázdný a nebo hned za prvek na řádku  $h(x)$ ,
- **EICH** – varied-insertion coalesced hashing – vkládá se za poslední prvek řetězce, který je ještě v pomocné části. Pokud v pomocné části žádný není, vkládá se hned za prvek na pozici  $h(x)$ .

Tyto metody nepodporují přirozené efektivní algoritmy pro operaci DELETE.

### Hašování s lineárním přidáváním

Následující metoda nepoužívá žádné položky pro práci s tabulkou to znamená, že způsob nalezení dalšího řádku řetězce je zabudován přímo do metody. Metoda funguje tak, že pokud chceme vložit prvek do tabulky a nastane kolize, najdeme první následující volný řádek a tam prvek vložíme. Předpokládáme, že řádky jsou číslovány modulo  $m$ , čili vytvářejí cyklus délky  $m$ .

Tato metoda sice využívá minimální velikost paměti, ale v tabulce vznikají shluky obsazených řádků a proto je při velkém zaplnění pomalá. Navíc metoda nepodporuje efektivní operaci DELETE.

## Shrnutí

Zde uvedeme pořadí metod hašování podle očekávaného počtu testů.

### Neúspěšné vyhledávání:

1. Hašování s uspořádanými separovanými řetězci,
2. Hašování se separovanými řetězci = Hašování s přemísťováním,
3. Hašování se dvěma ukazateli,
4. VICH = LICH
5. EICH,
6. LISCH = EISCH,
7. Hašování s lineárním přidáváním.

### Úspěšné vyhledávání

1. H. s uspořádanými řetězci = H. se separovanými řetězci = H. s přemísťováním,
2. Hašování se dvěma ukazateli,
3. VICH,
4. LICH,
5. EICH,
6. EISCH,
7. LISCH,
8. Hašování s lineárním přidáváním.

### Poznámka

Metody se separovanými řetězci a srůstající hašování používají více paměti. Metoda s přemísťováním vyžaduje více času – na přemístění prvku. Otázka která z metod je nejlepší není proto jednoznačně rozhodnutelná a je nutné pečlivě zvážit všechny okolnosti nasazení metody a všechny naše požadavky na ní, než se rozhodneme, kterou použijeme.

### Univerzální hašování

Pro dobré fungování hašování potřebujeme mimo jiné, aby vstupní data byla rovnoměrně rozdělena a toho někdy není možné dosáhnout. Odstranit tento nedostatek se pokouší metoda *univerzální hašování*. Základní idea této metody je taková, že máme množinu  $H$  hašovacích funkcí z univerza do tabulky velikosti  $m$  takových, že pro  $S \subseteq U$ ,  $|S| \leq m$  se většina funkcí chová dobře v tom smyslu, že má malý počet kolizí. Hašovací funkci potom zvolíme z množiny  $H$  (takovou s rovnoměrným rozdělením). Jelikož funkci volíme my, můžeme požadavek rovnoměrného rozdělení zajistit.

## Perfektní hašování

Jiná možnost jak vyřešit kolize, je najít takzvanou *perfektní hašovací funkci*, tj. takovou které nepřipouští kolize. Nevýhoda této metody je, že nelze dost dobře implementovat operaci INSERT, proto se dá prakticky použít pouze tam, kde předpokládáme hodně operací MEMBER a jen velmi málo operací INSERT. Kolize se potom dají řešit třeba malou pomocnou tabulkou, kam se ukládají kolidující data.

Pro rozumné fungování metody je nutné, aby hašovací funkce byla rychle spočitatelná a aby její zadání nevyžadovalo mnoho paměti, nejvýhodnější je analytické zadání.

Naopak jedna z výhod je, že nalezení perfektní hašovací funkce, může trvat dlouho, neboť ho provádíme pouze jednou na začátku algoritmu.

## Externí hašování

Externí hašování řeší trochu jiný problém, než výše popsané metody. Chceme uložit data na externí médium a protože přístup k externím médiím je o několik řádů pomalejší, než práce v interní paměti, bude naším cílem minimalizovat počet přístupů do ní. Externí paměť bývá rozdělena na stránky a ty většinou načítáme do interní paměti celé. Tato operace je však velice pomalá. Problémem externího hašování je tedy nalézt datovou strukturu pro uložení dat na vnější paměti a algoritmy pro operace INSERT, DELETE a MEMBER, tak abychom použili co nejmenší počet komunikací mezi vnější a vnitřní paměti.

Metod externího hašování je opět mnoho. Některé používají pomocnou datovou strukturu v interní paměti, kterou často nazýváme adresář. Pokud metody nemají žádnou takovou pomocnou strukturu neobejdou se obvykle bez oblasti přetečení. Některé známější metody vnějšího hašování jsou například: „Litwinovo lineární hašování“, „Faginovo rozšiřitelné hašování“, „Cormackovo perfektní hašování“ nebo „Perfektní hašování Larsona a Kajli“.

## 2.5 Sekvenční třídění, porovnávací algoritmy, přihrádkové třídění, třídící sítě

TODO: trochu víc formalismu by tu neškodilo, taky je potřeba sjednotit óčkovou notaci (zřejmě prosté nahrazení symbolu  $O$  symbolem  $\Theta$  by stačilo, ale chce to ověřit).

### Sekvenční třídění a porovnávací algoritmy

Pojmy „sekvenční třídění“ a „porovnávací algoritmy“ mohou znamenat vlastně cokoliv, takže uvedu pár nejběžnějších třídících algoritmů a budu doufat, že to bude ke zkoušce stačit :-). Zdrojem mi budiž Wikipedie a kniha Algoritmy a programovací techniky Doc. P. Töpfera.

#### **Algoritmus** (*Selection sort, třídění výběrem*)

Selection sort je jeden z nejjednodušších třídících algoritmů. Jde o vnitřní třídění – tedy celá posloupnost prvků by měla být v paměti. Má časovou složitost  $\Theta(n^2)$  a obecně bývá pomalejší než insertion sort. Pracuje následovně:

Udrží si množinu setříděných prvků na začátku posloupnosti (pole), která je na začátku prázdná a na konci představuje celé pole. Zbytek pole za setříděnou množinou je neuspořádaný. V jednom kroku vždy vybere jeden prvek a vloží ho do utříděné části (kterou tím zvětší o 1 a zároveň zmenší nesetříděnou). Jeden krok algoritmu (kterých je  $n$  pro  $n$  prvků v každém případě) vypadá takto:

1. Najdi nejmenší prvek z nesetříděného úseku.
2. Vlož ho přesně za konec setříděného úseku (a prvek co tam byl původně si s ním vymění místo)

Heapsort, který popíšu později, může být považovaný za variantu selection sortu, protože také vybírá minimum a začleňuje do setříděné části.

**Algoritmus** (*Insertion sort, třídění vkládáním*)

Insertion sort je také relativně jednoduchý a na velké datové soubory neefektivní, ale jednoduchý na implementaci a rychlejší než nejprimitivnější algoritmy bubble sort a selection sort. Navíc je efektivní pro data, která jsou už částečně předtříděná – v nejhorším případě sice běží v čase  $O(n^2)$ , ale v nejlepším případě (úplné setřídění dat) je lineární – obecně běží v čase  $O(n+d)$ , kde  $d$  je počet inverzí ve tříděné posloupnosti. Navíc je stabilní (zachovává pořadí prvků se stejným klíčem) a „in-place“, tedy nepotřebuje žádné pomocné datové struktury. Proti selection sortu ale většinou potřebuje více přepisování (a to může u velkých datových struktur vadit).

V jednom kroku vždy vezme nějaký prvek (berou se po řadě od začátku pole), zapamatuje si jeho hodnotu, a dokud před ním jsou prvky s větším klíčem, posouvá je na pozici o 1 větší (čímž vždy přepíše následující, takže původní prvek se ztratí) a pokud narazí na prvek s menším klíčem, do za něj napíše onen zapamatovaný prvek (a místo tam je, protože celou cestu k němu posouval prvky). Algoritmus vypadá takto:

```
insert sort( array a ){
  for( i = 1; i < a.length - 1; ++i ){
    value = a[i];
    j = i-1;
    while( j >= 0 && a[j] > value ){
      a[j + 1] = a[j];
      j = j-1;
    }
    a[j+1] = value;
  }
}
```

Jednou z variant insertion sortu je *Shell sort*, který porovnává prvky ne vedle sebe, ale vzdálené o nějaký počet polí, který se postupně zmenšuje. Může dosahovat složitost  $O(n^{3/2})$  až  $O(n^{4/3})$ . S jistými úpravami se u něj dá dosáhnout až  $O(n \log^2 n)$ . Jiné vylepšení je *library sort*, který si při vkládání nechává mezery pro další prvky (podobně jako v knihovně nejsou poličky úplně plné) – ten může s velkou pravděpodobností běžet v čase  $O(n \log n)$ , ale zase potřebuje větší paměťový prostor.

### **Algoritmus** (*Bubble sort, bublinkové třídění*)

Bubble sort je velmi jednoduchý třídící algoritmus (asi nejjednodušší na implementaci), s časovou složitostí  $O(n^2)$ . V nejlepším případě (pro úplně setříděná data) mu ale stačí jen jeden průchod, takže  $O(n)$ . Většinou ale bývá pomalejší i než insertion sort, takže se na velké množiny dat nehodí.

Algoritmus prochází v jednom kroku celé pole a hledá pozice, kde se prvek s menším klíčem nachází bezprostředně za prvkem s větším klíčem. Takovéto dva prvky pak vymění. Kroky opakuje, dokud neprojde celé pole bez jediného prohození prvků (nebo v „tupější“ variantě  $n$ -krát pro  $n$  prvků, protože pak je zaručeno, že posloupnost bude pro libovolné pořadí prvků setříděná – ta má ale pak složitost  $O(n^2)$  v každém případě!).

Vylepšení algoritmu lze dosáhnout jednoduchou úvahou: největší prvek je už při prvním průchodu polem odsunutý až na konec. To se samozřejmě opakuje pro každý průchod (ve druhém je předposlední na druhém místě od konce atp.), takže lze průchody postupně zkracovat a konec pole už netestovat – dosáhneme tím v průměru dvojnásobné rychlosti.

Variantou bubble sortu je *shake sort* neboli *cocktail sort*, který střídavě prochází posloupnost prvků nejdřív od začátku a pak od konce (a přitom provádí to samé jako bubble sort). Tím může v některých případech o trochu třídění zrychlit – příkladem budiž posloupnost prvků (2, 3, 4, 5, 1), která potřebuje jen 1 průchod cocktail-sortem tam a jeden zpět, ale pro bubble-sort by potřebovala 4.

Dalším vylepšením bubble sortu je *Comb sort*, který o něco zvyšuje rychlost. Je založen na stejné myšlence jako shell sort – tedy nejsou porovnávány prvky bezprostředně za sebou, ale prvky posunuté o nějaký ofset – ten je na začátku roven délce posloupnosti, a postupně se dělí „zkracovacím faktorem“ (běžná hodnota 1.3) až dosáhne jedné. Složitost se pohybuje mezi  $O(n^2)$  v nejhorším případě a  $O(n \log n)$  v nejlepším. V průměrném případě jde stále o  $O(n^2)$ , ale s menší konstantou než u bubble-sortu (TODO: tohle je potřeba set-sakra ověřit ... opsané z německé wiki a „talk:Comb sort“ na anglické, takže fakt „důvěryhodné“).

### **Algoritmus** (*Heap sort, třídění haldou*)

Heapsort je také třídící algoritmus založený na porovnávání a myšlenkově vychází ze selection sortu, ke kterému přidává práci s haldou. Většinou bývá pro typická vstupní data pomalejší než quicksort, ale zaručuje časovou složitost  $O(n \log n)$  i v nejhorším případě. Jde o „in-place“ algoritmus (halda se může nacházet přímo v nesetříděné části pole), ale není „stabilní“.

Algoritmus sám, máme-li vyřešené operace na haldě, je velice jednoduchý – nejdříve pro každý prvek opakuje jeho vložení do haldy (takže postupně vytvoří  $n$ -prvkovou haldu, která se s každým krokem zvětšuje o 1), pro implementaci haldy na začátku pole je vhodný „max-heap“, a potom opakuje odebrání maxima a jeho přesun na volné místo hned za konci zmenšivší se haldy – takže od konce pole postupně roste směrem k začátku setříděná posloupnost.

Upravený heapsort s použitím ternární haldy dosahuje o multiplikatívni konstantu lepší výsledky, existuje i (prý :-)) složitá varianta *smoothsort*, která se blíží časové složitosti  $O(n)$ , pokud jsou data částečně předtříděná – heapsort totiž pracuje pro libovolnou posloupnost v čase  $O(n \log n)$ .

### Algoritmus (*Merge sort, třídění sléváním*)

Dalším třídícím algoritmem založeným na porovnávání prvků je mergesort. Je stabilní, takže zachovává pořadí dat se stejným klíčem. Jde o příklad algoritmu typu „rozděl a panuj“, stejně jako u níže popsaného quicksortu. Byl vynalezen Johnem Von Neumannem. Je založen na rozdělení posloupnosti na dvě zhruba stejné poloviny, rekurzivním setřídění a potom „slévání“ dvou již setříděných posloupností. Jeho časová složitost je  $O(n \log n)$  i v nejhorším případě, provádí většinou méně porovnání než quicksort, má větší nároky na paměť v případě rekurzivního volání (existuje ale i nerekurzivní verze), ale většinou nepracuje na místě a potřebuje alokovat paměť pro výstup setříděných posloupností (i toto se dá odstranit, ale je to zbytečně složité a přílišné zrychlení oproti použití jiného algoritmu nepřinese). Jeho přístup ho ale činí ideálním k použití na médiích se sekvenčním přístupem k datům (např. pásky). Jde tedy použít i ke třídění na vnější paměti – detaily viz sekce o databázích.

Postup práce je následující:

1. Rozděl nesetříděnou posloupnost na dvě (zhruba) poloviční části
2. Pokud mají více než jeden prvek, setříd' je rekurzivním zavoláním mergesortu (tj. pro každou z nich pokračuj od kroku 1 do konce algoritmu), jinak pokračuj následujícím krokem.
3. Slij dvě setříděné posloupnosti do jedné – vyber z obou posloupností první prvek, a pak opakovaně prvky porovnávej, zapisuj do setříděné posloupnosti menší z nich a doplňuj dvojici z té poloviční posloupnosti, odkud pocházel zapsaný prvek.

### Algoritmus (*Quicksort*)

Quicksort je jedním z nejrychlejších algoritmů pro třídění na vnitřní paměti, přestože v nejhorším případě může jeho časová složitost dosáhnout až  $\Theta(n^2)$ . Pro ideální i průměrná data dosahuje  $\Theta(n \log n)$ . Je také založen na principu „rozděl a panuj“, i když poněkud jiným způsobem než předchozí zmiňovaný, od něhož se liší i tím, že není stabilní.

Algoritmus nejdřív vybere nějaký prvek, tzv. *pivot*, a prvky s klíčem větší než *pivot* přesune do jiné části pole než ty s klíčem menším. Pak rekurzivně třídí obě části pole – když se dostane k polím délky 1, problém je vyřešen. Postup vypadá takto:

1. Vyber *pivot* (jeden prvek ze seznamu). Tady jde o největší magii, protože k dosažení nejlepší rychlosti by se měl pokaždé vybírat medián. Nejjednodušší je vybrat první, ale tento výběr ovlivňuje výslednou rychlost práce, takže se vyplatí např. vzít tři prvky, porovnat je a vzít si z nich ten prostřední.
2. Postupuj od začátku pole a hledej první prvek větší nebo rovný než *pivot*. Až ho najdeš, postupuj od konce a najdi první prvek menší než *pivot*.
3. Prvky prohoď a opakuj krok 2 a 3, dokud se hledání od začátku a od konce nepotká na nějaké pozici – tu pojmenujeme třeba *k*.
4. Rekurzivním voláním setříd' prvky  $(0, \dots, k)$  a  $(k + 1, \dots, n - 1)$  (má-li tříděné pole délku *n*) – to znamená pro obě části pole pokračuj od kroku 1. Pokud je

$k = 0$  nebo  $k = n - 2$ , není třeba už rekurzivního volání, protože posloupnosti délky 1 jsou setříděné.

Pro algoritmus existuje i nerekurzivní verze (stačí rekurzi nahradit zásobníkem úseků čekajících na zpracování). Je vidět, že na volbě pivotu závisí všechno – pokud pokaždé jako pivot volím 1. nebo  $n - 1$ . hodnotu v poli v pořadí podle velikosti, dělím pak vždy na části o délce 1 a  $n - 1$ , takže tento rekurzivní krok provedu až  $n$ -krát a dostanu se k času  $\Theta(n^2)$ . Samozřejmě, díky existenci algoritmu pro nalezení mediánu v čase  $\Theta(n)$  je možné i tady dosáhnout zaručené složitosti  $\Theta(n \log n)$ , ale v praxi je to kvůli vysoké multiplikační konstantě nepoužitelné – k výběru pivotu se většinou s úspěchem užívá nějaká jednoduchá heuristika, jak je nastíněno v popisu algoritmu samotného.

Heapsort bývá pomalejší než quicksort, ale zaručuje nízkou časovou složitost i pro nejhorší případ a navíc potřebuje méně paměti – nároky quicksortu navíc (kromě tříděné posloupnosti) jsou  $O(\log n)$  minimálně, kvůli nutnosti použití rekurzivního volání nebo zásobníku. Oproti mergesortu ho nelze použít na data se sekvenčním přístupem, tyto nevýhody ale vyvažuje relativní jednoduchostí implementace a rychlostí v průměrném případě.

Variantou quicksortu je *introsort*, který ho kombinuje s heapsortem, pokud hloubka rekurze dosáhne nějakých nepříjemných hodnot – tak je zaručena časová složitost  $\Theta(n \log n)$  i v nejhorším případě (samozřejmě je to ale v nejhorším případě pořád pomalejší než použití jen heapsortu). Jedna z variant tohoto algoritmu se dá použít k hledání  $k$ -tého nejmenšího prvku (tedy i mediánu), kdy dosahuje složitosti  $O(n)$  průměrně až  $O(n^2)$  nejhůře.

## Přihrádkové třídění

### **Algoritmus** (*Bucket sort, Radix sort, přihrádkové třídění*)

Radix sort je zvláštní třídící algoritmus – jeho složitost je totiž lineární. Dosahuje to tím, že neporovnává všechny tříděné prvky (složitost problému třídění pomocí porovnávání je  $\Theta(n \log n)$ , takže by to jinak nebylo možné), je ho ale možné použít jen pro třídění dat podle klíče z nějaké ne příliš velké množiny – max. rozsah tříděných hodnot závisí na tom, jak velké pole si můžeme dovolit vymezit v paměti pro tento účel.

Nejjednodušší varianta (tzv. *pigeonhole sort*, nebo-li *counting sort*) opravdu počítá s klíči ze zadaného rozmezí  $[l, h]$ . Pro něj si připraví cílové pole velikosti  $h - l + 1$ , tj. „přihrádky“. Do nich pak přímo podle klíče přehazuje čtené prvky (jestliže přihrádky realizujeme jako seznamy, bude třídění dokonce stabilní). Nakonec projde přihrádky od začátku do konce a co v nich najde, to vypíše (a výstup bude setříděný). Variantou counting sortu je *bucket sort*, kdy se do jedné přihrádky nedávají jen prvky se stejným klíčem, ale prvky s klíčem v nějakém malém rozmezí – ty pak lze setřídít rychle, protože jich zřejmě nebude mnoho, a navíc se ušetří paměť.

Protože ale klíče velikosti max. tisíců hodnot jsou většinou trochu málo, v praxi se běžně používají složitější varianty – ty zahrnují několik průchodů nahoře popsaného algoritmu, při nichž se třídí jenom podle části klíče. Ty se dělí na ty,

které začínají od nejméně významné části klíče (*least significant digit radix sort*) a ty, které jdou od nejvýznamnější části (*most significant digit*). První z nich mají tu výhodu, že lze zachovat stabilitu třídění, druhá zase může třídít i podle klíčů různé délky a zastavovat se po nalezení unikátních prefixů, takže se hodí např. pro lexikografické třídění podle řetězcových klíčů.

Třídění typu *least significant digit* vypadá následovně:

1. Vezmi nejméně významnou část klíče (určitý počet bitů).
2. Rozděl podle této části klíče data do přihrádek, ale v nich zachovej jejich pořadí (to je nutné kvůli následnému průchodu, zároveň to dělá z tohoto algoritmu stabilní třídění).
3. Opakuj toto pro další (významnější) část klíče.

Most significant digit varianta (rekurzivní verze, je založená na bucket sortu) běhá takto:

1. Vezmi nejvýznamnější část klíče (první písmeno, například).
2. Rozděl prvky podle této části do přihrádek (takže v jedné se jich octne docela hodně)
3. Rekurzivně seříd' každou z přihrádek (začni podle další části klíče), pokud je v ní více než jeden prvek (tohle zaručí zastavení za rozlišujícím prefixem).
4. Slep přihrádky do jedné (seříděné) posloupnosti.

Popisované algoritmy většinou potřebují  $O(n + (h - l))$  času k třídění, je-li  $h - l$  (zhruba) počet přihrádek – to znamená, že sice jde o složitost lineární, ale lineární i v počtu přihrádek, což se nemusí vždy oproti konvenčnímu třídění vyplatit. Navíc jsou problémem vysoké nároky na paměť (nelze třídění provést „na místě“ v jediném poli). Pro malou množinu hodnot klíčů (nebo u most significant digit varianty krátké odlišující prefixy) jsou ale časově efektivnější.

## Třídící sítě

Zdrojem této sekce jsou zápisky z přednášek Prof. L. Kučery Algoritmy a datové struktury II.

### **Definice** (*Bitonická posloupnost*)

Řekneme, že posloupnost prvků je *bitonická*, pokud po spojení do cyklu (tedy nultý prvek za  $n$ -tý) obsahuje dva monotónní úseky. Nebo-li obsahuje až na fázový posuv dva monotónní úseky.

### **Definice** (*Komparátor*)

*Komparátor* je speciální typ hradla (představme si pod tím nedělitelnou elektronickou součástku, případně jen virtuální), která má dva výstupy a dva vstupy. Pokud na vstupy přivedeme dva prvky (klíče, čísla), z levého výstupu vydá menší z nich a z pravého výstupu větší (takže vlastně porovná dva prvky a na výstup je vyvlivne ve správném pořadí). Pracuje v konstantním čase.

### Definice (Třídící síť)

Třídící síť je správně sestavená množina komparátorů dohromady spojená vstupy a výstupy tak, že při přivedení posloupnosti délky  $n$  na vstup ji vydá setříděnou na výstupu. Komparátory v ní jsou rozčleněné do hladin, jejichž počet pak udává celkovou dobu výpočtu – předpokládá se tam, že komparátory v jednotlivých vrstvách pracují paralelně, takže třídící sítě mohou dosahovat časové složitosti pouhých  $O(\log n)$ . Algoritmus s takovou časovou složitostí sice existuje, ale má velmi vysokou multiplikativní konstantu, takže se v praxi nepoužívá. Příkladem třídící sítě je i bitonické třídění.

### Algoritmus (Bitonické třídění)

Bitonická třídící síť je založena na použití bitonických posloupností a rekurze. Obvod (pro třídění dat délky  $n$ ) se dělí na dvě části:

- První část setřídí (rekurzivně)  $1/2$  vstupu vzestupně, druhou polovinu sestupně a tím vytvoří bitonickou posloupnost. Obsahuje tedy dvě třídící sítě pro třídění posloupností délky  $\frac{n}{2}$ .
- Druhá část třídí jen bitonické posloupnosti – první její vrstva rozdělí bitonickou posloupnost na vstup na dvě bitonické posloupnosti (z větších a menších čísel). Další vrstvy už jsou opět implementovány rekurzivně – tedy druhá vrstva dostane dvě posloupnosti a vyrobí z nich čtyři atd., až nakonec dojde k „bitonickým posloupnostem“ délky 1.

K rozdělení jedné bitonické posloupnosti délky  $k$  na dvě stačí jen  $\frac{k}{2}$  komparátorů, které porovnávají vždy  $i$ -tý a  $k + i$ -tý prvek. Dojde sice k nějakému fázovému posuvu, ale to ničemu nevádí. Dobře je to vidět při znázornění na kružnici, doporučuji prohlédnout si postup v programu Algovision Prof. Kučery (<http://kam.mff.cuni.cz/~ludek/AlgovisionPage.html>).

Je vidět, že počet vrstev potřebných k dělení bitonických posloupností délky  $N$  je  $\log_2 N$  ( $B(N) = \log N$ ). Pro celkový počet vrstev, a tedy dobu zpracování –  $T(n)$  nám vychází následující vzorec

$$T(N) = T\left(\frac{n}{2}\right) + B(N) = \log N + \log(N/2) + \dots + 1$$

z čehož díky vzorci pro součet aritmetické posloupnosti  $1 + 2 + \dots + k = \frac{k(k+1)}{2}$  vyjde

$$T(N) = O\left(\frac{1}{2} \log^2 N\right)$$

## 2.6 Grafové algoritmy

TODO: nejake vyuziti tech algoritmu (staci prikklad plusminus ke kazdemu druhu ulohy)

### Graf

## Definice

*Graf*  $G$  je dvojice  $(V, E)$ , kde  $V$  je množina bodů (*vrcholů*) a  $E$  množina jejich dvojic (*hran*). Je-li  $E$  množinou neuspořádaných dvojic, jde o *neorientovaný* graf. Jsou-li dvojice uspořádané, jedná se o *orientovaný* graf. Velikost množiny  $V$  se značí  $n$ , velikost  $E$  je  $m$  -  $|V| = n$ ,  $|E| = m$ .

Graf je možné strojově reprezentovat např. pomocí *matice sousednosti* - matice, kde je na souřadnicích  $(u, v)$  hodnota 1, pokud z  $u$  do  $v$  je hrana a 0 jinak. Pro neorientované grafy je souměrná podle hlavní osy. Matice zabírá  $\Theta(n^2)$  místa v paměti. Další možností jsou *seznamy sousedů* - dvě pole, jedno příslušné vrcholům, druhé hranám. V prvním jsou uloženy indexy do druhého pole, určující kde začínají seznamy hran vedoucích z vrcholu (příslušejícímu k indexu v prvním poli). Paměťová náročnost je  $\Theta(m + n)$ .

## Prohledávání do hloubky a do šířky

Algoritmy, které postupně projdou všechny vrcholy daného souvislého neorientovaného grafu.

**Algoritmus** (*Prohledávání do šířky/Breadth-First Search*)

Prochází všechny vrcholy grafu postupně po vrstvách vzdáleností od iniciálního vrcholu. K implementaci se používá fronta (FIFO).

```
BFS( V - vrcholy, E - hrany, s - startovací vrchol ){
    obarvi vrcholy bíle, nastav jim nekonečnou vzdálenost od s a předchůdce NULL;
    dej do fronty vrchol s;

    while( neprázdna fronta ){
        vyber z fronty vrchol v;
        foreach( všechny bíle obarvené sousedy v = u ){
            obarvi u šedě a nastav mu vzdálenost d(v) + 1 a předchůdce v;
            dej vrchol u do fronty;
        }
        v přebarvi na černo a vyhoď z fronty.
    }
}
```

Běží v čase  $\Theta(m + n)$ , protože každý vrchol testuje 2x pro každou hranu, do fronty ho dává 1x a obarvení mu mění 2x. Tento algoritmus je základem několika dalších, např. pro testování souvislosti grafu, hledání minimální kostry nebo nejkratší cesty.

**Algoritmus** (*Prohledávání do hloubky/Depth-First Search*)

Prochází postupně všechny vrcholy - do hloubky (pro každý vrchol nejdřív navštíví první jeho nenavštívený sousední vrchol, pak první sousední tohoto vrcholu atp. až dojde k vrcholu bez nenavštívených sousedů, pak se vrací a prochází další ještě nenavštívené sousedy). Pro implementaci se používá buď zásobník, nebo rekurze. Zásobníková verze vypadá stejně jako prohledávání do šířky (místo fronty je zásobník).

Rekurzivní verze - při zavolání na startovní vrchol projde celý graf:

```
DFS(v - vrchol){  
  
    označ v jako navštívený;  
    foreach( všechny nenavštívené sousedy v = u )  
        DFS( u );  
}
```

Časová složitost je  $\Theta(m + n)$ , stejně jako u prohledávání do šířky.

## Souvislost

### Definice

*Cesta* v grafu  $G = (V, E)$  z vrcholu  $a$  do vrcholu  $b$  je posloupnost  $v_0, v_1, \dots, v_n$  taková, že  $v_0 = a$ ,  $v_n = b$  a pro všechna  $v_i$ ,  $i \in \{1, \dots, n\}$  je  $(v_{i-1}, v_i) \in E$ . Graf  $G = (V, E)$  je *souvislý*, pokud pro každé dva vrcholy  $u, v \in V$  existuje v  $G$  cesta z  $u$  do  $v$ . Toto platí pro orientované i neorientované grafy.

### Algoritmus (*Testování souvislosti grafu/počítání komponent souvislosti*)

Algoritmus využívá prohledávání do šířky (nebo do hloubky) - v 1 kroku vždy najde dosud nenavštívený vrchol, začne z něj procházet graf a takto projde (oddělí) jednu komponentu souvislosti. Pokud skončí po prvním kroku, graf je souvislý. Počet kroků, potřebných k navštívení všech vrcholů grafu, je zároveň počtem komponent souvislosti.

Časová složitost je  $\Theta(m + n)$ , protože o algoritmu platí to samé co o prohledávání do šířky - žádný vrchol nebude přidán do fronty více než jednou a testován více než 2x pro každou hranu.

## Topologické třídění

### Definice

*Topologické uspořádání* vrcholů orientovaného grafu  $G = (V, \vec{E})$  je funkce  $t : V \rightarrow \{1, \dots, n\}$  taková, že pro každou hranu  $(i, j) \in E$  je  $t(i) < t(j)$ . Lze provést pouze pro acyklické orientované grafy.

### Algoritmus (*Primitivní algoritmus*)

V každém kroku najde vrchol, z něhož nevedou žádné hrany. Přiřadí mu nejvyšší volné číslo (začíná od  $n$ ) a odstraní ho ze seznamu vrcholů. Uspořádání takto vytvořené je topologické, složitost algoritmu je  $\Theta(n(m + n))$ .

### Algoritmus (*Rychlý algoritmus*)

K topologickému uspořádání se dá použít modifikace prohledávání do hloubky. Není třeba ani graf předem testovat na přítomnost cyklů, algoritmus toto objeví. Pro každý navštívený vrchol si poznamená čas jeho opuštění, uspořádání podle klesajících časů opuštění je topologické.

```

topologické_třídění( v - vrchol ) {

    global t; // čas opuštění, iniciální hodnota 0

    označ v jako navštívený;
    foreach ( u in sousední vrcholy v ) {
        if ( u je navštívený, ale ne opuštěný ) {
            chyba - cyklus;
            return;
        }
        else if ( u není navštívený )
            topologické_třídění( u );
    }
    označ v jako opuštěný v čase t;
    t = t + 1;
}

```

Časová složitost zůstává stejná jako u prohledávání do šířky, tedy  $\Theta(m + n)$ , protože všechny kroky prováděné v rámci navštívení 1 vrcholu vyžadují jen konstatní počet operací.

### Poznámka

Topologické třídění se používá např. k zjištění nejvhodnějšího pořadí provedení navzájem závislých činností.

## Hledání nejkratší cesty v grafu

### Definice

*Ohodnocení hran - váhová funkce* je funkce, která každé (orientované) hraně přiřazuje její „délku“ nebo „cenu“ jejího projití. Definuje se jako  $w : E \rightarrow \mathbb{R}$ . *Délka* (orientované) cesty  $p = v_0, v_1, \dots, v_n$  v ohodnoceném grafu (grafu s váhovou funkcí) je potom  $w(p) = \sum_{i=1}^n w(v_{i-1}, v_i)$ .

*Vzdálenost* dvou vrcholů  $u, v$  (váha nejkr. cesty z  $u$  do  $v$ ) je  $\delta(u, v) = \min\{w(p) \mid p \text{ je cesta z } u \text{ do } v\}$ , pokud nějaká cesta z  $u$  do  $v$  existuje, jinak  $\delta(u, v) = \infty$ . *Nejkratší cesta*  $p$  z  $u$  do  $v$  je taková, pro kterou  $w(p) = \delta(u, v)$ .

### Poznámka

Pro hledání nejkratší cesty v obecném grafu bez ohodnocení hran (tj. délka cesty je počet hran na ní) stačí prohledávání do šířky.

### Algoritmus (*Algoritmus kritické cesty (pro DAG)*)

Pro hledání nejkratší cesty do všech bodů z jednoho zdroje v orientovaném acyklickém grafu (DAG) používá topologické třídění, které je pro takovýto graf proveditelné; spolu se zpřesňováním horních odhadů vzdáleností vrcholů.

Mám daný startovací vrchol  $s$ . Definuji  $d(s, v)$  jako horní odhad vzdálenosti  $s$  a  $v$ , tj. vždy  $d(s, v) \geq \delta(s, v)$  pro lib. vrchol  $v$ . Hodnoty  $d(s, v)$  před započítáním výpočtu inicializuji na  $+\infty$ .

V algoritmu se provádí operace „Relax“, znamenající zpřesnění odhadu  $d(s, v)$  za použití cesty vedoucí z  $s$  do  $v$ , končící hranou  $(u, v)$  – pokud má taková cesta nižší váhu než byl předchozí odhad  $d(s, v)$ , položím  $d(s, v) = d(s, u) + w(u, v)$ . Tato operace zachovává invariant  $d(s, v) \geq \delta(s, v)$ .

```
Relax (u, v) { //u = source, v = destination
  if (v.distance > u.distance + uv.weight) {
    v.distance := u.distance + uv.weight
    v.predecessor := u
  }
}

kritická cesta( V - vrcholy, E - hrany, s - startovací vrchol ){

  topologicky seřídí V;
  inicializace - nastav d(s,v) = nekonečno pro všechny vrcholy;
  foreach( vrchol v, v pořadí podle top. třídění ){
    proved' operaci Relax za použití cest
      vedoucích do v přes všechna možná u;
  }
}
```

Výsledek dává nejkratší cesty díky topologickému seřídění grafu – pro nejkr. cestu  $p$  z  $s$  do  $v$  platí  $t(v_i) < t(v_{i+1})$  a pokud mám  $d(s, u) = \delta(s, u)$  a provedu Relax na  $v$  podle  $(u, v)$ , pak dostanu  $d(s, v) = \delta(s, v)$ , z čehož se korektnost dá dokázat indukcí podle počtu hran na cestě.

Složitost algoritmu je  $\Theta(n + m)$ , protože taková je složitost topologického třídění a zbytek algoritmu každou hranu i každý vrchol testuje právě 1x.

#### Algoritmus (*Dijkstrův algoritmus*)

Pracuje na libovolném orientovaném grafu s nezáporným ohodnocením hran.

```
Dijkstra( V - vrcholy, E - hrany, s - startovací vrchol ){

  inicializace - nastav d(s,v) = nekonečno pro všechny vrcholy;
  S = prázdný; // množina "vyřízených" vrcholů
  Q = V;      // množina "nevyřízených" vrcholů

  while( Q není prázdná ){
    vyber u, vrchol s nejmenším d z množiny Q;
    vlož vrchol u do S;
    foreach( v, z u do v vede hrana )
      proved' operaci Relax pro v přes u;
  }
}
```

Časová složitost při implementaci množin  $S$  a  $Q$  pomocí haldy je:  $\Theta(n \cdot \log n)$  pro inicializaci ( $n$  vložení do haldy),  $\Theta(n \cdot \log n)$  celkem pro vybírání prvků s nejmenším

$d$ , jedno provedení Relax při změně  $d$  trvá  $\Theta(\log n)$  (úprava haldy) a provede se max.  $m$ -krát; tedy celkem  $\Theta((m + n) \cdot \log n)$ .

### Algoritmus (Bellman-Ford)

Bellman-Fordův algoritmus lze použít nejobecněji, ale je nejpomalejší. Funguje na libovolném grafu (pokud najde cyklus, jehož celková váha je záporná, a tedy nejkratší cesty nemají smysl, vrací chybu).

```

Bellman-Ford( V - vrcholy, E - hrany, s - startovací vrchol ){

    inicializace - nastav d(s,v) = nekonečno pro všechny vrcholy;
    d(s,s) = 0;

    // n-1 iterací, každá projde všechny hrany
    for( i = 1; i < |V|; ++i ) {
        foreach( hrana (u,v) z E )
            proved' operaci Relax pro v přes u;
    }

    // hledání záporného cyklu
    foreach( hrana (u,v) z E ){
        if ( d(v) > d(u) + w(u,v) ){
            chyba - záporný cyklus;
            return;
        }
    }
}

```

Složitost algoritmu je  $\Theta(m \cdot n)$ . Vždy najde nejkratší cestu, protože v grafu bez záporných cyklů může mít cesta max.  $n - 1$  vrcholů. Důkaz nalezení záporného cyklu sporem, se sumou vah všech hran na něm (položím  $< 0$ ).

### Poznámka (Nejkratší cesty pro všechny dvojice vrcholů)

Pro hledání nejkratších cest pro všechny dvojice vrcholů lze buď použít  $n$ -krát běh některého z předchozích algoritmů, nebo *Algoritmus „násobení matic“* či *Floyd-Warshallův algoritmus*. Ty oba používají matice sousednosti  $W_G$  a počítají matici vzdáleností  $D_G$ .

První z nich postupuje indukcí podle počtu hran na nejkr. cestě, vyrábí matice  $D_G(x)$  pro  $x$  hran na nejkratší cestě.  $D_G(1)$  je  $W_G$ , pro výpočet kroku  $i$  vždy  $D_G(i - 1)$  „vynásobí“  $D_G(1)$  použitím zvláštního „násobení“, kde násobení hodnot je nahrazeno sčítáním a sčítání výběrem minima. Složitost je s využitím asociativity takto definovaného „násobení“  $\Theta(n^3 \log n)$ .

Floyd-Warshallův algoritmus jde indukcí podle velikosti množiny vrcholů, povolených jako vnitřní vrcholy na cestách. Používá  $d_{u,v}(k)$  jako min. váhu cesty z  $u$  do  $v$  s vnitř. vrcholy z množiny  $\{1, \dots, k\}$ . V iniciálním kroku je taky  $D_G(1) = W(G)$ . Pro  $i$ -tý krok je  $d_{u,v}(i) = \min\{d_{u,v}(i - 1), d_{u,i}(i - 1) + d_{i,v}(i - 1)\}$ . Složitost je  $\Theta(n^3)$ , navíc jeden krok je velice rychlý – celkově je algoritmus většinou rychlejší

než Bellman-Fordův a pro záporné cykly se časem na diagonále objeví záp. číslo, proto je není třeba testovat předem.

### Minimální kostra grafu

Úkolem v této úloze je najít kostru  $T$  (acyklický souvislý podgraf) grafu  $(V, E)$  s celkovou minimální vahou hran. Vždy platí  $|T| = |V| - 1$ . Bez újmy na obecnosti lze předpokládat, že ohodnocení hran jsou nezáporná (lze ke všem přičíst konstantu a výsledek se nezmění).

**Algoritmus** (*Borůvkův / Kruskalův algoritmus*)

```

Borůvka( V - vrcholy, E - hrany ){

    S = setříděné hrany podle jejich váhy;
    přiřaď vrcholům čísla komponent souvislosti;
    F = {};          // tj. (V,F) je "les", kde každý vrchol je
                    // jedna komponenta souvislosti

    while( S není prázdná ){
        vyber z S další hranu (x,y);
        if ( číslo komponenty x != číslo komponenty y ){
            F += (x,y);
            slij komponenty příslušné k x a y;
        }
    }
    return ( (V,F) jako minimální kostru (v,E) );
}

```

Celková složitost je  $\Theta(m \log m)$  při použití spojových seznamů: Setřídění hran podle váhy  $\Theta(m \log m)$ , nalezení čísla komponenty konstantní čas, max. počet přečíslování komponent při slévání (přečíslovávám-li vždy menší ze sléváných komponent) pro 1 vrchol je  $\Theta(\log n)$ , tj. celkem  $\Theta(n \log n)$ .

Algoritmus je korektní - vždy nalezne kostru, protože přidá právě  $|V| - 1$  hran a nevytvoří nikdy cyklus. Minimalita kostry se dokáže sporem - mám-li  $F$  vrácenou algoritmem a  $H$  nějakou min. kostru, tak pokud je  $w(F) > w(H)$ , najdu hranu  $e \in F \setminus H$ , vezmu kostru  $H_1 = H \cup e \setminus f$  (a  $w(e) \leq w(f)$ ). Pokud mám  $\forall e$  nalezené  $f$  takové, že  $w(e) = w(f)$ , jsou  $F$  i  $H$  minimální, jinak  $H$  taky nebylo minimální, protože  $H_1$  je menší.

**Algoritmus** (*Jarníkův / Primův algoritmus*)

```

Jarník( V - vrcholy, E - hrany, r - startovní vrchol ){

    Q = V;          // množina používaných vrcholů, dosud nepřipojených
                    // ke kostře
    F = {};          // vznikající kostra, v každém okamžiku
                    // je strom

```

```

inicializace - nastav klíč(v) na nekonečno
    pro všechny vrcholy;
klíč(r) = 0;
soused(r) = NULL;

while( Q je neprázdná ){
    vyber u, prvek s nejmenším klíčem z Q;
    F += ( soused(u),u );
    foreach( vrchol v, z u do v vede hrana ){
        if ( v je v Q a klíč(v) > w(u,v) ){
            klíč(v) = w(u,v);
            soused(v) = u;
        }
    }
}
return ( (V,F) jako min. kostru (V,E) );
}

```

Složitost algoritmu je  $\Theta(m \log n)$ , pokud je  $Q$  reprezentováno jako bin. halda - nejvýše  $m$ -krát upravuji klíč nějakého vrcholu, což má v haldě složitost  $\Theta(\log n)$ , výběr minima max.  $n$ -krát  $\Theta(\log n)$  a inicializace jen  $\Theta(n)$ .

Vytvořený graf je kostra, protože nikdy nevzniká cyklus (připojuji právě vrcholy z  $Q$ , která je na konci prázdná). Důkaz minimality podle konstrukce – najdu první hranu  $e$  v min. kostře  $H$ , která není ve výsledku alg.  $F$ , pak najdu  $f \in H$ , t.ž.  $F \setminus e \cup f$  je kostra, z algoritmu je  $w(f) \geq w(e)$ . Vezmu  $H_1 = H \setminus f \cup e$ , vím, že  $w(H_1) \leq w(H)$  a tedy  $H_1$  je min. kostra, iterací tohoto postupně dostanu, že  $H_k = F$  je min. kostra.

## Toky v sítích

*není požadováno v IP a ISPS*

**Definice** (*Síť, tok*)

*Síť* je čtveřice  $(G, z, s, c)$ , kde  $G$  je (orientovaný) graf,  $z$  zdrojový a  $s$  cílový vrchol (stok, spotřebič) a  $c : E \rightarrow \mathbb{R}^+$  funkce kapacity hran. *Tok* sítě je taková funkce  $t : E \rightarrow \mathbb{R}^+$ , že pro každou hranu  $(u, v)$  je  $0 \leq t((u, v)) \leq c((u, v))$  a navíc pro každý vrchol  $v$  kromě  $z$  a  $s$  (*uzel sítě*) platí  $\sum_{e=(u,v)} t(e) = \sum_{e=(v,w)} t(e)$  (tj. *přebytek toku* - rozdíl toho co do vrcholu vteče a co z něj odteče  $\delta(t, v)$  je pro uzly sítě nulový). *Velikost toku* se definuje jako  $|t| = \delta(t, s)$ .

**Algoritmus** (*Ford-Fulkersonův algoritmus*)

Algoritmus používá myšlenku zlepšitelné cesty - tj. pokud existuje v grafu neorientovaná cesta ze  $z$  do  $s$  taková, že pro hrany ve směru od zdroje je  $t < c$  a pro hrany ve směru ke zdroji  $t > 0$ , pak mohou tok zlepšit (o minimum rezerv). Algoritmus opakuje takovýto krok, dokud je možné ho provést. Neřeší výběr cesty, proto je dost pomalý a pokud nejsou hodnoty  $t$  racionální čísla, může se i zacyklit.

Ve chvíli zastavení algoritmu získám max. tok, neboť množina  $A = \{v \mid \text{ze } z \text{ do } v \text{ vede zlepšitelná cesta}\}$  je v tom okamžiku řez (množina  $A \subset V$  taková, že  $z \in V, s \notin V$ ) a jeho velikost ( $\sum_{e \in E} c(e), e = (u, v), u \in A, v \notin A$ ) je stejná jako velikost získaného toku.

### Algoritmus (*Dinitzův algoritmus*)

Řeší výběr zlepšitelné cesty – vybírá vždy nejkratší cestu (což obecně popisuje *Edmunds-Karpův algoritmus*). Dinitzova varianta používá *sítě rezerv*, což je graf  $(V, R)$ , kde hrana  $e = (v, w) \in R$ , pokud má tok hranou kladnou rezervu, tj.  $r = c(v, w) - t(v, w) + t(w, v) > 0$ . Zlepšující cesta odpovídá normální orientované cestě v síti rezerv. Převod na pův. graf ze sítě rezerv je jednoduchý, mohu předpokládat, že jedním ze směrů mezi dvěma vrcholy neteče nic.

Průběh algoritmu: na začátku nastaví všem hranám rezervu  $r(v, w) = c(v, w)$ . Potom postupuje po *fázích* - v 1 fázi:

- Vyhodí ze sítě rezerv všechny hrany, které nejsou na nejkratší cestě  $z \rightarrow s$  (2x prohledávání do šířky).
- Vezme jednu z nejkr. cest v síti rezerv a zlepší podle ní tok.
- Vyhodí vzniklé slepé cesty v síti rezerv (testují jen hrany, co vyhazují, a jejich konc. vrcholy)
- Toto opakuje, dokud jsou v síti rezerv cesty  $z \rightarrow s$  dané nejkratší délky.

Další fázi algoritmus pokračuje, dokud existuje vůbec nějaká cesta  $z \rightarrow s$  v síti rezerv. Fázi je tím pádem max.  $n$  (max. délka cesty ze  $z$  do  $s$ ), v 1 fázi se prochází max.  $m$  cest (klesá počet použitelných hran), nalezení 1 cesty je  $O(n)$  (jdu přímo) a vyhazování slepých cest max  $O(m)$  celkem za fázi (každou hranu vyhodím jen jednou). Celková složitost je tedy  $O(n^2m)$ .

### Algoritmus (*Goldbergův algoritmus (preflow-push, algoritmus vlny)*)

Nehledá v grafu zlepšující cesty, v průběhu výpočtu v grafu není tok, ale vlna (ze zdroje teče vždy více nebo rovno než max. tok). *Preflow* – „vlna“ – je funkce  $t : E \rightarrow \mathbb{R}^+$  taková, že  $\forall e \in E : 0 \leq t(e) \leq c(e)$ , tedy přebytky toku ve vrcholech ( $\delta$ ) jsou povolené. Ve chvíli, kdy žádný vrchol nemá přebytek toku ( $\delta$ ), dostávám (maximální) tok. Pro každý vrchol  $v$  si algoritmus pamatuje „výšku“  $h(v)$ . Také pracuje se sítě rezerv.

- *Inicializace*:  $h(z) = n, h(v, v \neq z) = 0, t(e) = 0 \forall e, \delta(v) = 0 \forall v$ .
- *Úvodní preflow*: převede ze zdroje maximum možného ( $t(e) = c(e)$  po směru) do sousedních vrcholů.
- *Hlavní cyklus*: opakuje se, dokud existuje vnitřní vrchol  $v$  s kladným  $\delta$ . pro vrchol  $v$ :
  - pokud existuje hrana  $(v, w)$  nebo  $(w, v)$ , t.ž.  $r(e) > 0$  (v daném směru) a  $h(v) \geq h(w)$ , potom se převede  $\min(\delta(v), r(e))$  z  $v$  do  $w$ .
  - jinak se zvýší  $h(v)$  o 1.

Po celou dobu běhu algoritmu platí invariant  $e = (v, w), r(e) > 0 \Rightarrow h(v) \leq 1 + h(w)$ . To zaručuje, že nalezený tok po zastavení je maximální (zdroj je ve výšce  $n$ , stok 0, tedy každá cesta překonává někde rozdíl  $-2$ ). Vrcholy nejde zvedat

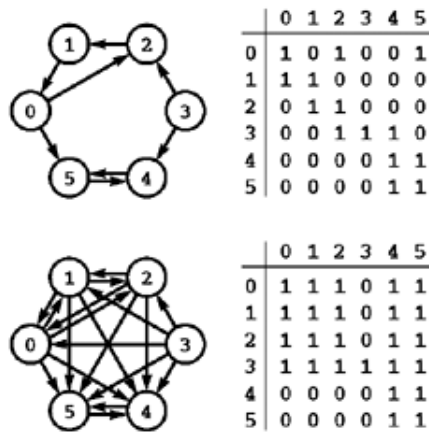
donekonečna, takže se algoritmus zastaví: pro každý vnitřní vrchol  $v$  platí, že je-li  $\delta(v) > 0$ , pak existuje v síti rezerv cesta  $v \rightarrow z$ . To zaručuje, že  $h(v) \leq 2n - 1$  - pokud mám vrchol  $v$  tak, že  $h(v) = 2n - 1$  a  $\delta(v) > 0$ , potom existuje cesta  $v \rightarrow z$  s kladnými rezervami a podle invariantu jde každá hrana na ní max. o 1 nahoru (tedy max. o  $n - 1$  celkem).

Složitost Goldbergova algoritmu je  $O(n^2 \cdot m)$ .

## 2.7 Tranzitivní uzávěr

### Definice

**Tranzitivní uzávěr** orientovaného grafu je orientovaný graf s původními vrcholy a platí, že existuje hrana z uzlu  $u$  do uzlu  $v$  právě tehdy, když v původním orientovaném grafu existuje libovolná orientovaná cesta z uzlu  $u$  do uzlu  $v$ .



Obr. 1: Tranzitivní uzávěr grafu (zdroj: <http://zorro.fime.vutbr.cz/graphs/foil36.html>)

### Poznámka

Platí, že matice dosažitelnosti v grafu  $G$  = matice sousednosti tranzitivního uzávěru grafu  $G$ .

### Algoritmus

Z každého vrcholu vypustit DFS (Depth-first search – prohledávání do hloubky), do společné matice zaznamenávat dosažené vrcholy (řádek odpovídá vrcholu, sloupce vrcholům, které jsou z něho dosažitelné) – složitost  $O(n(n + m))$ .

### Warshallův algoritmus

Iterativní konstrukce matice dosažitelnosti, postupně počítá matice  $W_k$ , kde  $w_{i,j}^{[k]} = 1$ , pokud mezi vrcholy  $i$  a  $j$  existuje cesta, jejíž všechny vnitřní vrcholy jsou vrcholy  $1 \dots k$ .

Z matice  $W_k$  lze spočítat matici  $W^{[k+1]}$  :  $W_{i,j}^{[k+1]} = W_{i,j}^{[k]} \vee (W_{i,k+1}^{[k]} \wedge W_{k+1,j}^{[k]})$  – buď vede mezi vrcholy  $i, j$  cesta, která nepoužije vrchol  $k + 1$ , nebo taková, která ho použije – v tom případě ale musí vést cesty mezi vrcholy  $i, k + 1$  a  $k + 1, j$ , které používají pouze vrcholy  $1 \dots k$ , jejich spojením je cesta mezi vrcholy  $i, j$

Matice  $W^1$  je matice incidence pôvodného grafu.  
Pseudokód (vstup: I – matice incidence,  $[0, 1]^{n \times n}$ ):

```
Procedure Warshall(I)
W:= I;
for k:=1 to n
begin
  for i:=1 to n
  begin
    for j:=1 to n
       $w_{i,j} = w_{i,j} \vee (w_{i,k} \wedge w_{k,j})$ 
    end
  end
end
return W;
```

Složitost algoritmu je jasne  $O(n^3)$  (potrebuje  $2n^3$  bitových operáci), což může být lepší pro grafy s hodně hranami (počet hran se blíží  $n^2$ ), než složitost  $n * DFS$  ( $n * (n + m) \approx n * (n + n^2) = n^2 + n^3$ )

TODO: ještě něco?

## 2.8 Algoritmy vyhľadávání v textu

Toto sú len veľmi stručné výťahy z wikipedie. Aktuálne sú tu len preto, aby si človek rýchlo vybavil, o čom tie algoritmy sú :-)

### Rabin-Karp

Umožňuje vyhľadávanie viacerých reťazcov v texte naraz - užitočné napr. na hľadanie plagiátov. Základnou myšlienkou je vyhľadávanie v texte pomocou hashov (rolling hashes - idea je  $s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m]$ )...

Algoritmus pre vyhľadávanie jedného reťazca:

```
1 function RabinKarp(string s[1..n], string sub[1..m])
2   hsub := hash(sub[1..m])
3   hs := hash(s[1..m])
4   for i from 1 to n-m+1
5     if hs = hsub
6       if s[i..i+m-1] = sub
7         return i
8     hs := hash(s[i+1..i+m])
9   return not found
```

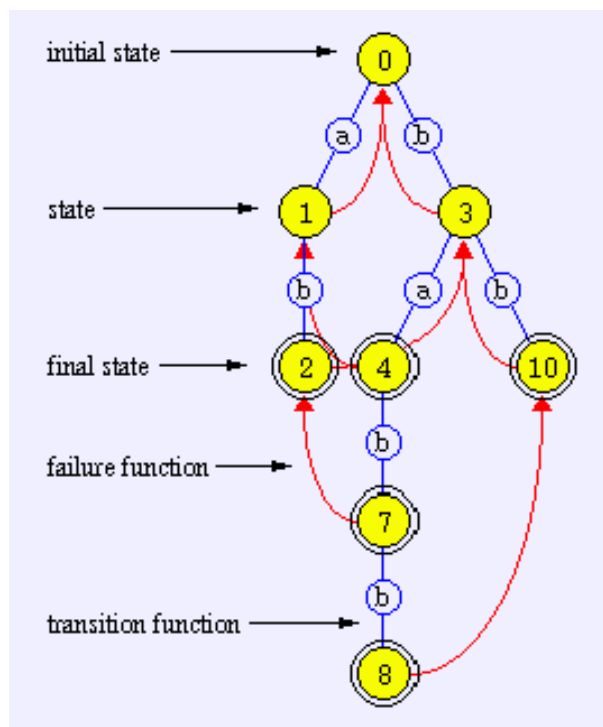
Najhoršia zložitost' je  $\Omega(mn)$ . Pri vyhľadávaní viacerých reťazcov len spočítame hashe všetkých hľadaných stringov a pri nájdení niektorého z hashov príslušný reťazec porovnáme s textom... Ostatné algoritmy spotrebujú čas  $O(n)$  na nájdenie 1 reťazca a teda  $O(nk)$  na vyhľadanie  $k$  reťazcov. Naproti tomu tento algoritmus má očakávanú zložitost'  $O(n+k)$  - pretože vyhľadávanie v hashovacej tabulke, či je hash podreťazca textu rovný hashu niektorého z hľadaného reťazcov, trvá  $O(1)$ .

## Aho-Corasick

Dokáže vyhľadávať viacero reťazcov naraz - používa na to trie-like štruktúru (konečný automat), ktorý obsahuje nasledujúce „prvky“:

1. konečná množina  $Q$  - stavy
2. konečná abeceda  $A$
3. transition funkcia  $g: Q \times A \rightarrow Q + \{fail\}$
4. failure funkcia  $h: Q \rightarrow Q + \{fail\}$ .  $h(q) = q'$  práve vtedy keď spomedzi všetkých stavov  $Q$  dáva  $q'$  najdlhší suffix z  $path(q)$ .
5. konečná množina  $F$  - koncové stavy

Príklad „hotového“ automatu pre slová  $P = \{ab, ba, babb, bb\}$ :



Zložitosť vyhľadávania je lineárna vzhľadom k dĺžke textu a počtu nájdených „slov“ (pozn.: ten môže byť až kvadratický - slovník a, aa, aaa, aaaa; reťazec aaaa). Trie štruktúru je možné vyrobiť raz a potom používať počas vyhľadávania - uchováваме si najdlhší match a používame suffix odkazy (aby sme udržali linearitu výpočtu).

Výstavba stromu sa provede prostým zařazovaním slov do trie-stromu podľa prefixů. Na tejto štruktúre je potom možné v lineárnom čase (vzhľadom k počtu znakov hľadaných slov) predpočítať hodnoty failure funkcie: automat vždy pustíme na suffix aktuálne zkoušeného slova, bez prvého znaku. Díky tomu, že průběžně ukládáme hodnoty nálezéných slov, pro každé písmeno provede max. 2 kroky (postup vpřed a uložení hodnoty, kam bych spadnul).

## Knuth-Morris-Pratt

Obdoba Aho-Corasick, ale hľadá len jedno slovo. Samozrejme nie je potrebná dopredná funkcia (vždy iba nasledujúci znak), používa sa „partial match“ tabuľka

(failure funkcia).

algorithm kmp\_search:

input:

S (the text to be searched)

W (the word sought)

m = 0 (the beginning of the current match in S)

i = 0 (the position of the current character in W)

an array of integers, T (the table, computed elsewhere)

while m + i is less than the length of S, do:

if W[i] = S[m + i],

i = i + 1

if i equals the length of W,

return m

otherwise,

m = m + i - T[i],

if i is greater than 0,

i = T[i]

(if we reach here, we have searched all of S unsuccessfully)

return the length of S

Zložitosť algoritmu je  $O(k)$  (k je dĺžka S) - cyklus je vykonaný najviac  $2k$  krát.

Algoritmus na výrobu tabuľky:

algorithm kmp\_table:

input:

W (the word to be analyzed)

T (the table to be filled)

i = 2 (the current position we are computing in T)

j = 0 (the zero-based index in W of the next  
character of the current candidate substring)

(the first few values are fixed but different

from what the algorithm might suggest)

let T[0] = -1

T[1] = 0

while i is less than the length of W, do:

(first case: the substring continues)

if W[i - 1] = W[j],

T[i] = j + 1

i = i + 1

j = j + 1

(second case: it doesn't, but we can fall back)

otherwise, if  $j > 0$ ,

$$j = T[j]$$

(third case: we have run out of candidates. Note  $j = 0$ )

otherwise,

$$T[i] = 0$$

$$i = i + 1$$

Zložitost tohoto algoritmu je  $O(n)$  ( $n$  je délka  $W$ ) - cyklus skončí najviac po  $2n$  iteráciách.

## 2.9 Algebraické algoritmy

### Diskrétní Fourierova Transformace (DFT)

Diskrétní Fourierova transformace se používá, chceme-li zachytit hodnotu (přepokládejme, že  $2\pi$ -periodické) funkce na intervalu  $[-\pi, \pi]$  v nějakých  $n$  bodech. To je dobré např. pro vzorkování elektrického nebo zvukového signálu a jiné operace. Pro nějakou funkci nám tak stačí znát vektor dimenze  $n$  (a  $n$  je počet vzorků na  $2\pi$ ).

Je to založeno na Fourierových řadách – dá se ukázat, že funkce  $1$ ,  $\cos kx$  a  $\sin kx$  pro  $k \geq 1$  tvoří ortogonální bázi prostoru spojitých funkcí na intervalu  $[-\pi, \pi]$ . Protože potřebujeme znát jenom konečný počet vzorků, stačí nám jen konečný podprostor s konečnou bází. Máme-li rozklad nějaké  $2\pi$ -periodické funkce do Fourierovy řady  $f(x) = c + \sum_{k=1}^{\infty} a_k \sin kx + \sum_{k=1}^{\infty} b_k \cos kx$ , dá se jednoduše ukázat, že pro hodnoty v bodech  $-\pi, -\pi + \frac{\pi}{n}, -\pi + 2\frac{\pi}{n}, \dots, -\pi + (n-1)\frac{\pi}{n}$  stačí sumy do  $\frac{n}{2} - 1$  pro sinusové řady a  $\frac{n}{2}$  pro kosinové – vyšší koeficienty v takových bodech jsou nulové. Takže  $n$  hodnot funkce  $f$  na intervalu  $[-\pi, \pi]$  lze reprezentovat vektorem  $n$  čísel n bázi  $1, \cos x, \dots, \cos \frac{n}{2}x, \sin x, \dots, \sin(\frac{n}{2} - 1)x$ .

Jednodušeji to lze ukázat v komplexních číslech – je známo, že

$$e^{ix} = \cos x + i \cdot \sin x$$

takže vektor hodnot funkce lze ekvivalentně reprezentovat v bázi  $e^{i \cdot 2\pi \frac{k}{n}}$ ,  $k \in \{0, \dots, n\}$ , neboť všechny vektory původní báze lze zapsat jako lineární kombinace vektorů nové báze. Definujeme hodnotu

$$\omega := e^{i \cdot 2\pi \frac{1}{n}} \text{ (a to je vlastně „něco jako“ } \sqrt[n]{1}\text{)}$$

vidíme, že  $\omega^k$  je  $n$ -periodická funkce, takže nezáleží na hranicích sumace ( $-\frac{n}{2} + 1, \dots, \frac{n}{2}$  je ekvivalentní  $0, \dots, n-1$ ). Potom se posloupnost  $n$  komplexních čísel  $\alpha_0, \dots, \alpha_{n-1}$  (např. hodnot naší funkce v bodech  $-\pi + \frac{2\pi k}{n}$ ,  $k \in \{0, \dots, n-1\}$ ) transformuje na posloupnost  $n$  komplexních čísel  $A_0, \dots, A_{n-1}$  (do báze  $\omega^i$ ,  $i \in \{0, \dots, n-1\}$ ) použitím vzorečku:

$$A_j = \sum_{k=0}^{n-1} \alpha_k \omega^{kj} \quad j = 0, \dots, n-1$$

Tento převod označujeme jako *diskrétní Fourierovu transformaci*.

*Inverzní diskrétní Fourierova transformace* je opačný problém – z  $n$  Fourierových koeficientů  $A_k$  chceme zpětně vypočítat hodnoty funkce  $\alpha_k$  v bodech  $-\pi + \frac{2\pi k}{n}$ ,  $k \in \{0, \dots, n-1\}$ . Platí:

$$\alpha_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k \omega^{-kj} \quad j = 0, \dots, n-1$$

### Důkaz

Definujeme matici  $W : W_{p,q} = \omega^{pq}$ , potom  $A = W\alpha$  (vektorově), takže  $a = W^{-1}A$ . Definujeme  $W' : W'_{p,q} = \omega^{-pq}$  a dokážeme, že  $W \cdot W' = n \cdot I_n$ . Máme

$$(W \cdot W')_{p,q} = \sum_{s=0}^{n-1} W_{p,s} \cdot W'_{s,q} = \sum_{s=0}^{n-1} \omega^{(p-q) \cdot s}$$

a potom pro

- $p = q$  platí  $\sum_{s=0}^{n-1} \omega^{(p-q) \cdot s} = \sum_{s=0}^{n-1} \omega^0 = \sum_{s=0}^{n-1} 1 = n$
- $p \neq q$  definujeme

$$Q := \omega^{p-q}$$

a dostaneme geometrickou posloupnost  $Q^0 + Q^1 + \dots + Q^{n-1}$ , pro jejíž součet prvních  $n$  členů platí vzorec

$$\sum_{s=0}^{n-1} Q^s = Q^0 \frac{Q^{n-1+1} - 1}{Q - 1} = 1 \frac{1 - 1}{Q - 1} = 0$$

### Algoritmus (*Fast Fourier transform (FFT)*)

Fast Fourier transform je algoritmus pro počítání diskrétní Fourierovy transformace vektorů rozměru  $n = 2^k$  v čase  $\Theta(n \log n)$ . Mám-li matici Fourierových koeficientů  $W$ ,  $W_{p,q} = \alpha_q \omega^{pq}$ , mohu ji rozdělit na liché a sudé sloupce, u sudých vyjádřit  $\omega^q$  a pro spodní polovinu řádek (se sumami jdoucími po dvou) mohu snížit exponent u  $\omega$  o  $n/2$  (díky periodicitě) a vyjdou stejná čísla:

$$A_j = \sum_{k=0}^{n-1} \alpha_k \omega^{kj} \quad j \in \{0, \dots, n-1\}$$

$$A_j = \sum_{k=0}^{\frac{n}{2}-1} \alpha_{2k} \omega^{2kj} + \omega^j \sum_{k=0}^{\frac{n}{2}-1} \alpha_{2k+1} \omega^{2kj} \quad j \in \{0, \dots, \frac{n}{2}-1\}$$

$$A_{j+\frac{n}{2}} = \sum_{k=0}^{\frac{n}{2}-1} \alpha_{2k} \omega^{2k(j+\frac{n}{2})} + \omega^{(j+\frac{n}{2})} \sum_{k=0}^{\frac{n}{2}-1} \alpha_{2k+1} \omega^{2k(j+\frac{n}{2})} \quad j \in \{0, \dots, \frac{n}{2}-1\}$$

*Poznámka: pro rychlé a jednoduché pochopení těch blektů co jsem tu napsal doporučuji Kučerův program Algovision*

<http://kam.mff.cuni.cz/~ludek/AlgovisionPage.html>  
*DFT je tam názorně a přehledně ukázaná.*

TODO: Související obecné „věci“ o Fourierově transformaci, použití při spektrální analýze (Nyquist-Shannon sampling theorem), datové kompresi (Diskrétní kosinová transformace), násobení polynomů (+násobení velkých integerů).

## Euklidův algoritmus

Euklidův algoritmus je postup (algoritmus), kterým lze určit největšího společného dělitele dvou přirozených čísel, tzn. nejvyšší číslo takové, že beze zbytku dělí obě čísla.

Algoritmus (pomocí rekurze):

```
function gcd(a, b)
  if b = 0 return a
  else return gcd(b, a mod b)
```

Algoritmus (pomocí iterace):

```
function gcd(a, b)
  while b ≠ 0
    t := b
    b := a mod b
    a := t
  return a
```

Algoritmus (jednoduchý ale neefektivní):

```
function gcd(a, b)
  while b ≠ 0
    if a > b
      a := a - b
    else
      b := b - a
  return a
```

Doba provádění programu je závislá na počtu průchodů hlavní smyčkou. Ten je maximální tehdy, jsou-li počáteční hodnoty  $u$  a  $v$  rovné dvěma po sobě jdoucím členům Fibonacciho posloupnosti. Maximální počet provedených opakování je tedy  $\log_{\phi}(3 - \phi)v \approx 4,785 \log v + 0,6273 = O(\log v)$ . Průměrný počet kroků pak je o něco nižší, přibližně  $\frac{12 \ln 2}{\pi^2} \log v \approx 1,9405 \log v = O(\log v)$ .

## 2.10 Základy kryptografie, RSA, DES

### Základy kryptografie

TODO

## RSA (Rivest-Shamir-Adleman)

Asymetrická šifra (různé klíče pro šifrování a dešifrování), použitelná jako šifra s veřejným klíčem.

### Inicializace:

1. vybrat dvě dostatečně velká prvočísla  $p, q$
2.  $n := p \cdot q$
3. spočítat totient:  $\varphi(n) := (p - 1) \cdot (q - 1)$   
(Eulerův totient  $\varphi(n)$  je počet čísel menších než  $n$ , která jsou s  $n$  nesoudělná)
4. vybrat  $e$  takové, že  $1 < e < \varphi(n)$  a  $e$  je nesoudělné s  $\varphi(n)$   
–  $e$  bude *veřejný klíč (public key)*
5. vybrat  $d$  tak, aby

$$d \cdot e \equiv 1 \pmod{\varphi(n)}$$

takové  $d$  lze najít rozšířeným euklidovým algoritmem  
–  $d$  bude *dešifrovací klíč (private key)*

### Šifrování:

1. Alice posílá public key Bobovi (čísla  $n$  a  $e$ ), nechává si private key
2. Bob chce Alici poslat zprávu  $m$  (musí být převedena na celé číslo  $m < n$ )
3. Bob spočítá :

$$c = m^e \pmod{n}$$

4. Bob odešle  $c$  Alici

### Dešifrování:

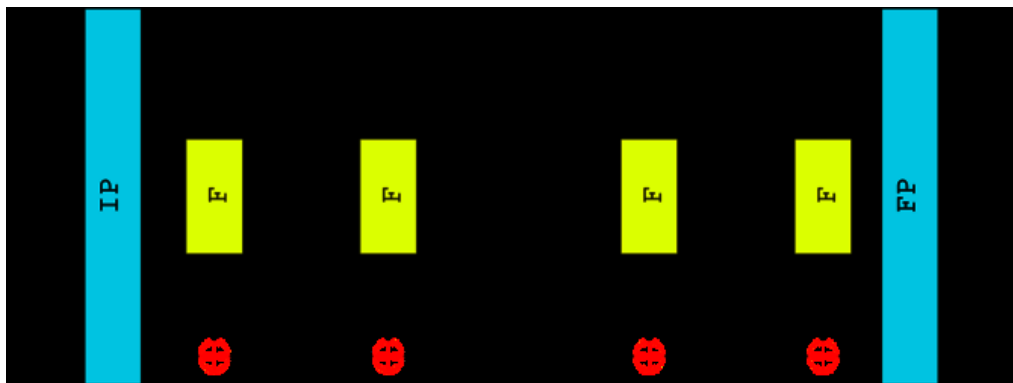
1. Alice přijala  $c$
2. Spočítá:

$$m = c^d \pmod{n}$$

Šifra (to, že to vůbec funguje, tedy, že  $m = (m^e)^d$ ) se opírá o několik netriviálních vět algebry...

## DES (Data Encryption Standard)

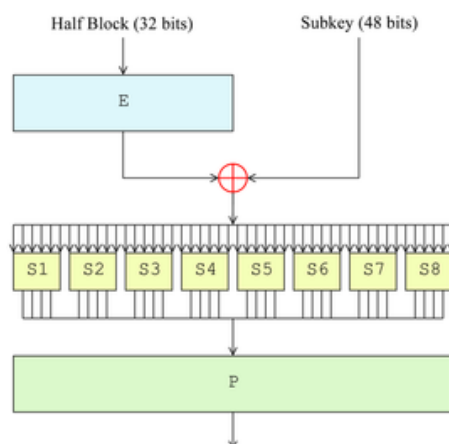
- bloková šifra (vstup plaintext - 64bitů, výstup ciphertext 64bitů)
- symetrický klíč (stejný pro šifrování i dešifrování) – strany si ho musí vyměnit po bezpečném kanále
- klíč – 64bitů, z nich se používá ale pouze 56bitů (zbytek se zahodí nebo funguje jako kontrola parity)
- původně implementována hardwarem
- stejný algoritmus (i hardware) použitý jak pro šifrování, tak pro dešifrování



Obr. 2: Schéma hlavní sítě algoritmu DES

### Šifrování:

- vstup projde iniciální permutací (IP:64b  $\rightarrow$  64b), na konci probíhá inverzní finální permutace (FP), následuje 16 identických kol šifrování:
- blok 64 bitů se rozdělí na dvě půlky po 32bitech,
  - pravá půlka slouží jako vstup pro funkci F a také je v dalším kole použita jako levá část
  - levá půlka se xoruje s výstupem funkce F a výsledek je použit v dalším kole jako pravá část
- celý cyklus se provede 16x a v závěru se ještě aplikuje finální permutace



Obr. 3: Funkce  $F$  v algoritmu DES

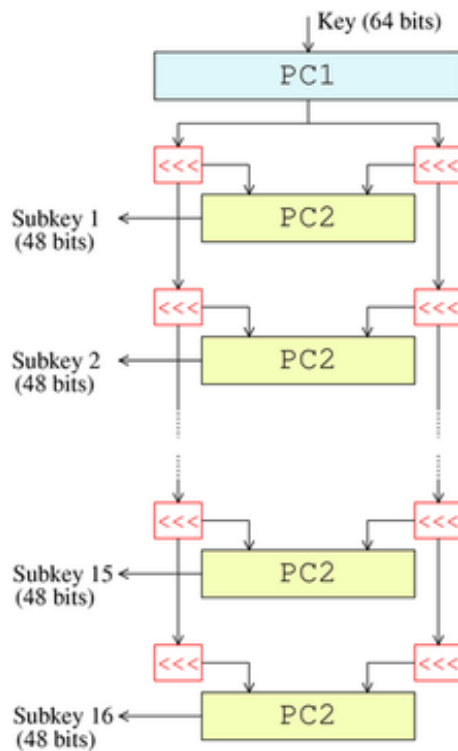
### Funkce F:

- ve funkci F probíhá míchání s klíčem. V každém kole vstupuje do funkce F 32bitů z „pravé půlky“ a 48bitový subklíč (odvozen z 56bitového klíče, detaily později).

- 32bitů z pravé půlky je nejprve expandováno na 48 bitů (fixní expanzní permutací, na obrázku označena E), potom xorováno se subklíčem.
- Výsledek xorování se rozdělí na 8 bloků po 6bitech. Každý blok je pak vstupem jedné z osmi S funkcí. Každá S funkce převádí 6 bitů na 4 bity (nelineární transformací, „zadrátované“).
- Výstupy S funkcí se opět spojí do jednoho bloku ( $8 \times 4 = 32$ bitů) – to je výsledek celé funkce F.

### Dešifrování:

Díky prohazování poloviček v jednotlivých kolech lze dešifrování provádět stejnou funkcí (na stejném hardwaru), jako šifrování. Pouze je potřeba používat subklíče v opačném pořadí.



Obr. 4: Subklíče v algoritmu DES

### Subklíče:

- protože je klíč původně 64bitový, ale ve skutečnosti se používá pouze 56bitů (ostatní se zahazují, nebo slouží pro kontrolu parity), nejprve je vybráno těchto 56bitů funkcí PC1 (Permuted Choice 1)
- Dále se vždy pro každé kolo 56 bitů rozdělí na dvě půlky po 28bitech. Každá z těchto půlek se bitově posune doleva (o jeden nebo dva bity, to je pevně určeno pro každé kolo). Takto posunuté půlky se vloží jako vstup funkce PC2, která vygeneruje 48bitový subklíč. Obě půlky také slouží jako vstup pro další kolo.

- Algoritmus zaručuje, že každý bit z původního 56bitového klíče je použit asi ve 14-ti ze 16-ti subklíčů.
- Pro dešifrování se klíče musí generovat v opačném pořadí (místo doleva se posouvá doprava).

## 3 Databáze

### Požadavky

- Podstata a architektury DB systémů
- Konceptuální, logická a fyzická úroveň pohledů na data
- Algoritmy návrhu schémat relací, normální formy, referenční integrita
- Transakční zpracování, vlastnosti transakcí, uzamykací protokoly, zablokování
- ER-diagramy, metody návrhů IS
- SQL
- Indexy, trigger, uložené procedury, uživatelé, uživatelská práva
- Vícevrstevné architektury
- Vazba databází na internetové technologie
- Organizace dat na vnější paměti, B-stromy a jejich varianty.

### 3.1 Podstata a architektury DB systémů

Zdroje: Wikipedie, slidy Dr. T. Skopala k Databázovým systémům

#### **Definice** (*Databáze*)

Databáze je logicky uspořádaná (integrovaná) kolekce navzájem souvisejících dat. Je sebevysvětlující, protože data jsou uchovávána společně s popisy, známými jako metadata (také schéma databáze). Data jsou ukládána tak, aby na nich bylo možné provádět strojové dotazy – získat pro nějaké parametry vyhovující podmnožinu záznamů.

Někdy se slovem „databáze“ myslí obecně celý databázový systém.

#### **Definice** (*Systém řízení báze dat*)

Systém řízení báze dat (SŘBD, anglicky database management system, DBMS) je obecný softwarový systém, který řídí sdílený přístup k databázi, a poskytuje mechanismy, pomáhající zajistit bezpečnost a integritu uložených dat. Spravuje databázi a zajišťuje provádění dotazů.

#### **Definice** (*Databázový systém*)

Databázovým systémem rozumíme trojici, sestávající z:

- databáze
- systému řízení báze dat
- chudáka admina

## Smysl databázi

Hlavním smyslem databáze je schraňovat datové záznamy a informace za účelem:

- sdílení dat více uživatelům,
- zajištění unifikovaného rozhraní a jazyků definice dat a manipulace s daty,
- znovuvyužitelnosti dat,
- bezspornosti dat a
- snížení objemu dat (odstranění redundance).

## Databázové modely

### Definice (*schéma, model*)

Typicky pro každou databázi existuje strukturální popis druhů dat v ní udržovaných, ten nazýváme *schéma*. Schéma popisuje objekty reprezentované v databázi a vztahy mezi nimi. Je několik možných způsobů organizace schémat (modelování databázové struktury), známých jako *modely*. V modelu jde nejen o způsob strukturování dat, definuje se také sada operací nad daty proveditelná. Relační model například definuje operace jako „select“ nebo „join“. I když tyto operace se nemusejí přímo vyskytovat v dotazovacím jazyce, tvoří základ, na kterém je jazyk postaven. Nejdůležitější modely v této sekci popíšeme.

### Poznámka

Většina databázových systémů je založena na jednom konkrétním modelu, ale čím dál častější je podpora více přístupů. Pro každý logický model existuje více fyzických přístupů implementace a většina systémů dovolí uživateli nějakou úroveň jejich kontroly a úprav, protože toto má velký vliv na výkon systému. Příkladem nechť jsou indexy, provozované nad relačním modelem.

### „Plochý“ model

Toto sice nevyhovuje úplně definici modelu, přesto se jako triviální případ uvádí. Představuje jedinou dvoudimensionální tabulku, kde data v jednom sloupci jsou považována za popis stejné vlastnosti (takže mají podobné hodnoty) a data v jednom řádku se uvažují jako popis jediného objektu.

### Relační model

Relační model je založen na predikátové logice a teorii množin. Většina fyzicky implementovaných databázových systémů ve skutečnosti používá jen aproximaci matematicky definovaného relačního modelu. Jeho základem jsou *relace* (dvoudimensionální tabulky), *atributy* (jejich pojmenované sloupce) a *domény* (množiny hodnot, které se ve sloupcích mohou objevit). Hlavní datovou strukturou je tabulka, kde se nachází informace o nějaké konkrétní třídě entit. Každá entita té třídy je potom reprezentována řádkem v tabulce –  $n$ -ticí atributů.

Všechny relace (tj. tabulky) musí splňovat základní pravidla – pořadí sloupců nesmí hrát roli, v tabulce se nesmí vyskytovat identické řádky a každý řádek musí obsahovat jen jednu hodnotu pro každý svůj atribut. Relační databáze obsahuje více tabulek, mezi kterými lze popisovat vztahy (všech různých kardinalit, tj.

1 : 1, 1 :  $n$  apod.). Vztahy vznikají i implicitně např. uložením stejné hodnoty jednoho atributu do dvou řádků v tabulce. K tabulkám lze přidat informaci o tom, která podmnožina atributů funguje jako *klíč*, tj. unikátně identifikuje každý řádek, některý z klíčů může být označen jako primární. Některé klíče mohou mít nějaký vztah k vnějšímu světu, jiné jsou jen pro vnitřní potřeby schématu databáze (generovaná ID).

### Hierarchický model

V hierarchickém modelu jsou data organizována do stromové struktury – každý uzel má odkaz na nadřazený (k popisu hierarchie) a setříděné pole záznamů na stejné úrovni. Tyto struktury byly používány ve starých mainframeových databázích, nyní je můžeme vidět např. ve struktuře XML dokumentů. Dovolují vztahy 1 :  $N$  mezi dvěma druhy dat, což je velice efektivní k popisu různých reálných vztahů (obsahy, řazení odstavců textu, tříděné informace). Nevýhodou je ale nutnost znát celou cestu k záznamu ve struktuře a neschopnost systému reprezentovat redundance v datech (strom nemá cykly).

### Síťový model

Síťový model organizuje data pomocí dvou hlavních prvků, *záznamů* a *množin*. Záznamy obsahují pole dat, množiny definují vztahy 1 :  $N$  mezi záznamy (jeden *vlastník*, mnoho *prvků*). Záznam může být vlastníkem i prvkem v několika různých množinách. Jde vlastně o variantu hierarchického modelu, protože síťový model je také založen na konceptu více struktur nižší úrovně závislých na strukturách úrovně vyšší. Už ale umožňuje reprezentovat i redundantní data. Operace nad tímto modelem probíhají „navigačním“ stylem: program si uchovává svoji současnou pozici mezi záznamy a postupuje podle závislostí, ve kterých se daný záznam nachází. Záznamy mohou být i vyhledávány podle klíče.

Fyzicky jsou většinou množiny – vztahy – reprezentovány přímo ukazateli na umístění dat na disku, což zajišťuje vysoký výkon při vyhledávání, ale zvyšuje náklady na reorganizace. Smysl síťové navigace mezi objekty se používá i v objektových modelech.

### Objektový model

Objektový model je aplikací přístupů známých z objektově-orientovaného programování. Je založen na sbližování programové aplikace a databáze, hlavně ve smyslu použití datových typů (objektů) definovaných na jednom místě; ty zpřístupňuje k použití v nějakém běžném programovacím jazyce. Odstraní se tak nutnost zbytečných konverzí dat. Přináší do databází také věci jako zapouzdření nebo polymorfismus. Problémem objektových modelů je neexistence standardů (nebo spíš produktů, které by je implementovaly).

Kombinací objektového a relačního přístupu vznikají *objektově-relační* databáze – relační databáze, dovolující uživateli definovat vlastní datové typy a operace na nich. Obsahují pak hybrid mezi procedurálním a dotazovacím programovacím jazykem.

## Architektury databázových systémů

Zdroj: Wiki ČVUT (státnice na FELu ;-))

Architektury databázových systémů se obecně dělí na

- *centralizované* (kde se databáze předpokládá fyzicky na jednom počítači) a
- *distribuované*,

případně na

- *jednouživatelské* a
- *víceuživatelské*.

### Distribuované databázové systémy

*Distribuovaný systém řízení báze dat* je vlastně speciálním případem obecného distribuovaného výpočetního systému. Jeho implementace zahrnuje fyzické rozložení dat (včetně možných replikací databáze) na více počítačů – *uzlů*, přičemž jejich popis je integrován v globálním databázovém schématu. Data v uzlech mohou být zpracovávána lokálními SŘBD, komunikace je organizována v síťovém provozu pomocí speciálního softwaru, který umí zacházet s distribuovanými daty. Fyzicky se řeší rozložení do uzlů, svázaných komunikačními kanály, a jeho transparence (neviditelnost – navenek se má tvářit jako jednolitý systém). Každý uzel v síti je sám o sobě databázový systém a z každého uzlu lze zpřístupnit data kdekoliv v síti.

Dále se dělí na dva typy:

- Federativní databáze – neexistuje globální schéma ani centrální řídicí autorita, řízení je také distribuované.
- Heterogenní databázové systémy – jednotlivé autonomní SŘBD existují (vznikly nezávisle na sobě) a jsou integrovány, aby spolu mohly komunikovat.

Výhodou oproti centralizovaným systémům je vyšší efektivita (data mohou být uložena blízko místa nejčastějšího používání), zvýšená dostupnost, výkonnost a rozšiřitelnost; nevýhodou zůstává problém složitosti implementace, distribuce řízení a nižší bezpečnost takových řešení.

### Víceuživatelské databázové systémy

*Víceuživatelské* jsou takové systémy, které umožňují vícenásobný uživatelský přístup k datům ve stejném okamžiku. V důsledku možného současného přístupu více uživatelů je nutné systém zabezpečit tak, aby i nadále zajišťoval integritu a konzistenci uložených dat. Existují obecně dva možné přístupy:

- Uzamykání – Dříve často používaná metoda založená na uzamykání aktualizovaných záznamů, v případě masivního využití aktualizčních příkazů u ní ale může docházet k značným prodlevám.
- Multiversion Concurrency Control – Modernější vynález. Jeho princip spočívá v tom, že při požadavku o aktualizaci záznamu v tabulce je vytvořena kopie záznamu, která není pro ostatní uživatele až do provedení commitu viditelná.

## 3.2 Konceptuální, logická a fyzická úroveň pohledu na data

TODO: sjednotit terminologii, snad to popisuje to co tu má být, ale zdroje jsou pochybné (Wikipedie tady neodvádí zrovna ideální práci a ČVUT Wiki se moc nerozepisuje).

### Definice (*Datové modelování*)

*Datové modelování* je proces vytvoření konkrétního datového modelu (schématu) databáze pomocí aplikace nějakého abstraktního databázového modelu. Datové modelování zahrnuje kromě definice struktury a organizace dat ještě další implicitní nebo explicitní omezení na data do struktury ukládaná.

### Vrstvy modelování

Druhy datových modelů mohou být tři typů, podle tří různých pohledů na databáze (tři „vrstvy“, které se navzájem doplňují):

- konceptuální schéma (datový model) – nejabstraktnější, popisuje význam organizace databáze – třídy entit a jejich vztahy.
- logické schéma – popisuje význam konceptuálního schématu z hlediska databázové implementace – popisy tabulek, programových tříd nebo XML tagů (podle zvoleného databázového modelu)
- fyzické schéma – nejkonkrétnější, popisuje fyzické uložení dat a stroje na kterých systém poběží.

Na tomto rozdělení je důležitá nezávislost jednotlivých vrstev – takže se implementace jedné z nich může změnit, aniž by bylo nutné výrazně upravovat ostatní (samozřejmě musí zůstat konzistentní vzhledem k ostatním vrstvám). Během implementace nějaké databázové aplikace se začíná vytvořením konceptuálního schématu, pokračuje jeho upřesnění logickým schématem a nakonec jeho fyzickou implementací podle fyzického schématu (modelu).

### Poznámka

V tomto pohledu (který je podle standardu ANSI z r. 1975) jsou databázové modely, popsané v předchozí sekci, příklady abstraktních logických datových modelů. Někde je však tato úroveň označována jako „fyzická“ a „jiná logická“ se vtěsňuje ještě mezi ni a konceptuální.

### Konceptuální schéma

*Konceptuální schéma* (datový model) popisuje podstatné objekty (*třídy entit*, „koncepty“), jejich charakteristiky (*atributy*) a vztahy mezi nimi (asociace mezi dvojicemi tříd entit). Nepopisuje přímo implementaci v databázi, jen význam nějakého celku, který bude databází představován. Jde o modelování „datové reality“, z pohledu uživatele (analytika, konstruktéra databáze).

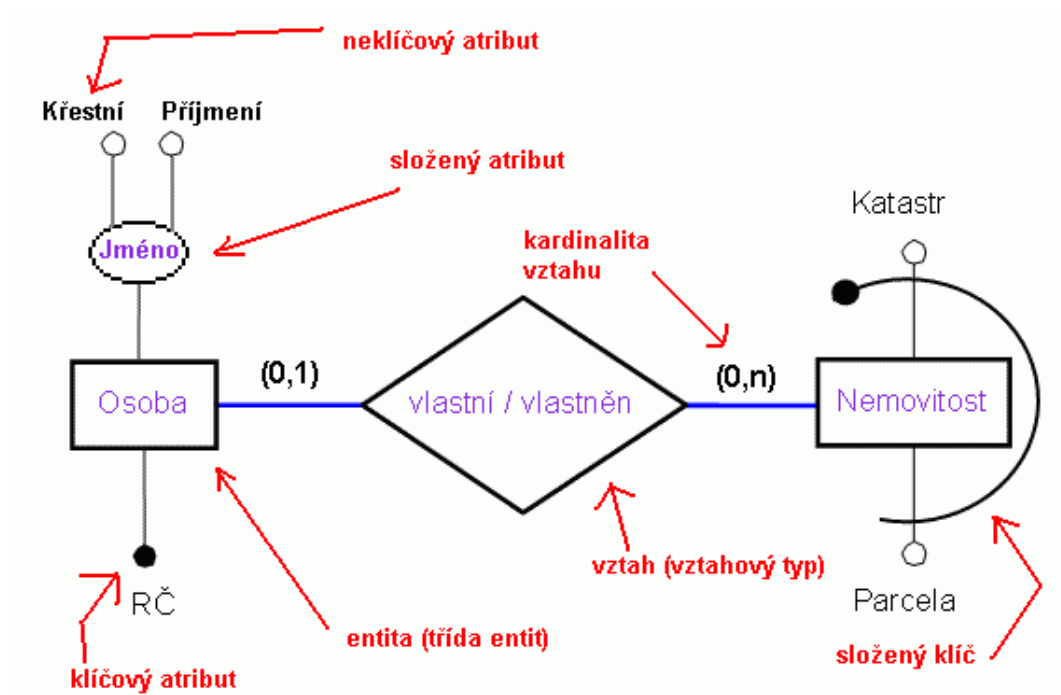
### Příklady

Pár příkladů vztahů mezi třídami entit (z Wikipedie):

- Each PERSON may be the vendor in one or more ORDERS.
- Each ORDER must be from one and only one PERSON.

- PERSON is a sub-type of PARTY. (Meaning that every instance of PERSON is also an instance of PARTY.)

De-facto standardem pro konceptuální datové modelování jsou *ER-diagramy* (entity-relationship diagramy). Hodí se hlavně pro „plochá“ formátovaná data (takže třeba pro objektové nebo relační databáze, ale ne pro XML apod.). Používají dva typy „objektů“ – *entity* (třídy entit) a *vztahy*. Jde o obdobu UML z objektového programování. Příklad ER-diagramu se vztahem dvou entit je na následujícím obrázku (popisuje i další vlastnosti – atributy entit a kardinality vztahů):



(Obrázek je upravený, rozšířený a popsán příkladem ze slidů Dr. T. Skopala k Databázovým systémům)

### Logické schéma

*Logické schéma* je datový model organizace nějakého specifického celku pomocí jednoho z databázových modelů – podle databázových modelů popsaných v předchozí sekci, tj. např. pomocí relačních tabulek, objektových tříd nebo XML. Svojí úrovní abstrakce se nachází mezi konceptuálním a fyzickým schématem.

TODO: to zřejmě nebyl dobrý nápad nacpat datové modely do architektury DB – vhodnější by to bylo přesunout, je to ale nutné trochu učesat aby seděla terminologie.

### Fyzické schéma

*Fyzické datové modely* jsou modely, které používají databázové stroje směrem k nižším vrstvám (operačního) systému. V zásadě jde o různé způsoby fyzického uložení dat (tedy schémata organizace souborů) – sekvenční soubory, B-stromy apod.

### 3.3 Algoritmy návrhu schémat relací

#### Normální formy

##### Normalizace, anomálie

Normalizace databází je technika návrhu relačních databázových tabulek, při které se minimalizují duplicity informací - a zamezuje se tak nekonzistentnosti dat. Stupně normalizace se „popisují“ pomocí *normálních forem* - čím vyšší forma, tím vyšší striktnost...

Problémy řešené normalizací:

- *update anomaly* – př.: tabulka (člověk, adresa, skill); kdyby se nevykonala update správně, může tabulka zůstat v nekonzistentním stavu (např. by se mohly změnit jen některé adresy jednoho člověka)
- *insertion anomaly* – za jistých okolností by se některá fakta nedala zaznamenat, např. v tabulce (fakulta, datum založení, kurz) můžeme zaznamenat jen data pro fakulty, které mají kurzy...
- *deletion anomaly* – za jistých okolností by se mohlo stát, že vymazání některých faktů by způsobilo vymazání dat reprezentujících jiná fakta. V předchozí tabulce bude fakulta vymazána úplně, když se vzdá všech kurzů.

Ideálně by relační databáze měla být navržena tak, aby vylučovala možnost takových anomálií. Normalizace obvykle zahrnuje dekomponování nenormalizované tabulky na dvě nebo více tabulek takových, že po jejich spojení (join) dostaneme všechny původní informace.

Abychom mohli definovat normální formy, potřebujeme znát funkční závislosti jednotlivých atributů entit relační databáze a vědět, které atributy jsou klíčové a které ne.

##### Definice (*Funkční závislosti*)

Řekneme, že atribut  $B$  je **funkčně závislý** na atributu  $A$  (značíme  $A \rightarrow B$ ), jestliže pro každou hodnotu atributu  $A$  existuje právě jedna hodnota atributu  $B$ . Rozšířené funkční závislosti se definují pro množinu atributů (pro každou  $n$ -tici atributů z nějaké množiny existuje právě jedna hodnota závislého(závislých) atributu(atributů)).

Funkční závislosti splňují tzv. *Armstrongova pravidla*, což zahrnuje pro množiny atributů  $X, Y, Z$ :

1. triviální závislost:  $X \supseteq Y \Rightarrow X \rightarrow Y$
2. transitivitu:  $X \rightarrow Y \wedge Y \rightarrow Z \Rightarrow X \rightarrow Z$
3. kompozici:  $X \rightarrow Y \wedge X \rightarrow Z \Rightarrow X \rightarrow YZ$
4. dekompozici:  $X \rightarrow YZ \Rightarrow X \rightarrow Y \wedge X \rightarrow Z$

##### Definice (*Klíč*)

**Nadklíčem**, někdy též **superklíčem**, schématu  $A$  rozumíme každou podmnožinu množiny  $A$ , na níž  $A$  funkčně závisí. Jinak řečeno nadklíč je množina atributů, která jednoznačně určuje řádek tabulky.

**Klíč**, nebo také **potenciální klíč** (candidate key), schématu  $A$  je takový nadklíč schématu  $A$ , jehož žádná vlastní podmnožina není nadklíčem  $A$ . Čili minimální nadklíč.

Každý atribut, který je obsažen alespoň v jednom potenciálním klíči se nazývá **klíčový**, ostatní atributy jsou **neklíčové**.

**Definice** (*Normální formy*)

- *První normální forma*

- Tabulka je v první normální formě, jestliže lze do každého pole dosadit pouze jednoduchý datový typ (jsou dále nedělitelné). To zahrnuje i neexistenci více sloupců tabulky se stejným druhem obsahu:

$$\left. \begin{array}{l} (\text{manager, podřízený1, podřízený2, podřízený3}) \\ (\text{manager, podřízení-vice\_hodnot\_v\_jednom\_sloupci}) \end{array} \right\} \rightarrow (\text{manager, podřízený})$$

- *Druhá normální forma*

- Existuje klíč a všechna neklíčová pole jsou funkcí celého klíče (a tedy ne jen jeho částí).

$$(\text{custID, name, address, city, state, zip}) \rightarrow \begin{array}{l} (\text{custID, name, address, zip}) \\ + (\text{zip, city, state}) \end{array}$$

- *Třetí normální forma*

- Tabulka je ve třetí normální formě, jestliže každý neklíčový atribut není transitivně závislý na žádném klíči schématu (resp. každý neklíčový atribut je přímo závislý na klíči schématu) neboli je-li ve druhé normální formě a zároveň neexistuje jediná závislost neklíčových sloupců tabulky.

$$(\text{deptID, deptName, managerID, hireDate}) \rightarrow (\text{deptID, deptName, managerID})$$

Atribut „hireDate“ je sice funkčně závislý na klíči deptID, ale jen proto, že hireDate závisí na managerID, které závisí na deptID.

- *Boyce-Coddova normální forma*

- Pro každou netriviální závislost  $X \rightarrow Y$  platí, že  $X$  obsahuje klíč schématu  $R$  ( $X$  je nadklíč).

## Algoritmy návrhu schémat relací

Schémata relací by měla být navrhována tak, aby odpovídala předem připravenému konceptuálnímu modelu (např. pomocí ER diagramů) a zároveň pokud možno splňovala co nejpřísnější požadavky na normální formy. Pro modelování relační databáze existují dva přístupy:

1. Získání množiny relačních schémat (ručně nebo převodem z např. ER diagramu) a provádění normalizace pro každou tabulku zvlášť

2. Návrh tzv. univerzálního schématu databáze – jedna velká tabulka pro celou databázi (vč. platných funkčních závislostí) a normalizace prováděná globálně

První možnost je relativně intuitivní (s ER diagramy) a jednoduchá, ale hrozí riziko přílišného rozdrobení databáze na velký počet malých tabulek (a nadbytečný i vzhledem k požadované normální formě). V druhém způsobu jsou entity jednotlivých relací „vypozorovány“ jako efekt funkčních závislostí, což není příliš průhledné a jednoduše proveditelné, ale minimalizuje to šanci na rozdrobení databáze. Oba přístupy lze také zkombinovat – převést ER model databáze do schémat a některá (nebo až všechna) potom před normalizací sloučit.

### Normalizace

Jediným způsobem, jak u nějakého obecného relačního schématu dosáhnout normální formy (obecně se požaduje většinou 3NF nebo BCNF), je rozdělení na několik podschemat. Dá se to provést ručně nebo algoritmicky a existuje více přístupů podle požadavku na normální formu, *bezeztrátovost* (dekompozice relace  $R(A, F)$  do  $R_1(A_1, F_1)$  a  $R_2(A_2, F_2)$  je bezztrátová, když  $A_1 \cap A_2 \rightarrow A_1$  nebo  $A_1 \cap A_2 \rightarrow A_2$ , tedy opětovným spojením do původní relace nevzniknou další řádky) nebo *pokrytí závislostí* (dekompozice  $R(A, F)$  do  $R_1(A_1, F_1)$  zachovává pokrytí závislostí, když  $F^+ = F_1^+ \cup F_2^+$  – nesmí se ztratit závislost ani v rámci dílčího schématu, ani jdoucí napříč schématy).

### Algoritmus (Dekompozice)

Dekompozice je algoritmus, který relační schéma převede do Boyce-Coddovy normální formy. Zaručuje zachování bezztrátovosti, ale už ne pokrytí závislostí (bez ohledu na algoritmus toto u BCNF někdy není možné). Jeho běh vypadá následovně:

1. Vyber nějaké schéma, které není v BCNF.
2. Vezmi pro něj neklíčovou závislost  $X \rightarrow Y$  (tak že  $X$  není klíč) a dekomponuj podle ní – vyhoď ze schématu  $Y$  a dej  $XY$  do zvláštní tabulky.
3. Opakuj od kroku 1, dokud existuje schéma, které není v BCNF.

### Algoritmus (Syntéza)

Algoritmus syntézy obecně dosahuje třetí normální formy a zachovává pokrytí závislostí (ale ne bezztrátovost). Pro relační schéma  $R$  s množinou funkčních závislostí  $F$  vypadá následovně:

1. Udělej minimální pokrytí  $F$  (vzhledem k tranzitivitě), nazvi ho  $G$ .
2. Sluč funkční závislosti z  $G$  se stejnou levou stranou a z každé vytvoř jedno schéma.
3. Zahod' schémata, která jsou podmnožiny jiných.

Nakonec je možné sloučit schémata s funkčně ekviv. klíči ( $K1 \leftrightarrow K2$ ), ale může to porušit normální formu, které bylo dosaženo! Pro zachování bezztrátovosti lze do přidat nějaké schéma, obsahující univerzální klíč celého původního (neděleného) schématu.

## Poznámka

Pro nalezení minimálního pokrytí atributů se používá pomocný algoritmus, který se chová takto:

1. Dekomponuj všechny funkční závislosti na elementární (na pravé straně je jen jeden sloupec)
2. Odstraň z nich redundantní atributy (takové z levé strany, které funkčně závisí na jiných z levé strany)
3. Odstraň redundantní funkční závislosti (tj. takové, které jsou tranzitivním důsledkem jiných – pravá strana funkčně závisí na levé, i když z množiny funkčních závislostí onu redundantní odstraním)

Pro druhý i třetí krok je potřeba získat *atributový uzávěr* (množina všech atributů i tranzitivně závislých na levé straně) – to se opakovaně zkouší, jestli díky funkčním závislostem nedostanu z atributů původní množiny nějaké další atributy (dokud nacházím další, přidávám je do množiny a opakuji).

## Referenční integrita

- pomáhá udržovat vztahy v relačně propojených databázových tabulkách, zabráňuje vzniku nekonzistentních dat
- kontrola přípustných hodnot
- kontrola existence položky s daným klíčem v druhé tabulce (podle cizího klíče)

Chování při porušení integrity:

- ON UPDATE, ON DELETE - podmínka spuštění akce
- ON ... RESTRICT - defaultní řešení (hlášení chyby)
- CASCADE - kaskádová aktualizace/smazání (smaže příslušné řádky v odkazované tabulce)
- SET NULL - nastavení odkazovaných řádků závislé tabulky na NULL
- SET DEFAULT - nastavení pevně určené hodnoty
- NO ACTION

## 3.4 Transakční zpracování, vlastnosti transakcí, uzamykací protokoly, zablokování

### Definice (*Transakce*)

*Transakce* je jistá posloupnost nebo specifikace posloupnosti akcí práce s databází, jako jsou čtení, zápis nebo výpočet, se kterou se zachází jako s jedním celkem.

Hlavním smyslem používání transakcí, tj. *transakčního zpracování*, je udržení databáze v konzistentním stavu. Jestliže na sobě některé operace závisí, sdružíme je do jedné transakce a tím zabezpečíme, že budou vykonány buď všechny, nebo žádná. Databáze tak před i po vykonání transakce bude v konzistentním stavu. Aby se uživateli transakce jevila jako jedna atomická operace, je nutné zavést příkazy

COMMIT a ROLLBACK. První z nich signalizuje databázi úspěšnost provedení transakce, tj. veškeré změny v databázi se stanou trvalými a jsou zviditelněny pro ostatní transakce, druhý příkaz signalizuje opak, tj. databáze musí být uvedena do původního stavu.

Tyto příkazy většinou není nutné volat explicitně, např. příkaz COMMIT je vyvolán po normálním ukončení programu realizujícího transakci. Příkaz ROLLBACK pro svou funkci vyžaduje použití tzv. *žurnálu* (logu) na nějakém stabilním paměťovém médiu. Žurnál obsahuje historii všech změn databáze v jisté časové periodě.

Jednoduchá transakce vypadá většinou takto:

1. Začátek transakce,
2. provedení několika dotazů – čtení a zápisů (žádné změny v databázi nejsou zatím vidět pro okolní svět),
3. Potvrzení (příkaz COMMIT) transakce (pokud se transakce povedla, změny v databázi se stanou viditelné).

Pokud nějaký z provedených dotazů selže, systém by měl celou transakci zrušit a vrátit databázi do stavu v jakém byla před zahájením transakce (operace ROLLBACK).

Transakční zpracování je také ochrana databáze před hardwarovými nebo softwarovými chybami, které mohou zanechat databázi po částečném zpracování transakce v nekonzistentním stavu. Pokud počítač selže uprostřed provádění některé transakce, transakční zpracování zaručí, že všechny operace z nepotvrzených („uncommitted“) transakcí budou zrušeny.

## Vlastnosti transakcí

Podívejme se nyní na vlastnosti požadované po transakcích. Obvykle se používá zkratka prvních písmen anglických názvů vlastností **ACID** – atomicity, consistency, isolation (independence), durability.

**atomicita** – transakce se tváří jako jeden celek, musí buď proběhnout celá, nebo vůbec ne.

**konzistence** – transakce transformuje databázi z jednoho konzistentního stavu do jiného konzistentního stavu.

**nezávislost** – transakce jsou nezávislé, tj. dílčí efekty transakce nejsou viditelné jiným transakcím.

**trvanlivost** – efekty úspěšně ukončené (potvrzené, „committed“) transakce jsou nevratně uloženy do databáze a nemohou být zrušeny.

Transakce mohou být v uživatelských programech prováděny paralelně (spíše zdánlivě paralelně, stejně jako je paralelismus multitaskingu na jednoprocessorových strojích jen zdánlivý, zajistí to ale možnost paralelizace „nedatabázových“ akcí a pomalé transakce nebrzdí rychlé). Je zřejmé, že posloupnost transakcí může být zpracována paralelně různým způsobem. Každá transakce se skládá z několika akcí. Stanovené pořadí provádění akcí více transakcí v čase nazveme **rozvrhem**.

Rozvrh, který splňuje následující podmínky, budeme nazývat **legální**:

- Objekt je nutné mít uzamknutý, pokud k němu chce transakce přistupovat.
- Transakce se nebude pokoušet uzamknout objekt již uzamknutý jinou transakcí (nebo musí počkat, než bude objekt odemknut).

Důležitými pojmy pro paralelní zpracování jsou sériovost či uspořádatelnost. **Sériové rozvrhy** zachovávají operace každé transakce pohromadě (a provádí se jen jedna transakce najednou). Pro  $n$  transakcí tedy existuje  $n!$  různých sériových rozvrhů. Pro získání korektního výsledku však můžeme použít i rozvrhu, kde jsou operace různých transakcí navzájem prokládány. Přirozeným požadavkem na korektnost je, aby efekt paralelního zpracování transakcí byl týž, jako kdyby transakce byly provedeny v nějakém sériovém rozvrhu. Předpokládáme-li totiž, že každá transakce je korektní program, měl by vést výsledek sériového zpracování ke konzistentnímu stavu. O systému zpracování transakcí, který zaručuje dosažení konzistentního stavu nebo stejného stavu jako sériové rozvrhy, se říká, že zaručuje **uspořádatelnost**.

Mohou se vyskytnout problémy, které uspořádatelnosti zamezují. Ty nazýváme *konflikty*. Plynou z pořadí dvojic akcí různých transakcí na stejném objektu. Existují tři typy konfliktních situací:

1. WRITE-WRITE – přepsání nepotvrzených dat
2. READ-WRITE – neopakovatelné čtení
3. WRITE-READ – čtení nepotvrzených („uncommitted“) dat

Řekneme, že rozvrh je *konfliktově uspořádatelný*, je-li konfliktově ekvivalentní nějakému sériovému rozvrhu (tedy jsou v něm stejné, tj. žádné konflikty). Test na konfliktovou uspořádatelnost se dá provést jako test acykličnosti grafu, ve kterém konfliktní situace představují hrany a transakce vrcholy. Konfliktová uspořádatelnost je slabší podmínka než uspořádatelnost – nezohledňuje ROLLBACK (*zotavitelnost* – zachování konzistence, i když kterákoliv transakce selže) a dynamickou povahu databáze (vkládání a mazání objektů). Zotavitelnosti se dá dosáhnout tak, že každá transakce  $T$  je potvrzena až poté, co jsou potvrzeny všechny ostatní transakce, které změny data čtená v  $T$ . Pokud v zotavitelném rozvrhu dochází ke čtení změn pouze potvrzených transakcí, nemůže dojít ani k jejich *kaskádovému rušení*.

Při zpracování (i uspořádatelného) rozvrhu může dojít k situaci *uváznutí* – *deadlocku*. To nastane tehdy, pokud jedna transakce  $T_1$  čeká na zámek na objekt, který má přidělený  $T_2$  a naopak. Situaci lze zobecnit i na více transakcí. Uváznutí lze buď přímo zamezit charakterem rozvrhu, nebo detekovat (hledáním cyklu v grafu čekajících transakcí, tzv. „waits-for“ grafu) a jednu z transakcí „zabít“ a spustit znovu.

K zajištění uspořádatelnosti a zotavitelnosti a zabezpečení proti kaskádovým rollbackům a deadlocku se používají různá schémata (požadavky na rozvrhy). Jedním z nich jsou uzamykací protokoly.

## Uzamykací protokoly

Vytváření rozvrhů a testování jejich uspořádatelnosti není pro praxi zřejmě ten nejvhodnější způsob. Pokud ale budeme transakce konstruovat podle určitých pravidel, tak za určitých předpokladů bude každý jejich rozvrh uspořádatelný. Soustavě takových pravidel se říká **protokol**.

Nejznámější protokoly jsou založeny na dynamickém zamykání a odemykání objektů v databázi. Zamykání (operace LOCK) je akce, kterou vyvolá transakce na objektu, aby ho chránila před přístupem ostatních transakcí.

**Definice** (*Dobře formovaná transakce*)

Transakci nazveme **dobře formovanou** pokud podporuje přirozené požadavky na transakce:

1. transakce zamyká objekt, chce-li k němu přistupovat,
2. transakce nezamyká objekt, který již je touto transakcí uzamčený,
3. transakce neodmyká objekt, který není touto transakcí zamčený,
4. po ukončení transakce jsou všechny objekty uzamčené touto transakcí odemčeny.

**Dvoufázový protokol (2PL)** – Dvoufázová transakce v první fázi zamyká vše co je potřeba a od prvního odemknutí (druhá fáze) již jen odemyká co měla zamčeno (již žádná operace LOCK). Tedy transakce musí mít všechny objekty uzamčeny předtím, než nějaký objekt odemkne. Dá se dokázat, že pokud jsou všechny transakce v dané množině transakcí dobře formované a dvoufázové, pak každý jejich legální rozvrh je uspořadatelný.

Dvoufázový protokol zajišťuje uspořadatelnost, ale ne zotavitelnost ani bezpečnost proti kaskádovému rušení transakcí nebo uváznutí.

**Striktní dvoufázový protokol (S2PL)** – Problémy 2PL jsou nezotavitelnost a kaskádové rušení transakcí. Tyto nedostatky lze odstranit pomocí striktních dvoufázových protokolů, které uvolňují zámky až po skončení transakce (COMMIT). Zřejmá nevýhoda je omezení paralelismu. 2PL navíc stále nevylučuje možnost deadlocku.

**Konzervativní dvoufázový protokol (C2PL)** – Rozdíl oproti 2PL je ten, že transakce žádá o všechny své zámky, ještě než se začne vykonávat. To sice vede občas k zbytečnému zamykání (nevíme co přesně budeme potřebovat, tak radši zamkneme víc), ale stačí to již k prevenci uváznutí (deadlocku).

### „Vylepšení“ zamykacích protokolů

**Sdílené a výlučné zámky** – Nevýhodou 2PL je, že objekt může mít uzamčený pouze jedna transakce. Abychom uzamykání provedli precizněji, je dobré vzít na vědomí rozdíl mezi operacemi READ a WRITE. *Výlučný zámek* (W\_LOCK) může být aplikován na objekty jak pro operaci READ tak pro WRITE, *sdílený zámek* (R\_LOCK) uzamyká objekt, který chceme pouze číst. Jeden objekt potom může být uzamčen sdíleným zámkem více transakcí a zvyšuje se tak možnost paralelního zpracování. Budeme-li s těmito zámky zacházet stejně jako u 2PL, opět máme zaručenou uspořadatelnost rozvrhu, ovšem nikoliv absenci uváznutí.

**Strukturované uzamykání (multiple granularity)** – Objekty jsou v tomto případě chápány hierarchicky dle relace *obsahuje*. Například databáze obsahuje soubory, které obsahují stránky a ty zase obsahují jednotlivé záznamy. Na tuto hierarchii se můžeme dívat jako na strom, ve kterém každý vrchol obsahuje své potomky. Když transakce zamyká objekt (vrchol) zamyká také všechny jeho potomky. Protokol se tak snaží minimalizovat počet zámků, tím snížit režii a zvýšit možnosti paralelního zpracování.

### Alternativní protokoly

**Časová razítka** – Další z protokolů zaručující uspořádatelnost je využití časových razítek. Na začátku dostane transakce  $T$  *časové razítko* –  $TS(T)$  (časová razítka jsou unikátní a v čase rostou), abychom věděli pořadí, ve kterém by měli být transakce vykonány. Každý objekt v databázi má *čtecí razítko* –  $RTS(O)$  (read timestamp), které je aktualizováno, když je objekt čten, a *zapisovací razítko* –  $WTS(O)$  (write timestamp), které je aktualizováno, když nějaká transakce objekt mění.

Pokud chce transakce  $T$  číst objekt  $O$  mohou nastat dva případy:

- $TS(T) < WTS(O)$ , tzn. někdo změnil objekt  $O$  potom co byla spuštěna transakce  $T$ . V tomto případě musí být transakce zrušena a spuštěna znovu (a tedy s jiným časovým razítkem).
- $TS(T) > WTS(O)$ , tzn. je bezpečné objekt číst. V tomto případě  $T$  přečte  $O$  a  $RTS(O)$  je nastaveno na  $\max\{TS(T), RTS(O)\}$ .

Pokud chce transakce  $T$  zapisovat do objektu  $O$  rozlišujeme případy tři:

- $TS(T) < RTS(O)$ , tzn. někdo četl  $O$  poté co byla spuštěna  $T$  a předpokládáme, že si pořídil lokální kopii. Nemůžeme tedy  $O$  změnit, protože by lokální kopie přestala být platná a tedy je nutné  $T$  zrušit a spustit znovu.
- $TS(T) < WTS(O)$ , tzn. někdo změnil  $O$  po startu  $T$ . V tomto případě přeskočíme write operaci a pokračujeme dále normálně.  $T$  nemusí být restartována.
- V ostatních případech  $T$  změní  $O$  a  $WTS(O)$  je nastaveno na  $TS(T)$ .

**Optimistické protokoly** – V situaci kdy se většina transakcí neovlivňuje, je režie výše uvedených protokolů zbytečně velká a můžeme použít takzvaný optimistický protokol. V protokolu můžeme rozlišit tři fáze.

1. **Fáze čtení:** Čtou se objekty z databáze do lokální paměti a jsou na nich prováděny potřebné změny.
2. **Fáze kontroly:** Po dokončení všech změn v lokální paměti je vyvolán pokus o zapsání výsledků do databáze. Algoritmus zkontroluje, zda nehrozí potenciální kolize s již potvrzenými transakcemi, nebo s některými právě probíhajícími. Pokud konflikt existuje, je třeba spustit algoritmus pro řešení kolizí, který se je snaží vyřešit. Pokud se mu to nepodaří, je využita poslední možnost a tou je zrušení a restartování transakce.
3. **Fáze zápisu:** Pokud nehrozí žádné konflikty, jsou data z lokální paměti zapsány do databáze a transakce potvrzena.

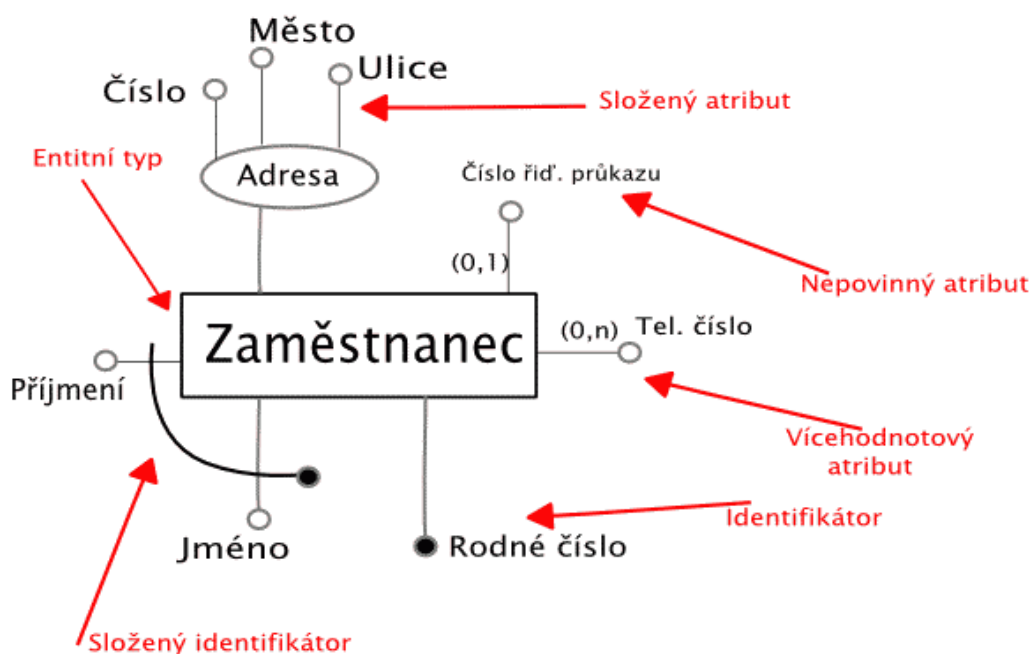
### 3.5 ER-diagramy, metody návrhu IS

ER-diagramy jsou de-facto standard pro konceptuální datové modelování. Jsou vhodné hlavně pro „plochá“ neformátovaná data, tj. hlavně pro relační, objektově-relační nebo objektové databáze. Nejsou vhodné pro multimediální nebo hierarchická data (jako např. XML). E-R v názvu znamená *entity-relationship* modelování, tedy modelování s pomocí (tříd) entit a jejich vztahů. ER model databáze definuje její konceptuální schéma. Jde vlastně o obdobu UML schémat v objektovém programování.

#### Entitní typ

*Entitní typ* (v diagramu se značí hranatým rámečkem) reprezentuje nějakou třídu entit (např. „Zaměstnanec“). Každý entitní typ má nějaké *atributy* (např. „jméno“), z nichž některé mohou být *identifikátory*, tj. takové, které jednoznačně určují instanci entity. Pokud nemá žádné identifikátory explicitně označené, jsou jimi všechny atributy dohromady (tzv. složený identifikátor). Identifikátory mohou být i víceatributové.

Atributy entitních typů mohou být *jednoduché* nebo *složené*, *povinné* či *nepovinné*, případně *jednohodnotové* a *vícehodnotové*. Jejich zobrazení ukazuje následující obrázek:

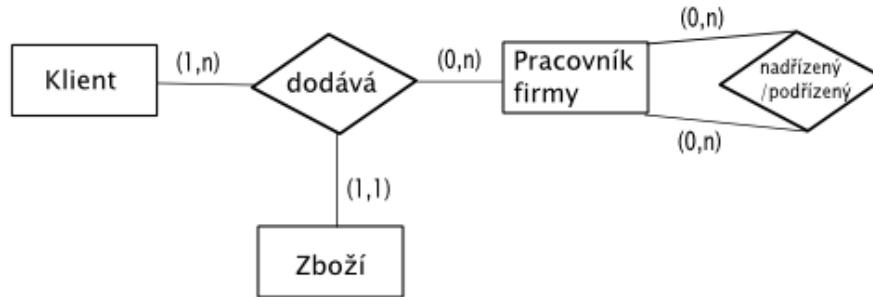


(Entitní typ se všemožnými druhy atributů)

#### Vztahový typ

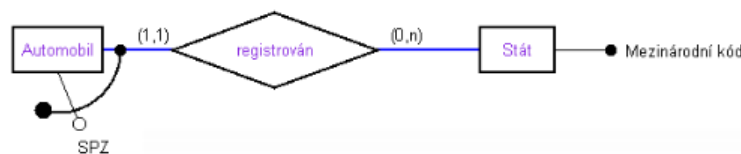
*Vztahový typ* (v diagramu značený kosočtvercem) popisuje vztahy mezi jednotlivými entitami – s těmi entitami, se kterými je v nějakém vztahu, je spojen čarou. Vztah může mít danou i *kardinalitu* (kolik entit z každé strany do vztahu vstupuje), která může být typu 1 : 1, 1 : n, m : n a je značená vedle čáry spojující vztahový typ s entitou. Entity ve vztahu mohou mít navíc *povinné* či *nepovinné členství* (vstupovat do něj vždy nebo jen někdy).

Vztahy mohou být buď binární nebo obecně  $n$ -ární, ale více než ternární vztahy se většinou neobjevují. Vztahy mohou být i rekurzivní, tj. do vztahů vstupují entity stejného typu. Instance vztahového typu je jednoznačně určena identifikátory instancí entit ve vztahu. Některé entitní typy mohou být spolidentifikovány (nebo přímo identifikovány) vztahem – pak se nazývají *slabé entitní typy*.



(Vztahové typy)

Obrázek ukazuje ternární vztah s různými kardinalitami – klientovi někdo dodává zboží jednou až  $n$ -krát, pracovník dodává nula až  $n$ -krát zboží (tj. jde o nepovinné členství ve vztahu, můžou existovat pracovníci, kteří nic nedodávají) a zboží je vždy někomu dodáváno právě jednou. Na zaměstnancích je zároveň ukázán rekurzivní binární vztah.

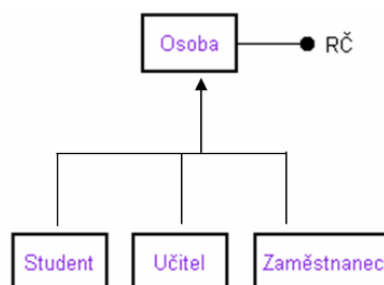


(Slabý entitní typ. Zdroj: slidy Dr. T. Skopala k Databázovým systémům)

Tento obrázek ukazuje, jak vypadá slabý entitní typ – automobil je identifikován svojí SPZ a zároveň státem, ve kterém je registrován.

### ISA hierarchie

ISA hierarchie je rozšíření ER diagramů o „dědičnost“ entit – tj. rozdělení entitních typů na subtypy (a přidání dalších vztahů nebo atributů pro subtypy). V ISA hierarchii se povoluje pouze jednonásobná dědičnost, navíc potomci nějakého entitního typu musí být jednoznačně identifikováni předkem (tj. všechny entity v hierarchii sdílí identifikátor).



(ISA hierarchie. Zdroj: slidy Dr. T. Skopala k Databázovým systémům)

### Úpravy ER diagramů

V ER diagramu je možné provádět víceméně „ekvivalentní“ úpravy (výsledný diagram reprezentuje stejný koncept databáze), např. pro odstranění vztahů s kardinalitou  $m : n$  (převod na dva vztahy kardinalitami  $1 : n$  a *průnikový entitní typ*, který je vztahy určený, takže je slabý). Dalším důvodem úprav může být zbavení se ISA hierarchie. To se dá provést více způsoby, přičemž žádný z nich nefunguje úplně obecně:

- agregace atributů a vztahů potomka do předka a úprava kardinalit (převod na nepovinné atributy a nepovinné členství ve vztahu)
- odstranění předka a duplikace všech jeho atributů a vztahů v potomcích
- nahrazení ISA vztahu klasickým vztahem (z potomků vzniknou slabé entitní typy)

Jiná úprava je odstranění vícehodnotového atributu – převede se na vztah s kardinalitou  $1 : n$  a slabý entitní typ.

### Korektní ER schéma

V *korektním* ER schématu všechny entity a vztahy splňují:

- Žádný entitní typ nemá více než jednoho ISA předka.
- ISA vztahy netvoří orientovaný cyklus.
- Identifikační vztahy netvoří orientovaný cyklus.
- Potomek v ISA hierarchii není identifikačně závislý na žádném entitním typu (je již identifikován předkem).
- Jména entitních a vztahových typů jsou jednoznačná.

TODO: Co je ksakru „metody návrhů IS“?

## 3.6 SQL

Zdroje: slidy z přednášek Databázové systémy a Databázové aplikace Dr. T. Skopala a Dr. M. Kopeckého.

### Standardy SQL

SQL (*Structured query language*) je standardní jazyk pro přístup k relačním databázím (a dotazování nad nimi). Je zároveň jazykem pro definici dat (definition data language), vytváření a modifikace schémat (tabulek), manipulaci s daty (data manipulation language), vkládání, aktualizace, mazání dat, řízení transakcí, definici integritních omezení aj. Jeho syntaxe odráží snahu o co nejpřirozenější formulace požadavků – je podobná anglickým „větám“.

SQL je standard podle norem ANSI/ISO a existuje v několika (zpětně kompatibilních) verzích (označovaných podle roku uvedení):

**SQL 86** – první „náštel“, průnik implementací SQL firmy IBM

**SQL 89** -- malá revize motivovaná komerční sférou, mnoho detailů ponecháno implementaci

**SQL 92** — mnohem silnější a obsáhlejší jazyk. Zahrnuje už

- modifikace schémat, tabulky s metadaty,
- vnější spojení, množinové operace
- kaskádové mazání/aktualizace podle cizích klíčů, transakce
- kurzory, výjimky

Standard existuje ve čtyřech verzích: Entry, Transitional, Intermediate a Full.

**SQL 1999** -- přináší mnoho nových vlastností, např.

- objektově-relační rozšíření
- nové datové typy – reference, pole, full-text
- podpora pro externí datové soubory, multimédia
- trigger, role, programovací jazyk, regulární výrazy, rekurzivní dotazy ...

**SQL 2003** -- další rozšíření, např. XML management

Komerční systémy implementují SQL podle různých norem, někdy jenom SQL-92 Entry, dnes nejčastěji SQL-99, ale nikdy úplně striktně. Některé věci chybí a naopak mají všechny spoustu nepřenositelných rozšíření – např. specifická rozšíření pro procedurální, transakční a další funkcionalitu (T-SQL (Microsoft SQL Server), PL-SQL (Oracle) ). S novými verzemi se kompatibilita zlepšuje, často je možné používat obojí syntax. Přenos aplikace za běhu na jinou platformu je ale stále velice náročný – a to tím náročnější, čím víc věcí mimo SQL-92 Entry obsahuje. Pro otestování, zda je špatně syntax SQL, nebo zda jen daná databázová platforma nepodporuje některý prvek, slouží SQL validátory (které testují SQL podle norem).

## Dotazy v SQL

Hlavním nástrojem dotazů v SQL je příkaz **SELECT**. Sdílí prvky relačního kalkulu i relační algebry – obsahuje práci se sloupci, kvantifikátory a agregační funkce z relačního kalkulu a další operace – projekce, selekce, spojení, množinové operace – z relační algebry. Na rozdíl od striktní formulace relačního modelu databáze povoluje duplikátní řádky a NULLové hodnoty atributů.

Netříděný dotaz v SQL sestává z:

- příkazu(ů) **SELECT** (hlavní logika dotazování), to obsahuje vždy
- může obsahovat i množinové operace nad výsledky příkazů **SELECT – UNION, INTERSECTION ...**

Výsledky nemají definované uspořádání (resp. jejich pořadí je určeno implementací vyhodnocení dotazu).

Příkaz **SELECT** vypadá následovně (tato verze už zahrnuje i třídění výsledků):

```

SELECT [DISTINCT]
  výraz1 [[AS] c_alias1] [, ...]
FROM
  zdroj1 [[AS] t_alias1] [, ...]
[WHERE podmínka_ř]
[GROUP BY výraz_g1 [, ...]
[HAVING podmínka_s]]
[ORDER BY výraz_o1 [, ...] ASC/DESC]

```

Kde

- výrazy mohou být sloupce, sloupce s agregačními funkcemi, výsledky dalších funkcí ...
  - výraz = <název sloupce>, <konstanta>,  
(DISTINCT) COUNT( <název sloupce> ),  
[DISTINCT] [ SUM | AVG ]( <výraz> ),  
[ MIN | MAX ]( <výraz> )
  - a navíc lze použít operátory +, -, \*, /.
- zdroje jsou tabulky nebo vnořené selecty
- výrazy i zdroje být přejmenovány pomocí AS, např. pro odkazování uvnitř dotazu nebo jména na výstupu (od SQL-92)
- podmínka je logická podmínka (spojovaná logickými spojkami AND, OR) na hodnoty dat ve zdrojích:
  - podmínka = <výraz> BETWEEN <x> AND <y>, <výraz> LIKE "%\_ ... ",  
<výraz> IS [NOT] NULL,  
<výraz> > = <> <= < > [ <výraz>/ ALL / ANY <dotaz> ],  
<výraz> NOT IN [<seznam hodnot> / <dotaz> ], EXIST ( <dotaz> )
- GROUP BY znamená agregaci podle unikátních hodnot jmenovaných sloupců (v ostatních sloupcích vznikají množiny hodnot, které se spolu s oněmi unikátními vyskytují na stejných řádkách)
- HAVING označuje podmínku na agregaci
- ORDER BY definuje, podle hodnot ve kterých sloupcích nebo podle kterých jiných výrazů nad nimi provedených se má výsledek setřídít (ASC požaduje vzestupné setřídění, DESC sestupné).

SQL nemá příkaz na omezení rozsahu na některé řádky (jako např. „potřebuji jen 50.-100. řádek výpisu“), a to lze řešit buď složitě standardně (počítání kolik hodnot je menších než vybraná, navíc náročné na hardware) nebo pomocí některého nepřenositelného rozšíření.

Pořadí vyhodnocování jednoho příkazu SELECT (nebereme v úvahu optimalizace):

1. Nejprve se zkombinují data ze všech zdrojů (tabulek, pohledů, poddotazů). Pokud jsou odděleny čárkami, provede se kartézský součin (to samé co CROSS JOIN), v SQL-92 a vyšším i složitější spojení – JOIN ON (vnitřní spojení podle podmínky), NATURAL JOIN („přirozené“ spojení podle stejných hodnot stejné pojmenovaných sloupců), OUTER JOIN („vnější“ spojení, do kterého jsou zahrnuty i záznamy, pro které v jednom ze zdrojů není nalezeno nic, co by odpovídalo podmínce, doplněné NULLovými hodnotami) atd.

2. Vyřadí se vzniklé řádky, které nevyhovují podmínce (**WHERE**)
3. Zbylé řádky se seskupí do skupin se stejnými hodnotami uvedených výrazů (**GROUP BY**), každá skupina obsahuje atomické sloupce s hodnotami uvedených výrazů a množinové sloupce se skupinami ostatních hodnot sloupců.
4. Vyřadí se skupiny, nevyhovující podmínce (**HAVING**)
5. Výsledky se setřídí podle požadavků
6. Vygeneruje se výstup s požadovanými hodnotami
7. V případě **DISTINCT** se vyřadí duplicitní řádky

### Poznámka

- Klauzule **GROUP BY** setřídí před vytvořením skupin všechny řádky dle výrazů v klauzuli. Proto by se měl seskupovat co nejmenší možný počet řádek. Pokud je možné řádky odfiltrovat pomocí **WHERE**, je výsledek efektivnější, než následné odstraňování celých skupin.
- Klauzule **DISTINCT** třídí výsledné záznamy (před operací **ORDER BY**), aby našla duplicitní záznamy. Pokud to jde, je vhodné se bez ní obejít.
- Klauzule **ORDER BY** by měla být použita jen v nutných případech. Není příliš vhodné ji používat v definicích pohledů, nad kterými se dále dělají další dotazy.

### Definice a manipulace s daty, ostatní příkazy

Standard SQL podporuje několik druhů datových typů:

- textové v národní a globální (UTF) znakové sadě (několika druhů – proměnné a pevné délky): **CHARACTER(n)**, **NCHAR(n)**, **CHAR VARYING(n)**
- číselné typy – **NUMERIC(p[,s])**, **INTEGER**, **INT**, **SMALLINT**, **FLOAT(přesnost)**, **REAL**, **DOUBLE PRECISION**
- datumové typy – **DATE**, **TIME**, **TIMESTAMP**, **TIMESTAMP(přesnost.sekund)** **WITH TIMEZONE**

Databázové servery ne vždy podporují všechny uvedené typy. Nemusí je podporovat nativně, někdy si pouze „přeloží“ název typu na podobný nativně podporovaný typ.

#### Příkaz **CREATE TABLE**

Tento příkaz slouží k vytvoření nové tabulky. Je nutné definovat její název, atributy a jejich domény (datové typy); dále je možné definovat integritní omezení (klíče, cizí klíče, odkazy, podmínky). Příkaz vypadá následovně:

```
CREATE TABLE <název> <def. sloupce/i.o. tabulky, ...>
```

A uvnitř potom

```
def. sloupce = <název> <dat.typ>
  [DEFAULT NULL|<hodnota>] [<i.o.sloupce>]
dat.typ = [VARCHAR(n) | BIT(n) | INTEGER | FLOAT | DECIMAL ...]
i.o.sloupce = [CONSTRAINT <jméno>] [NOT NULL / UNIQUE / PRIMARY KEY],
  REFERENCES <tabulka>(<sloupec>) <akce>, CHECK <podmínka>
```

```
akce = [ON UPDATE / ON DELETE]
      [CASCADE / SET NULL / SET DEFAULT / NO ACTION(hlášení chyby) ]
i.o.tabulky = UNIQUE, PRIMARY KEY <sloupec, ... >,
FOREIGN KEY <sloupec, ... >,
REFERENCES <tabulka>(<sloupec, ... >),
CHECK( <podmínka> )
```

## Příkazy pro manipulaci se schématem

- Úprava tabulky:

```
ALTER TABLE <název> ADD {COLUMN} <def.sloupec>, ADD <i.o.tabulky>,
ALTER COLUMN <sloupec> [ SET / DROP ], DROP COLUMN <sloupec>,
DROP CONSTRAINT <jméno i.o.>
```

- Smazání tabulky (není to samé jako vymazání všech dat z tabulky!):

```
DROP TABLE <tabulka>
```

- Vytvoření „pohledu“ – navenek se chová jako tabulka, ale vnitřně se při každém dotazu provede vnořený dotaz (který definicí pohledu zapisují):

```
CREATE VIEW <název "tabulky"> ( <sloupec, ... > )
AS <dotaz> {WITH [ LOCAL / CASCADED ] CHECK OPTION }
```

Některé databázové platformy umožňují do takto vytvořených pohledů i zapisovat.

## Příkazy pro manipulaci s daty

- Vložení nových dat do tabulky

```
INSERT INTO <tabulka> ( <sloupec, ... > )
[VALUES ( <výraz, ... > ) / (<dotaz>) ]
```

- Úprava dat (na řádcích které vyhovují podmínce se nastaví zadané hodnoty vybraným sloupcům):

```
UPDATE <tabulka> SET
( <sloupec> = [ NULL / <výraz> / <dotaz> ] , ... )
WHERE (<podmínka>)
```

- Smazání řádků vyhovujících podmínce z tabulky:

```
DELETE FROM <tabulka> ( WHERE <podmínka> )
```

TODO: doplnit ?

## 3.7 Indexy, triggery, uložené procedury, uživatelé

TODO: pořádná definice indexu

### Index

Index je obvykle definován výběrem tabulky a jejího konkrétního sloupce (nebo sloupců), nad kterými si designér databáze přeje dotazování urychlit; dále pak technickým určením typu. Chování a způsoby uložení indexů se mohou významně lišit podle použité databázové technologie. Výjimku mohou tvořit například full-textové indexy, které jsou v některých případech (nerelační databáze typu Lotus Notes) definovány nad celou databází, nikoliv nad konkrétní tabulkou.

### Použití indexu

Na první pohled by se mohlo zdát, že čím víc indexů, tím lepší chování databáze a že po vytvoření indexů pro všechny sloupce všech tabulkách dosáhneme maximálního zrychlení. Tento přístup naráží bohužel na dva zásadní problémy:

1. Každý index zabírá v paměti vyhrazené pro databázi nezanedbatelné množství místa (vzhledem k paměti vyhrazené pro tabulku). Při existenci mnoha indexů se může stát, že paměť zabraná pro jejich chod je skoro stejně velká, jako paměť zabraná jejími daty - zvláště u rozsáhlých tabulek (typu faktových tabulek v datovém skladu) může něco takového být nepřijatelné.
2. Každý index zpomaluje operace, které mění obsah indexovaných sloupců (například SQL příkazy UPDATE, INSERT). To je dáno tím, že databáze se v případě takové operace nad indexovaným sloupcem musí postarat nejen o změny v datech tabulky, ale i o změny v datech indexu.

### Typy indexů

Indexy mohou mít svůj typ, který blíže určuje, jakým způsobem má být přistupováno k datům tabulky optimalizováno. Označení se různí, ale nejčastěji je to:

- PRIMARY - Tento typ se v každé tabulce může vyskytovat nejvýše jednou. Definuje sloupec tabulky, který svou hodnotou jednoznačně identifikuje záznam. Ve většině případů se dodržuje konvence takový sloupec nazvat ID a jeho datový typ stanovit jako celé číslo (není-li potřeba jinak). Databázový server by měl být schopen nedopustit, aby byla do sloupce, k němuž se tento typ indexu vztahuje, byla vložena hodnota, která již v tabulce existuje (většinou takový pokus končí chybovou hláškou).
- UNIQUE - Tento typ je podobný PRIMARY co do jednoznačnosti záznamu v tabulce (jak naznačuje i jeho název) a dopadu, který to na práci s databází má; ale může se vyskytovat u více sloupců tabulky. Podle účelu, ke kterému má tabulka sloužit, se občas definují indexy složené z více sloupců - potom opět nelze vložit záznam, který by již v této kombinaci někde v tabulce existoval.
- INDEX - Definicí indexu tohoto typu je v tabulce zajištěna optimalizace vyhledávání podle sloupce, ke kterému se daný index váže. Většinou si databázový server vytvoří a nadále udržuje vnitřní seznam odkazů na řádky tabulky, seřazený podle hodnot sloupce, k němuž se váže. Udržování takto seřazené posloupnosti urychluje vyhledávání (je možno použít některé interpolační numerické

metody), řazení i jiné zásahy do tabulky, které jsou omezeny podmínkou na dotyčné záznamy.

- FULL-TEXT - Vytvořením tohoto indexu se databázový server bude snažit optimalizovat full-textové vyhledávání v daném sloupci u dané tabulky.

## Triggery

Databázový trigger je programový kód, který je automaticky vykonán jako reakce na nějakou událost v určité databázové tabulce. Triggery mohou omezit přístup k určitým datům, provádět logování, nebo kontrolovat změny dat.

Rozlišujeme dvě hlavní třídy triggerů a to *řádkový trigger* a *dotazový (statement) trigger*. Řádkový trigger můžeme definovat pro každý řádek tabulky, zatímco dotazový trigger se vykoná pouze jednou pro konkrétní databázový dotaz. Každá třída triggerů může být několika typů. Jsou *before triggers* a *after triggers*, což značí kdy má být trigger vykonán. Také se můžeme setkat s *instead of triggers*, který je potom vykonán *místo* dotazu kterým byl spuštěn.

Jaké události mohou trigger spustit se pochopitelně liší databázový systém od systému, ale existují tři typické události, které to mohou být:

1. INSERT – nový záznam je vložen do databáze,
2. UPDATE – záznam je měněn,
3. DELETE – záznam je mazán.

Kromě těchto typických událostí může databázový systém umožňovat nastavovat triggery také na mazání, či vytváření celých tabulek, či dokonce přihlášení nebo odhlášení uživatele.

Hlavní vlastnosti a efekty databázových triggerů jsou:

- nepřijímají žádné parametry nebo argumenty,
- nemohou volat operace pro řízení transakcí COMMIT a ROLLBACK,
- mají přístup k datům, které budou měněny, je tedy možné vykonávat akce na základě nich,
- nemohou vracet záznamy,
- obtížně se ladí,
- mnoho triggerů nebo složité triggery mohou práci s databází velice zpomalit a navíc znepřehlednit.

## K čemu triggery používat?

Triggery se v databázích používají z několika důvodů, které mohou souviset s konzistencí dat, jejich údržbou, nebo mohou být způsob, kterým databáze komunikuje s okolím. Podívejme se na některá typická schémata:

- **Konzistence dat** – Trigger může provést výpočet a na základě toho povolit nebo nepovolit změnu dat v databázi. Například trigger může zakázat smazání zákazníka z databáze v případě, kdy má u nás nějaký dluh a podobně.
- **Logování** – Trigger může evidovat kdo, kdy a jak měnil data. Lze tak dohledat pracovníka, který zadal špatné údaje nebo zjistit, v kolik hodin došlo k včerejší uzávěrce.

- **Verzování dat** – Díky triggerům lze snadno naprogramovat aplikaci tak, aby jedna tabulka udržovala historii změn tabulky jiné. To lze s úspěchem použít třeba jako bezpečnostní mechanismus.
- **Zasílání zpráv** – Trigger může spustit nějaký externí program nebo proces. Například může trigger autorovi poslat e-mail, pokud byl k jeho článku přidán příspěvek.

## Uložené procedury

Uložená procedura (anglicky stored procedure) je databázový objekt, který neobsahuje data, ale část programu, který se nad daty v databázi má vykonávat.

Uložená procedura je především procedura. Jedná se o část programu, který je (nebo by aspoň měl být) jasně funkčně oddělený od svého okolí, má interface (seznam parametrů) pro komunikaci s jinými moduly programu. Může mít vlastní lokální proměnné neviditelné pro ostatní části programu.

Uložená procedura je uložená (rozuměj: uložená v databázi). To znamená, že se k ní lze chovat stejně jako ke každému jinému objektu databáze (indexu, pohledu, triggeru apod.). Lze jí založit, upravovat a smazat pomocí příkazů dotazovacího jazyka databáze (v případě relační databáze obvykle pomocí příkazů DDL SQL).

Pro psaní uložených procedur je obvykle používán specifický jazyk konkrétní databáze, který je rozšířením jejího dotazovacího jazyka (hezkým příkladem je databáze Oracle s procedurálním jazykem PL/SQL, který je rozšířením klasického dotazovacího jazyka SQL).

## Proč ukládat procedury?

- **Jednotné rozhraní** – Použití uložených procedur vychází z faktu, že většina operací nad daty v databázi probíhá stejně bez ohledu na to, kdo operaci provádí. Příklad: Pokud je třeba uložit do tabulky zákazníků nového zákazníka, tak se to z pohledu databáze děje stejně pro zákazníka internetového obchodu, pro zákazníka, kterého zadává pracovnice telefonického centra přes formulář programu napsaného například v C++ , nebo pro zákazníky, kteří jsou vkládáni automaticky na základě textového reportu, který přijde každý den z „kamenných“ prodejních míst a je zpracováván pomocí programu napsaného v PowerBuilderu. Je tedy celkem dobrý důvod, aby existovala uložená procedura „Zapiš nového zákazníka“, kterou by mohly volat všechny tři výše uvedené aplikace - alternativou bez uložené procedury by bylo, že bych podobnou proceduru musel napsat ve třech verzích - jednou v C++, jednou v Power Builderu a jednou v rámci programu pro internetový obchod (třeba ASP nebo PHP).
- **Skrytí datových operací** – Druhou výhodou použití uložených procedur je, že se nemusím (v programu na „klientské“ straně) zabývat tím, jak jsou data uložena v konkrétních tabulkách. V našem případě je mi jedno, jak si databáze uvnitř pamatuje zákazníky - prostě zadám jako parametr procedury jméno, příjmení, číslo kreditky a co si zákazník koupil - a databáze (resp. její uložená procedura) si to nějak přebere. Uložené procedury se v případě databázových aplikací staly základním kamenem pro realizaci architektury kli-

ent/server, kdy je na jedné straně (klientská část) realizována v běžném procedurálním programovacím jazyku komunikace s uživatelem (formuláře nebo třeba webové stránky) a na druhé straně (serverová část) je pomocí uložených procedur realizována správa dat v relační databázi. Obě části (klientská a serverová) mezi sebou komunikují přes co nejjednodušší rozhraní - voláním uložených procedur.

TODO: uživatelé

### 3.8 Vícevrstevné architektury

TODO: předělat, tohle je jen copy & paste z Wiki a slajdů VUT Brno

#### Multitier architecture

Multi-tier architecture (often referred to as n-tier architecture) is a client-server architecture, originally designed by Jonathon Bolster of Hematites Corp, in which an application is executed by more than one distinct software agent. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of "multi-tier architecture" refers to three-tier architecture

#### Základ kooperativního zpracování

Faktory ovlivňující architekturu:

- požadavky na interoperabilitu zdrojů
- růst velikosti zdrojů
- růst počtu klientů

Typy služeb v databázové technologii:

- prezentační služby: příjem vstupu, zobrazování výsledků
- prezentační logika: řízení interakce (hierarchie menu, obrazovek)
- logika aplikace: operace realizující algoritmus aplikace
- logika dat: podpora operací s daty (integritní omezení, ...)
- datové služby: akce s databází (definice a manipulace, transakční zpracování, ...)
- služby ovládání souborů: vlastní V/V operace

#### Varianty architektury klient-server

##### • Klient-server se vzdálenými daty

Na serveru jsou jen datové služby a ovládání souborů, zbytek zajišťují klienti. Problémem jsou velké nároky na přenosovou kapacitu od klienta k serveru a HW zatížení klientských stanic

##### • Klient-server se vzdálenou prezentací

Na klientské stanici jsou jen prezentační služby a prezentační logika, zbytek je na serveru. Nevýhodou je právě zátěž na HW serveru.

- **Klient-server s rozdělenou logikou**

Část logiky aplikací i logiky dat je na serveru a část zpracovává klient. Jde o vyvážené řešení, které má ale horší rozšířitelnost.

- **Třívrstvá architektura**

Zahrnuje dva servery – aplikační a databázový, spojené rychlou linkou. Z hlediska zátěže a rozšířitelnosti nejvýhodnější.

### **Přínos architektury klient/server a třívrstvé architektury**

- pružnější rozdělení práce
- lze použít horizontální (více serverů) i vertikální (výkonnější server) škálování
- aplikace mohou běžet na levnějších zařízeních
- na klientských stanicích lze používat oblíbený prezentační software
- standardizovaný přístup umožňuje zpřístupnit další zdroje
- centralizace dat podporuje účinnější ochranu
- u třívrstvé architektury centralizace údržby aplikace, možnost využití sdílených objektů (business objects) několika aplikacemi

### **Podpora pro rozdělení zátěže v architektuře klient/server**

- deklarativní integritní omezení
- databázové trigger
- uložené podprogramy

### **Three-tier architecture**

'Three-tier' is a client-server architecture in which the user interface, functional process logic ("business rules"), data storage and data access are developed and maintained as independent modules, most often on separate platforms. The term "three-tier" or "three-layer", as well as the concept of multitier architectures, seems to have originated within Rational Software.

The three-tier model is considered to be a software architecture and a software design pattern.

Apart from the usual advantages of modular software with well defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently as requirements or technology change. For example, a change of operating system from Microsoft Windows to Unix would only affect the user interface code.

Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic may consist of one or more separate modules running on a workstation or application server, and an RDBMS on a database server or mainframe contains the data storage logic. The middle tier may be multi-tiered itself (in which case the overall architecture is called an "n-tier architecture").

The 3-Tier architecture has the following 3-tiers:

- Presentation Tier

- Application Tier/Logic Tier/Business Logic Tier
- Data Tier

### Web Development usage

In the Web development field, three-tier is often used to refer to Websites, commonly Electronic commerce websites, which are built using three tiers:

- A front end Web server serving static content
- A middle dynamic content processing and generation level Application server, for example Java EE platform.
- A back end Database, comprising both data sets and the Database management system or RDBMS software that manages and provides access to the data.

### Other Considerations

To further confuse issues, the particular data transfer method between the 3 tiers must also be considered. The data exchange may be file-based, client-server, event-based, etc. Protocols involved may include one or more of SNMP, CORBA, Java RMI, Sockets, UDP, or other proprietary combinations/permutations of the above types and others. Typically a single "middle-ware" implementation of a single protocol is chosen as the "standard" within a given system, such as J2EE (which is Java specific) or CORBA (which is language/OS neutral.) The importance of the decision of which protocol is chosen affects such issues as the ability to include legacy applications/libraries, performance, maintainability, etc. When choosing a "middle-ware protocol" (not to be confused with the "middle-of-the-three-tiers") engineers should not be swayed by "public opinion" about a protocol's modern-ness, but should consider the technical benefits and suitability to solve a problem. (for example CGI is very old and "out of date" but is still quite useful and powerful, so is shell scripting, and UDP for that matter)

Ideally the high-level system abstract design is based on business rules and not on the front-end/back-end technologies. The tiers should be populated with functionality in such a way as to minimize dependencies, and isolate functionalities in a coherent manner - knowing that everything is likely to change, and changes should be made in the fewest number of places, and be testable.

## 3.9 Vazba databází na internetové technologie

TODO: všechno

## 3.10 Organizace dat na vnější paměti, B-stromy a jejich varianty

Vnější paměť

### Definice (Vnější paměť)

*Vnější paměť* je úložiště dat (paměťové médium), u kterého je rychlost načítání dat zpravidla nízká a přístup k nim ne úplně přímý (záleží na uspořádání dat na médiu), ne-li pouze sekvenční (oproti vnitřní paměti s rychlým náhodným přístupem a menší kapacitou). Příkladem vnější paměti je pevný disk nebo magnetická páska.

*Magnetické pásky* poskytují vysokou kapacitu, ale nízkou rychlost a pouze sekvenční přístup. Pro jejich kapacitu je důležitá hustota záznamu, potřebují meziblokové mezery pro vyrovnání nepřesnosti přetáčení pásky.

*Pevné disky* umožňují přímý přístup, ale jeho rychlost není vždy stejná. Ovlivňuje ji fyzická vzdálenost dat – pevný disk má několik *válců*, na nichž jsou uloženy jednotlivé datové *stopy*. K válcům přísluší *čtecí hlavy* (je jich stejně jako válců, ale pohybují se všechny současně, takže může v 1 okamžiku číst jen jedna). Disky jsou většinou rozděleny na *sektory* – nejmenší jednotku dat, kterou je možné načíst nebo uložit (zpravidla jednotky KB). Pro rychlost přístupu k datům jsou důležité tyto veličiny (výrobce disků jsou zpravidla udávány průměrné hodnoty):

- *seek (s)* – přesun na jinou stopu, dnes zpravidla kolem 4-8 ms
- *(average) rotational delay (r)* – otočení válce –1 půlotáčka, pro nejčastější 7200rpm disk je to cca 4 ms
- *block transfer time (btt)* – doba přenesení 1 bloku po sběrnici, na ATA/100 disku se 4 KB bloky teoreticky 0.04 ms

Pokud jsou data umístěna na disku za sebou sekvenčně, rychlost jejich načtení je mnohem vyšší než při náhodném rozmístění, protože není třeba provádět přesuny mezi stopami a otáčení válců navíc.

### Příklad

*Jak vypadá načtení 1 MB dat z pevného disku?* Předpokládejme, že na 1 stopu se vejde 512 KB a 1 blok má 4 KB. Jsou-li data umístěna na disku sekvenčně, potřebuji pro načtení 1 MB dat najít první blok a potom číst dvě celé stopy (2 otáčky), tj. celkem  $s+r+(2 \cdot 2r)$  (a přenos po sběrnici lze zanedbat, protože probíhá zároveň se čtením). Pokud jsou data na disku náhodně rozprostřena, potřebuji celkem 256-krát najít blok a načíst ho:  $256 \cdot (s + r + btt)$ , takže operace trvá až 100-krát déle.

### Soubor

#### Definice (Záznam, klíč)

*(Logický) záznam* je jednotka dat (např. v databázi), má *atributy* (z nichž každý má jméno a doménu – povolenou množinu hodnot). Logickému záznamu v reprezentaci na disku odpovídá *fyzický záznam* (nějaké délky  $R$  – pevné nebo proměnné), který navíc může obsahovat ještě další data – oddělovače, ukazatele, hlavičky.

*Klíč* je množina atributů, která jednoznačně identifikuje záznam; proti tomu *vyhledávací klíč* je množina atributů, pro kterou lze nalézt množinu odpovídajících záznamů. Vyhledávací klíče jsou tří druhů: hodnotový („obyčejné“ hodnoty některých atributů), hašovaný a relativní (přímo pozice v souboru).

### Definice (Soubor)

(Homogenní) *soubor* je multimnožina záznamů. Fyzicky na vnější paměti je organizován do *bloků (stránek)* (velikosti  $B$ , typicky několika KB) – hl. jednotkou přenosu dat mezi vnitřní a vnější pamětí. Poměr velikosti záznamu k velikosti bloku ( $B/R$ ) se nazývá *blokovací faktor* ( $\lfloor b \rfloor$ ). Záznamy mohou být rozprostřeny i přes několik stránek, nebo může být pouze jeden záznam na 1 stránku; ideální (ale ne vždy dosažitelné) je, pokud beze zbytku zaplňují stránky. Na souboru jsou definovány operace se záznamy: *insert*, *delete*, *update* a *fetch*.

### Definice (Dotaz)

*Dotaz* je každá funkce, která každému svému argumentu přiřadí odpovídající množinu záznamů ze souboru („totální vyčíslitelná funkce definovaná na souboru“). Dotazy mohou být těchto typů:

- Načtení všech záznamů (`SELECT * FROM tabulka`)
- Na úplnou shodu (`SELECT * FROM tabulka WHERE sloupec1 = 'hodnota' AND sloupec2 = 'hodnota'` pro tabulku se 2 sloupci – dány jsou všechny atributy)
- Na částečnou shodu (`SELECT * FROM tabulka WHERE sloupec1 = 'hodnota'` pro tabulku se 2 sloupci – zadaná je jen část atributů)
- Na intervalovou shodu (částečnou nebo úplnou) (`SELECT * FROM tabulka WHERE sloupec1 > 'hodnota'`)

U souborů se sleduje rychlost provedení těchto operací.

## Statické metody organizace souboru

### Definice (Schéma organizace souboru)

*Schéma organizace souboru* je popis logické paměťové struktury, do níž lze zobrazit logický soubor, spolu s algoritmy operací nad touto strukturou. Ta je obvykle tvořena z logických stránek (bloků pevné délky) a může popisovat více provázaných log. souborů, z nichž *primární soubor* je ten, který obsahuje uživatelská data. Operace definované nad schématem org. souboru jsou kromě operací nad soubory ještě *build*, *reorganization*, *open* a *close*.

Proti němu stojí *fyzické schéma souboru* – struktura nad fyzickými soubory, nejbližší hardwaru je *implementační schéma souboru*.

Zajištění *Vyváženosti struktury* znamená zajištění omezení cesty při vyhledávání nějakým výrazem (zaručení asymptotické složitosti), navíc zaručení rovnoměrnosti zaplnění struktury – *faktor naplnění stránek*. Schémata, která splňují obě podmínky, se nazývají *dynamická*, ostatní jsou označována jako *statická*.

### Poznámka (Časové odhady)

Pro schémata organizace souborů se počítají časové odhady provedení jednotlivých operací – jednodušších, jako je přístup k záznamu ( $T_F$ ), *rewrite* – přepis během 1 otáčky disku ( $T_{RW}$ ), příp. sekvenční čtení; dále i složitějších jako vyhledání záznamu, přidání, smazání a úprava záznamu, reorganizace struktury nebo načtení celého souboru.

### Hromada (neuspořádaný sekvenční soubor)

*Hromada(heap)* je naprosto nejjednodušší schéma organizace souboru, kdy jsou záznamy v souboru jen náhodně seřazeny za sebou. Časová složitost vyhledávání je  $O(n)$ , pokud  $n$  je počet záznamů. Jde o *nehomogenní soubor*, kde záznamy obvykle nemají pevnou délku.

### Uspořádaný sekvenční soubor

V *uspořádaném sekvenčním souboru* jsou záznamy řazeny podle klíče. Aktualizované záznamy se umístí do zvláštního souboru a až při další operaci „reorganization“ jsou přidány do primárního. Složitost nalezení záznamu je také  $O(n)$ , ale pokud se hledá podle klíče, podle kterého jsou záznamy seřazeny, a navíc je soubor na médiu s přímým přístupem, sníží se na  $O(\log n)$ .

### Index-sekvenční soubor

Toto schéma uvažuje primární soubor jako sekvenční, uspořádaný podle primárního klíče. Nad ním je pak vytvořen (třeba i víceúrovňový) *index*. Ten sestává ze seznamu čísel stránek a minimálních hodnot klíče jim odpovídajících záznamů. Pokud má index víc úrovní, provádí se pro vyšší úroveň to samé na blocích indexů úrovně o 1 nižší. Nejvyšší úroveň indexu se obvykle vejde do 1 bloku, tzv. *master*.

Počet potřebných úrovní pro  $n$  záznamů se dá spočítat jako  $\lceil \log_p \frac{n}{|b|} \rceil$ , kde  $p = \lfloor \frac{B}{V+P} \rfloor$  při velikosti klíče  $V$  a pointeru na stránku  $P$ . Problémem je přidávání nových záznamů, kdy se tyto řetězí za sebe v tzv. *oblasti přetečení* (každý z nich má pointer na další záznam v oblasti přetečení). Pro oddálení nutnosti vkládání do oblasti přetečení lze iniciálně bloky plnit na méně než 100%.

### Indexovaný soubor

*Indexovaný soubor* znamená primární soubor plus indexy pro různé vyhledávací klíče. Neindexují se už stránky, ale přímo záznamy, a proto primární soubor nemusí být nutně setříděný. Index může být podobný jako u index-sekvenčního souboru, pro záznamy se stejným klíčem je ale vhodné, aby byly na všech úrovních indexu kromě poslední sloučené. Při aktualizaci se nepoužívá oblast přetečení, mění se pouze index.

Existuje i několik dalších variant indexů. Pro zmenšení náročnosti dotazů na kombinovanou částečnou shodu se používá *kombinovaný index* pro více atributů, u něhož je ale nutné předem zjistit na které kombinace atributů budou často pokládány dotazy, a pro takové kombinace tento index teprve vytvořit. *Clusterovaný index* zaručuje, že záznamy s podobnou hodnotou indexovaného atributu jsou blízko sebe v primárním souboru – např. pokud je primární soubor podle tohoto atributu setříděný. Tento index lze použít jen pro 1 atribut.

*Bitové mapy* se dají použít jako index pro atributy s malou doménou (množinou možných hodnot) – pro každou hodnotu této domény se vyrobí vektor bitů stejné délky, jako je počet záznamů v primárním souboru, kde jednička na  $i$ -té pozici indikuje, že  $i$ -tý záznam má právě tuto hodnotu atributu. To umožňuje jednoduché provádění booleovských dotazů na tento atribut. Vektory bitů navíc lze komprimovat, takže nezabírají tolik místa.

## Soubor s přímým přístupem

V tomto schématu jsou záznamy v primárním souboru („adresovém prostoru“ velikosti  $M$ ) rozptýleny pomocí *hashovací funkce*. Často se používá funkce  $h = k \bmod M'$ , kde  $M'$  je nejbližší prvočíslo menší než velikost adresového prostoru. Hashovací funkcí se určuje buď jenom číslo stránky, nebo i relativní pozice v ní. Při hashování vznikají kolize, které se dají řešit *otevřeným adresováním* (řetězením kolizních záznamů za sebe), *rehashováním* (další funkcí) nebo použitím *oblasti přetečení*. Snaha je většinou umístit kolizní záznamy do stejné stránky.

Pokud je hashovací funkce prostá, jedná se o *perfektní hashování*. Toho ale v praxi vlastně nelze dosáhnout, takže se tento výraz používá i pro označení stavu, kdy je pro nalezení záznamu potřeba nejvýš  $O(1)$  přístupů k médiu. Očekávaná délka řetězce kolizí při počtu  $N$  záznamů v prostoru velikosti  $M$  je  $1/(1 - \frac{N}{M})$ .

## Třídění na vnější paměti

### Algoritmus (*Třídění sléváním (Mergesort)*)

Tento algoritmus se používá pro třídění dat, která se nevejdou do vnitřní paměti. Dá se použít i při sekvenčním přístupu k datovým souborům. Nejjednodušší verze bez bufferů vypadá takto:

- inicializace: na začátku každého kroku data rozdělí do 2 souborů
- načte 2 záznamy, každý z jednoho souboru a porovná je
- ve správném pořadí je zapíše do výstupního souboru, ze vstupního souboru si načte další dva
- v prvním kroku získám uspořádané posloupnosti délky 2; v dalších krocích vždy porovnáím načtené prvky, zapíšu menší z nich a ze souboru odkud tento pocházel si načtu další, takže získám vždy uspořádané posloupnosti dvojnásobné délky než v předchozím kroku
- po  $\lceil \log n \rceil$  krocích je soubor s  $n$  záznamy setříděný.

Vylepšení se dosáhne např. přímo střídavým zapisováním výstupu do 2 souborů, kdy se zbavím nutnosti na začátku každého kroku data dělit, nebo použitím více souborů najednou. Je taky možné využít rostoucích posloupností prvků, které se v souboru nacházejí již před započítáním třídění.

### Algoritmus (*Třídění haldou*)

Pro třídění ve vnitřní paměti se používá algoritmus *třídění haldou (heapsort)*, který se dá zakomponovat do vylepšení třídění sléváním (viz níže). Jeho základem je datová struktura *halda* (konkrétně maximální halda, max-heap), reprezentovaná jako pole záznamů, na kterém je binární stromová struktura: záznam  $k$  má vždy vyšší klíč než jeho dva synové, nacházející se na pozicích  $2k+1$  a  $2k+2$  při číslování od 0 (pokud tato pozice není větší než velikost haldy, v opačném případě záznam nemá syny). Na pozici 0 se tak nachází záznam s nejvyšším klíčem. Postup třídění je následovný:

- největší prvek (z pozice 0) se prohodí s tím prvkem, jehož číslo pozice odpovídá aktuální velikosti haldy

- velikost haldy se zmenší o 1
- dokud neplatí podmínka, že klíč prvku získaného z konce haldy je větší než oba klíče jeho synů, prohazuje se tento se synem s větším klíčem (a tak posouvá v haldě dál)
- toto se opakuje, dokud je velikost haldy větší než 1, odzadu tak v poli vzniká setříděná posloupnost

Časová složitost algoritmu je  $O(n \cdot \log n)$  pro pole záznamů velikosti  $n$ .

### Algoritmus (*n*-cestné třídění)

Pokud mám k dispozici ve vnitřní paměti  $n + 1$  stránek, mohu postupovat následovně:

- v 1. kroku načíst do paměti  $n$  stránek
- ty setřídít pomocí heapsortu (nebo i quicksortu apod.) a získat tak delší setříděné úseky (*běhy*)
- slévat vždy  $n$  nejkratších běhů (pomocí mergesortu) a vytvářet tak jeden běh
- toto opakovat, dokud existuje více než 1 běh.

Čas. složitost pro  $M$  stránek v souboru je  $O(2M \lceil \log_n M/n \rceil)$ .

### Algoritmus (*Dvojitá halda*)

Delší běhy při slévání se dají vytvářet pomocí dvojitě haldy – v paměti mám dvě haldy z celkem  $n$  prvků, opakovaně z první haldy odebírám a zapisuji minimální prvek do výstupního běhu a načítám další prvek, pokud ten je větší než minimum haldy, vloží ho do první haldy, pokud je menší, vloží ho do druhé haldy, která vzniká od konce mého pole. Až se první halda vyčerpá, použiji druhou a začnu nový běh. Toto v nejhorším případě dává stejnou velikost běhů jako obyčejná halda, průměrně je 2x lepší.

## B-stromy

### Definice (*B-strom*)

B-strom řádu  $m$  je výškově vyvážený strom, který má násl. vlastnosti:

1. Kořen má minimálně 2 syny, pokud není sám listem.
2. Každý jiný uzel kromě listů má nejméně  $\lceil \frac{m}{2} \rceil$  a nejvíce  $m$  synů a vždy o 1 méně dat. záznamů (listy mají jen datové záznamy).
3. Klíče všech záznamů v  $i$ -tém podstromu uzlu  $A$  jsou větší než klíč  $i$ -tého záznamu uzlu  $A$  a menší nebo rovny klíči  $i + 1$ -tého záznamu.
4. všechny *větve* (cesty od kořene k listu) jsou stejně dlouhé.

Variantou jsou *redundantní B-stromy*, kdy všechna data jsou umístěna v listech, vnitřní uzly obsahují pouze vyhledávací klíče. Jiná možnost je použití pouze klíče a odkazu na celý záznam, místo vkládání kompletních záznamů do stromu.

### Algoritmus (*Operace na B-stromě*)

*Vyhledávání* v B-stromech podle klíče se provádí jednoduchým průchodem do hloubky.

*Vkládání* se děje nalezením místa pro vložení. Pokud ještě daný rodič neobsahuje  $m$  synů, jednoduše nového syna vložím. Pokud ano, je nutné rozštěpit rodičovský uzel na dva (do každého dám polovinu synů). Toto se může opakovat směrem k vyšším vrstvám, případně až do kořene.

*Odebírání* prvků je opačný postup, v případě podtečení uzlu (zůstane v něm méně než  $\lceil \frac{m}{2} \rceil$  synů) musím přebírat data od sousedních uzlů nebo slévat. V redundantních B-stromech není nutné při mazání odstraňovat vyhledávací klíč z vnitřních uzlů – prvek s touto hodnotou se ve stromě už nebude nacházet, ale vyhledávat podle jeho klíče je dál možné.

Lepší naplněnosti uzlů za cenu snížení rychlosti se dá dosáhnout pomocí *vyvažování stránek* – při přetečení stránky nejdříve kontroluji, jestli nejsou volné sousední; pokud ano, přerozdělím data a upravím klíče. Podobně je možné postupovat při mazání (i pokud není třeba slévat).

Dalším vylepšením je odložení štěpení – ke každému listu nebo skupině listů přísluší stránka přetečení, kam se vkládají záznamy, které se už do daného místa nevejdou. Nové vkládání a štěpení je provedeno až tehdy, jestliže se stránka přetečení i všechny příslušné uzly naplní. Takto upravený strom s více než 1 úrovní má vždy všechny listy zaplněné (za předpokladu nepoužití mazání). Přísluší-li stránka přetečení skupinám listů, musím je při mazání a přidávání listů taktéž štěpit a slévat.

### Definice (*B+ stromy*)

*B+ stromy* jsou mírným vylepšením B-stromů pro zrychlení intervalových dotazů: všechny uzly ve stejné úrovni (a nebo jenom listy) jsou spojeny do spojového seznamu (možná je jednosměrná i obousměrná varianta).

### Definice (*B\* stromy*)

*B\* stromy* (řádu  $m$ ) jsou úpravou B-stromů na základě vyvažování stránek. Druhá podmínka B-stromů se upraví tak, že každý uzel kromě kořene a listů má minimálně  $\lceil (2m - 1)/3 \rceil$  a max.  $m$  synů a odpovídající počet dat. záznamů. Listy mají opět jen stejné rozmezí pro počet dat. záznamů. Při vkládání prvků se štěpení odkládá opět do té doby, dokud nejsou plní i sourozenci daného listu; potom se štěpí buď 2 listy do 3, nebo 3 do 4 (buď s pomocí jednoho nebo dvou sousedních sourozenců). Odebírání podobně zahrnuje slévání 3 uzlů do 2 (nebo 4 do 3). Při obém lze ve složitější variantě zapojit ještě více uzlů.

### Definice (*Prefixové stromy (Trie)*)

Tento druh stromů slouží k uložení dat, klíčovaných řetězci. Jde o redundantní stromy, data jsou uložena až v listech; vyhledávací klíče jsou vždy co nejkratší možné prefixy řetězců, nutné k odlišení uzlů. Celé hodnoty klíčů (a další data) se nacházejí až v listech. Při vkládání a štěpení stránek se nějakou heuristikou hledá nejkratší prefix, který by vzniklé stránky oddělil. Vylepšená varianta neukládá u synů předponu klíče, kterou má rodič – je to paměťově efektivnější, ale zvyšuje výpočetní nároky.

**Definice** (*Stromy s proměnnou délkou záznamu*)

Jde o modifikaci B-stromu, která umožňuje do něj uložit záznamy proměnné délky. Listy se neštěpí podle počtu záznamů, ale zhruba na poloviny podle velikosti dat. Druhá podmínka B-stromů se upraví následovně: celková délka záznamů v jednom uzlu je minimálně  $\lceil B/2 \rceil$  a maximálně  $B$  (kde  $B$  je nějaká zvolená hodnota, větš. velikost stránky na disku). Existuje i varianta s podmínkou „2/3“, jako mají B\*-stromy.

Problémem této struktury je tendence delších záznamů ke stoupání ke kořeni, čímž se snižuje arita záznamů. To se řeší hledáním dělicího záznamu s min. délkou tak, aby vzniklé uzly splňovaly podmínky stromu (a je to docela náročné). Navíc štěpení je složitější – 1 stránka se může štěpit na 3 (vloží-li hodně dlouhý záznam), může dojít ke zmenšení stromu při vložení apod., běžně se používá obecný algoritmus nahrazování, jehož speciální případy jsou insert a delete.

**Definice** (*Vícerozměrné B-stromy*)

Používají se, je-li potřeba efektivně hledat záznamy podle více atributů. Jde o propojenou množinu stromů. K jednotlivým atributům přísluší prvky pole odkazů na seznamy stromů, ve kterých se podle daných atributů dá hledat. Pro první atribut je potřeba jen 1 strom, v něm je pro každý klíč odkaz na celý strom 2. atributu (pro další je to podobné). Stromy stejného atributu jsou ve spojovém seznamu. Mohu hledat všechny záznamy, pro které znám hodnoty všech atributů, nebo jenom jejich podmnožinu – vyžaduje to projít více stromů, ale není třeba množinových operací.

## 4 Programovací jazyky a překladače

### Požadavky

- Principy a základy implementace objektově orientovaných jazyků a jazyků s blokovou strukturou, běhová podpora vyšších programovacích jazyků
- Oddělený překlad, sestavení, řízení překladu
- Neprocedurální programování
- Struktura překladače, lexikální, syntaktická analýza
- Interpretované jazyky, virtuální stroje
- Pojmy a principy objektového návrhu
- Generické programování a knihovny
- Návrhové vzory

### 4.1 Principy a základy implementace objektově orientovaných jazyků a jazyků s blokovou strukturou, běhová podpora vyšších programovacích jazyků

*Základní vědomosti: Třída, Dědičnost, Polymorfismus, Obalení, Virtuální funkce. Běhová podpora vyšších programovacích jazyků: Statická podpora a dynamická pod-*

*pora, Rozdělení paměti, Stav paměti před spuštěním, Konstruktory, destruktory globálních proměnných, Volací konvence.*

TODO: jde hlavně o copy & paste z Wikipedie, takže by to chtělo omezit zbytečné kecy a přeložit to, co je anglicky. Otázkou je taky, jestli sem úvodní článek vůbec patří. Ja myslím že jo, ale jistý si nejsem.

## Strukturované programování

Počítačový program je nějakým způsobem zaznamenaný postup počítačových operací, který speciálním způsobem popisuje praktickou realizaci zadané úlohy (tedy algoritmus výpočtu). Program z *procedurálního* úhlu pohledu je vlastně přesná specifikace všech kroků, které musí počítač vykonat, aby došel k cíli, a jejich pořadí. Pro určování pořadí kroků se používají základní operace *řízení toku* – skoky, podmínky, cykly apod.

Jedním z důležitých konceptů procedurálního programování je *strukturované programování* – jeho idea je založena na rozdělení programu na *procedury* (rutiny, podrutiny, metody, funkce), které samy obsahují výčet výpočetních kroků k vykonání, mohou být ale spouštěny opakovaně a z libovolného místa v programu. Jejich výhodou je mnohem názornější pohled na strukturu programu a snazší udržování kódu, než v případě použití jen nejjednoduššího řízení toku (tedy hlavně skoků, které by se ve strukturovaném programování správně používat neměly).

Historically, several different structuring techniques or methodologies have been developed for writing structured programs. The most common are:

- *Dijkstra's structured programming*, where the logic of a program is a structure composed of similar sub-structures in a limited number of ways. This reduces understanding a program to understanding each structure on its own, and in relation to that containing it, a useful separation of concerns.
- *A view derived from Dijkstra's* which also advocates splitting programs into sub-sections with a single point of entry, but is strongly opposed to the concept of a single point of exit.
- *Data Structured Programming*, which is based on aligning data structures with program structures. This approach applied the fundamental structures proposed by Dijkstra, but as constructs that used the high-level structure of a program to be modeled on the underlying data structures being processed.

The two latter meanings for the term "structured programming" are more common. Years after Dijkstra (1969), object-oriented programming (OOP) was developed to handle very large or complex programs.

### **Definice** (*Programovací jazyk s blokovou strukturou*)

A language is described as "block-structured" when it has a syntax for enclosing structures between bracketed keywords, such as an if-statement bracketed by `if..fi`, or a code section bracketed by `BEGIN..END`.

However, a language is described as "comb-structured" when it has a syntax for enclosing structures within an ordered series of keywords. A "comb-structured" language has multiple structure keywords to define separate sections within a block, analogous to the multiple teeth or prongs in a comb separating sections of

the comb. For example, in Ada, a block is a 4-pronged comb with keywords DECLARE, BEGIN, EXCEPTION, END, and the if-statement in Ada is a 4-pronged comb with keywords IF, THEN, ELSE, END IF. Jako jazyk s „hřebenovou“ strukturou by se dalo tedy brát třeba i PL/SQL.

### Poznámka

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Since about 1970 when structured programming began to gain popularity as a technique, most new procedural programming languages have included features to encourage structured programming (and sometimes have left out features that would make unstructured programming easy). Some of the better known structured programming languages are Pascal, C, PL/I, and Ada.

### Věta o strukturovaném programu???

*The structured program theorem* is a result in programming language theory. It states that every computable function can be implemented in a programming language that combines subprograms in only three specific ways. These three control structures are

- Executing one subprogram, and then another subprogram (*sequence*)
- Executing one of two subprograms according to the value of a boolean variable (*selection*)
- Executing a subprogram until a boolean variable is true (*iteration*)

This observation did not originate with the structured programming movement; these structures are sufficient to describe the instruction cycle of a central processing unit, as well as the operation of a Turing machine. Therefore a processor is always executing a "structured program" in this sense, even if the instructions it reads from memory are not part of a structured program.

## Datové a řídicí struktury vyšších programovacích jazyků a jejich implementace

### Řízení toku

In computer science control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions or function calls of an imperative or functional program are executed or evaluated. Within an imperative programming language, a control flow statement is an instruction that when executed can cause a change in the subsequent control flow to differ from the natural sequential order in which the instructions are listed. For non-strict functional languages, functions and language constructs exist to achieve the same ends, but they are not necessarily called control flow statements.

The kinds of control flow statements available differ by language, but can be roughly categorized by their effect:

- continuation at a different statement (jump),

- executing a set of statements only if some condition is met (choice – if-then-else)
- executing a set of statements zero or more times, until some condition is met (loop), s podmínkou na začátku, na konci, uprostřed, nekonečné, s daným počtem opakování
- executing a set of distant statements, after which the flow of control may possibly return (subroutines, coroutines, and continuations),
- stopping the program, preventing any further execution (halt).

Interrupts and signals are low-level mechanisms that can alter the flow of control in a way similar to a subroutine, but are usually in response to some external stimulus or event rather than a control flow statement in the language.

## Výjimky

Výjimky jsou speciálním příkazem řízení toku, vyskytující se v některých vyšších programovacích jazycích. Základní myšlenkou je, že program může na nějakém místě vyhodit výjimku (příkaz `throw`), což způsobí, že provádění programu se zastaví a buď pokračuje tam, kde je výjimka „ošetřena“ (tzv. `catch` blok), nebo pokud takové místo není nalezeno, program skončí s chybou. Během hledání místa ošetření je datová hodnota výjimky uložena stranou a pak může být použita.

Při hledání místa ošetření výjimky (`try`-bloku, následovaného `catch`-blokem se správným datovým typem výjimky) se postupuje zpět po zásobníku volání funkcí, tato technika se nazývá „stack unwinding“ (odvíjení zásobníku). V některých jazycích (Java) lze definovat i akci, která se provede v každém případě, i pokud nastane výjimka, ještě před odvíjením zásobníku – `finally` blok.

## Volací konvence

Při volání procedur a funkcí je nejdůležitější zásobník. Ukládá se na něj

- kam se vrátit po volání
- argumenty funkce (v překladem definovaném pořadí – nutné mít ve všech modulech stejné; většinou se liší v závislosti na programovacím jazyku)
- návratová hodnota funkce
- ukazatel na sémanticky nadřazenou funkci (Pascal)

Dohromady všem těmto datům se někdy říká „aktivační záznam“ procedury. Po skončení funkce je nutné zásobník opět uklidit (vymazat zbytečná uložená data, většinou jen zůstává návratová hodnota) a která část programu to dělá (volaná nebo volající procedura), závisí opět na překladači a konvenci jazyka.

Volací konvence dvou nejtypičtějším jazyků:

- *Pascal*  
uklízí volaná funkce, argumenty se ukládají na zásobník zleva doprava (nejlevější nejdřív, tj. nejhlouběji)
- *C*  
uklízí funkce volající, argumenty se ukládají zprava doleva (tj. nejlevější je na vrcholu zásobníku. Je to kvůli funkcím s proměnným počtem parametrů. Volaná funkce musí podle prvního argumentu poznat, jaký je skutečný počet argumentů. Kdyby byl první argument někde hluboko v zásobníku, tak ví prd.)

## Organizace paměti

Paměť procesu (spuštěného programu) lze rozdělit do několika částí:

- *kód programu (kódový segment)*  
vytvořen při překladu, součást spustitelného souboru, neměnný a má pevnou délku; obvykle bývá chráněn proti zápisu
- *statická data (datový segment)*  
data programu, jejichž velikost je známa již při překladu a jejichž pozice se během programu nemění (je připraven kompilátorem a jeho formát je taktéž zadrátovaný ve spustitelném souboru, u inicializovaných statických dat je tam celý uložený); v jazyce C jde o globální proměnné a lokální data deklarovaná jako `static`, konstanty
- *halda (heap segment)*  
vytvářejí startovacím modulem (C Runtime library), ukládají se sem dynamicky vznikající objekty (`malloc`, `new`) – neinicializovaná data, i seznam volného místa
- *volná paměť*  
postupně jí zaplňuje z jedné strany zásobník a z druhé halda
- *zásobník (stack segment)*  
informace o volání procedur („aktivační záznamy“) — návratové adresy, parametry a návratové hodnoty (nejsou-li předávány v registrech), některé jazyky (Pascal, C) používají i pro úschovu lokálních dat. Typicky roste zásobník proti haldě (od „konce“ paměti k nižším adresám).

### Poznámka (Vnořené funkce)

V Pascalu mohou být funkce definované uvnitř jiné funkce. Ta vnitřní potřebuje přistupovat k proměnným té vnější. Proměnné jsou sice na zásobníku, ale pouhý odkaz na volající funkci nestačí, protože se vnořená funkce může volat rekurzivně. Proto je na zásobníku ukazatel na funkci sémanticky nadřazenou.

## Alokace místa pro různé typy proměnných

- Dynamicky alokované proměnné (přes pointer) se alokují na haldě. Opakovanou alokací a dealokací paměťových bloků různé velikosti vznikají v haldě „díry“ (střídavé úseky volného a naalokovaného místa). Existuje několik strategií pro vyhledání volného bloku požadované velikosti (first-fit, next-fit, buddy systém) a udržení informací o volném místě, které jsou většinou implementovány v knihovních funkcích jazyka (C, Pascal).
- Lokální proměnné se ukládají na zásobník, po skončení funkce, které přísluší, jsou zase odstraněny.
- Globální a statické se ukládají do segmentu pro statická data. Tady se díry tvořit nebudou, protože tyto proměnné vznikají na začátku a zanikají na konci programu (takže se formát segmentu nemění).

## Objektově-orientované programování

### Účel objektového programování

In the 1960s, language design was often based on textbook examples of programs, which were generally small (due to the size of a textbook); however, when programs become very large, the focus changes. In small programs, the most common statement is generally the assignment statement; however, in large programs (over 10,000 lines), the most common statement is typically the procedure-call to a subprogram. Ensuring parameters are correctly passed to the correct subprogram becomes a major issue.

Many small programs can be handled by coding a hierarchy of structures; however, in large programs, the organization is more a network of structures, and insistence on hierarchical structuring for data and procedures can produce cumbersome code with large amounts of "tramp data" to handle various options across the entire program.

Although structuring a program into a hierarchy might help to clarify some times of software, even for some special types of large programs, a small change, such as requesting a user-chosen new option (text font-color) could cause a massive ripple-effect with changing multiple subprograms to propagate the new data into the program's hierarchy. The object-oriented approach is allegedly more flexible, by separating a program into a network of subsystems, with each controlling their own data, algorithms, or devices across the entire program, but only accessible by first specifying named access to the subsystem object-class, not just by accidentally coding a similar global variable name. Rather than relying on a structured-programming hierarchy chart, object-oriented programming needs a call-reference index to trace which subsystems or classes are accessed from other locations.

### Definice (*Objektově orientované programování*)

Na objektově-orientované programování se dá nahlédnout jako na kolekci spolupracujících objektů – v protikladu k tradičnímu pohledu, kdy se za program považuje sled instrukcí pro počítač. V OOP je každý objekt schopný přijímat zprávy, zpracovávat data a posílat zprávy jiným objektům. Na každý objekt se tak dá nahlížet jako na nezávislý „malý stroj“ s vlastní rolí a zodpovědností. Zjednodušeně řečeno jde o dotažení konceptu *data + algoritmy = program*. Data tvoří s kódem, který je spravuje, jeden celek.

### Hlavní koncepty (a formálnější definice)

Objektově orientované programování (zkracováno na OOP, z anglického Object-oriented programming) je metodika vývoje softwaru, založená na následujících myšlenkách, koncepci:

- *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).
- *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.

- *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.
- *Skládání*: Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.
- *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). Umožňuje zacházet s množinou tříd, jako by byly všechny reprezentovány tím samým objektem. Například známá hierarchie: grafický objekt, bod, kružnice. Navíc je to prostředek pro úsporu práce při kódování.
- *Polymorfismus*: odkazovaný objekt se chová podle toho, jaký je jeho skutečný typ. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší. V praxi se tato vlastnost projevuje např. tak, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy (třídy, která přímo či nepřímo z této třídy dědí), která se může chovat jinak, než by se chovala instance rodičovské třídy, ovšem v rámci „mantinelů“, daných popisem rozhraní.

## Třída

*Třída* definuje abstraktní vlastnosti nějakého objektu, včetně obsáhnutých dat (atributy, pole (fields) a vlastnosti (properties)) a věcí, které může dělat (správaní, metody a schopnosti (features)). Například třída *Dog* by obsahovala věci společné pro všechny psy - např. atribúty rasa, barba srsti a schopnosti břechat.

Třídy poskytují v objektovo-orientovaném programu modularitu a strukturu. Třída by typicky měla být rozpoznatelná i ne-programátorovi, který se ale v dané doméně problémů orientuje – tzn. že charakteristiky třídy by měli „dát v kontextu smysl“. Podobně i kód třídy by měl být relativně „self-contained“. Vlastnosti a metody tříd se spolu nazývají i *members*.

## Implementace objektů

Z hlediska jazyka není velký rozdíl mezi složenými datovými typy a třídami. Deklarace třídy obsahuje, stejně jako u složeného dat. typu, datové položky. Navíc ale obsahuje i deklarace funkcí (metod), které s nimi pracují. Některé funkce mohou mít speciální vlastnosti – statické, virtuální, konstruktory, destruktory. Navíc většina jazyků přidává možnost označení kterýchkoliv položek jako veřejné nebo privátní. Třídy mohou někdy (C++, Java) obsahovat i vnořené datové typy (výčty, ... ) a dokonce vnořené třídy.

Za běhu je jedna instance třídy – objekt reprezentována v paměti pomocí:

- datových položek (stejně jako složený datový typ),

- skrytých pomocné položky umožňujících funkci virtuálních metod, výjimek, RTTI a dědičnosti (identifikace typu / jeho velikosti apod.)

*Implementace dědičnosti v C++:* Je-li třída B (přímým či nepřímým) potomkem třídy A, pak paměťová reprezentace objektu typu B obsahuje část, která má stejný tvar jako paměťová reprezentace samostatného objektu typu A. Z každého ukazatele na typ B je možno odvodit ukazatel na část typu A – tato konverze je implicitní, tj. není třeba ji explicitně uvádět ve zdrojovém kódu. Tato konverze může (obvykle pouze při násobné dědičnosti) vyžadovat jednoduchý výpočet (přičtení posunutí).

Z ukazatele na typ A je možno odvodit ukazatel na typ B, jen pokud konkrétní objekt, do kterého ukazuje ukazatel na typ A, je typu B (nebo jeho potomka). Zodpovědnost za ověření skutečného typu objektu má programátor a tuto konverzi je třeba explicitně vynutit přetypováním. Může to znamenat odečtení posunutí v paměti.

### Virtuální funkce

In object-oriented programming (OOP), a virtual function or virtual method is a function whose behavior, by virtue of being declared "virtual," is determined by the definition of a function with the same signature furthest in the inheritance lineage of the instantiated object on which it is called. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

The concept of the virtual function solves the following problem:

In OOP when a derived class inherits from a base class, an object of the derived class may be referred to (or cast) as either being the base class type or the derived class type. If there are base class functions overridden by the derived class, a problem then arises when a derived object has been cast as the base class type. When a derived object is referred to as being of the base's type, the desired function call behavior is ambiguous.

The distinction between virtual and not virtual is provided to solve this issue. If the function in question is designated "virtual" then the derived class's function would be called (if it exists). If it is not virtual, the base class's function would be called.

*Pozdní vazba (late binding; virtual call):* Je-li metoda nějaké třídy virtuální či čistě virtuální, pak všechny metody se stejným jménem, počtem a typy parametrů deklarované v potomcích třídy jsou považovány za různé implementace téže funkce. Která implementace se vybere, tedy které tělo bude zavoláno, se rozhoduje až za běhu programu podle skutečného typu celého objektu. Použije se tělo z posledního potomka, který definuje tuto funkci a je součástí celého objektu. Pozdní vazba má smysl pouze u vyvolání na objektu určeném odkazem.

Pozdní vazba je implementačně umožněná skrytým pointerem na *tabulku virtuálních funkcí* uvnitř každého objektu. Existuje pro každou třídu jedna. Při dědičnosti zůstává v celém objektu odkaz jeden, ale (i pro „nejvnitřnější“ bázovou třídu) odkazuje na tabulku odvozené třídy. V tabulce musí být proto pointery na funkce, deklarované už u bázové třídy, umístěny na začátku (aby bylo možné volat funkce bázové třídy mezi sebou bez změny kódu).

## 4.2 Oddělený překlad, sestavení, řízení překladu

### Struktura programu

Program se skládá z *modulů*:

- Překládány samostatně kompilátorem
- Spojovány linkerem

Modul z pohledu programátora

- Soubor s příponou `.cpp` (`.c`)

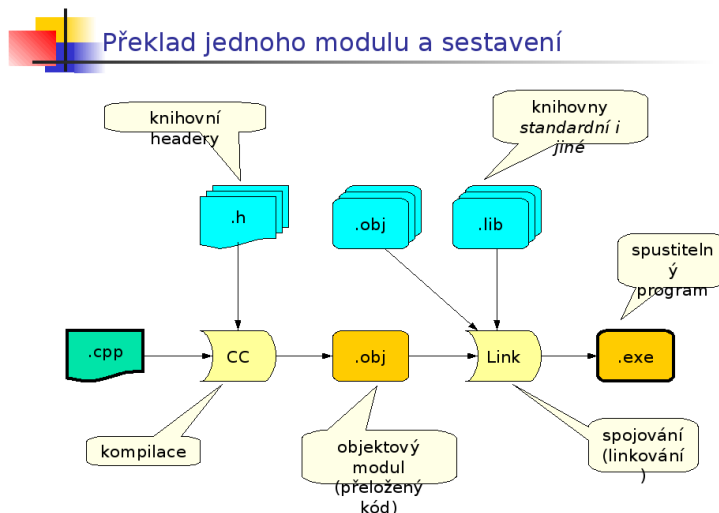
Hlavičkové soubory

- Soubory s příponou `.h`
- Deklarují (a někdy i definují) identifikátory používané ve více modulech
- Vkládány do modulů direktivou `include`
  - Direktivu zpracovává preprocesor čistě textově
  - Preprocesor je integrován v kompilátoru jako první fáze překladu

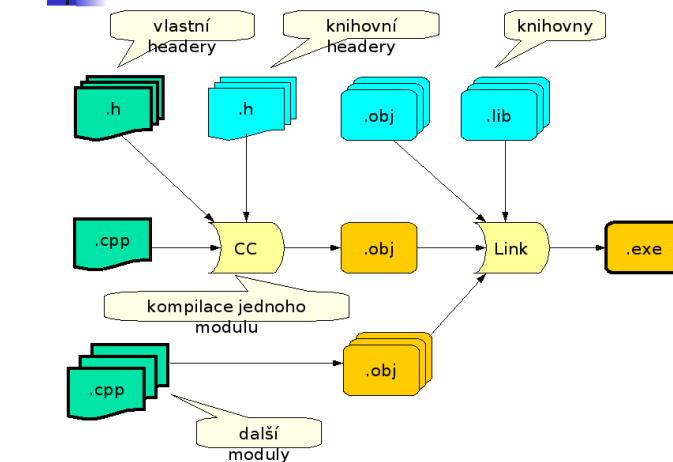
Modul z pohledu kompilátoru

- Samostatná jednotka překladu
- Výsledek práce preprocesoru

### Oddělený překlad



## Překlad více modulů – oddělený překlad



Smysl odděleného překladu modulů je urychlení celkového překladu – nepřekládat to, co se od minula nezměnilo. Oddělený překlad dnes díky automatizaci makefile (viz níže) a integrovanými prostředními není téměř pro programátora vidět.

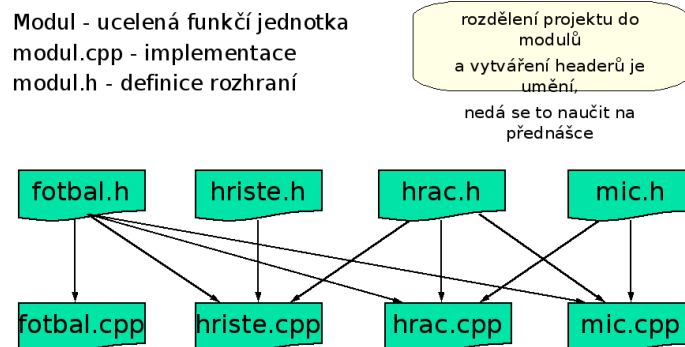
...při tomto slide je vhodné ujasnit si, ako funguje statické a dynamické linkovanie (ako, kde a kedy sa opravujú adresy objektov atď.):

- *Statické linkování*

Po odděleném překladu jednotlivé object moduly ještě neobsahují přímo adresy všech funkcí a externích identifikátorů, jen odkazy na ně. Linker se postará o jejich spojení dohromady. Je nutné, aby jména byla unikátní, takže u přetížených a virtuálních funkcí, jako je v C++, musí být jména zpotvořena tak, aby ukazovala i třídu, namespace, parametry a jejich typy. To má na starosti compiler a říká se tomu *name mangling*.

- *Dynamické linkování*

Nastává po volání operačního systému – zavedení dynamické knihovny do paměti. Jsou dvě možnosti jeho provedení, první je právě při zavádění knihovny, kdy se odkazy na všechny funkce (a mezi nimi navzájem) naplní správnými hodnotami (podle báze adresy, na kterou se knihovna do paměti nahraje). Druhá možnost je použití dvou pointerů při volání funkcí z knihovny – to se vytvoří tabulka skutečných adres, na kterou se z knihovny ukazuje. První možnost trvá déle při zavádění knihovny, druhá je zase pomalejší při provádění, ale umožňuje kód knihovny beze změn sdílet více procesy.



*Linker* je program, který přijímá jeden nebo více objektů generovaných kompilátorem a složí je v jeden spustitelný program.

Objektový kód, nebo objektový soubor je reprezentace kódu, který kompilátor nebo assembler vytvoří zpracováním zdrojového kódu. Objektové soubory obsahují kompaktní kód, často nazývaný „binárky“ :-). Linker se typicky používá na vytvoření spustitelného souboru nebo knihovny spojením (slinkováním) objektových souborů. Základní částí objektového souboru je strojový kód (kód přímo vykonávaný CPU počítače).

## Makefile

Smyslem programu *make* je řízení překladu a linkování. Popis závislostí jednotlivých modulů a hlavičkových na sobě je definován v 1 textovém souboru – *Makefile* (tj. které soubory je nutné mít aktuální/vytvořené pro překlad kterého souboru). Make vždy po změně souboru přeloží jen to, co na něm závisí. Formát souboru make:

```

targets: files;
        commands; #comment; line-begin\
        line contd.;
  
```

Targets – cíle činností / cílové soubory, možno definovat víc, při spuštění make bez parametrů se bere první; univ. nástroj (nejen pro překlad C/C++). Lze definovat i vlastní makra (příkazem <název makra> = <string>) a pak je používat ( $\{\text{makro}\}$ ).

## 4.3 Neprocedurální programování, logické programování

### Neprocedurální programování

*Deklarativní programování* je postaveno na paradigmatu, podle něhož je program založen na tom, co se počítá a ne jak se to počítá. Je zde deklarován vstup a výstup a celý program je chápán jako funkce vyhodnocující vstupy podávající jediný výstup.

Například i webovské stránky jsou deklarativní protože popisují, jak by stránka měla vypadat – titulek, font, text a obrázky – ale nepopisují, jak konkrétně zobrazit stránky na obrazovce.

*Logické programování a funkcionální programování* jsou poddruhy deklarativního programování. Logické programování využívá programování založené na vyhodnocování vzorů - tvrzení a cílů. Klasickým zástupcem jazyka pro podporu tohoto stylu je Prolog.

Tento přístup patří pod deklarativní programování stejně jako funkcionální programování, neboť deklaruje, co je vstupem a co výstupem, a nezabývá se jak výpočet probíhá. Naopak program jako posloupnost příkazů je paradigma imperativní.

*Funkcionální programování* patří mezi deklarativní programovací principy.

Alonzo Church vytvořil formální výpočtový model nazvaný  $\lambda$ -kalkul. Tento model slouží jako základ pro funkcionální jazyky. Funkcionální jazyky dělíme na:

- typované - Haskell
- netypované - Lisp, Scheme

Výpočtem funkcionálního programu je posloupnost vzájemně ekvivalentních výrazů, které se postupně zjednodušují. Výsledkem výpočtu je výraz v normální formě, tedy dále nezjednodušitelný. Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce.

## Prolog

Prolog je logický programovací jazyk. Název Prolog pochází z francouzského *programmation en logique* („logické programování“). Byl vytvořen Alainem Colmerauerem v roce 1972 jako pokus vytvořit programovací jazyk, který by umožňoval vyjadřování v logice místo psaní počítačových instrukcí. Prolog patří mezi tzv. deklarativní programovací jazyky, ve kterých programátor popisuje pouze cíl výpočtu, přičemž přesný postup, jakým se k výsledku program dostane, je ponechán na libovůli systému.

Prolog je využíván především v oboru umělé inteligence a v počítačové lingvistice (obzvláště zpracování přirozeného jazyka, pro nějž byl původně navržen). Syntaxe jazyka je velice jednoduchá a snadno použitelná právě proto, že byl původně určen pro počítačově nepříliš gramotné lingvisty.

Prolog je založen na *predikátové logice prvního řádu* (konkrétně se omezuje na Hornovy klauzule). Běh programu je pak představován aplikací dokazovacích technik na zadané klauzule. Základními využívanými přístupy jsou *unifikace*, *rekurze* a *backtracking*.

Interpret Prologu se snaží nalézt nejobecnější substituci, která splní daný cíl - tzn. nesubstituuje zbytečně, pokud nemusí (použití interních proměnných – \_123 atd.). Za dvě proměnné může být substituována jedna interní proměnná (např. při hledání svislé úsečky – konstantní X souřadnice) – tomu se říká *unifikace* proměnných. Pro proměnnou, jejíž hodnota může být libovolná, se v prologu užívá znak „-“.

Datové typy v prologu se nazývají *termy*. Základním datovým typem jsou *atomy* (začínají malým písmenem, nebo se skládají ze speciálních znaků (+ - \* / ... ,

nebo jsou to znakové řetězce ('text')). Dále „jsou“ v prologu čísla (v komerčních implementacích i reálná), proměnné (velké písmeno) a struktury (definované rekursivně - pomocí funktoru dané arity a příslušným počtem termů, které jsou jeho argumenty – okamžik(datum(1,1,1999),cas(10,10))). Posledním typem proměnných jsou seznamy, které jsou probírány později.

### **Základní principy:**

Programování v Prologu se výrazně liší od programování v běžných procedurálních jazycích jako např. C. Program v prologu je databáze faktů a pravidel (dohromady se faktům a pravidlům paradoxně říká procedury), nad kterými je možno klást dotazy formou tvrzení, u kterých Prolog zhodnocuje jejich pravdivost (dokazatelnost z údajů obsažených v databázi).

Například lze do databáze uložit fakt, že Monika je dívka:

```
dívka(monika).
```

Poté lze dokazatelnost tohoto faktu prověřit otázkou, na kterou Prolog odpoví yes (ano):

```
?- dívka(monika).  
yes.
```

Také se lze zeptat na všechny objekty, o kterých je známo, že jsou dívky (středníkem požadujeme další výsledky):

```
?- dívka(X).  
X = monika;  
no.
```

Pravidla (závislosti) se zapisují pomocí implikací, např.

```
syn(A,B) :- rodič(B,A), muž(A).
```

Tedy: pokud B je rodičem A a zároveň je A muž, pak A je synem B. První částí pravidla (tj. důsledku) se říká hlava a všemu co následuje za symbolem :- (tedy podmínkám, nutným pro splnění hlavy) se říká tělo. Podmínky ke splnění mohou být odděleny buď čárkou (pak jde o konjunkci, musejí být splněny všechny), nebo středníkem (disjunkce), přičemž čárky mají větší prioritu.

### **Příklad:**

Typickou ukázkou základů programování v Prologu jsou rodinné vztahy.

```
sourozenec(X,Y) :- rodič(Z,X), rodič(Z,Y).  
rodič(X,Y) :- otec(X,Y).  
rodič(X,Y) :- matka(X,Y).  
muž(X) :- otec(X,_).  
žena(X) :- matka(X,_).  
matka(marie,monika).  
otec(jiří,monika).  
otec(jiří,marek).  
otec(michal,tomáš).
```

Prázdný seznam je označen atomem [], neprázdný se tvoří pomocí funktoru ' . ' (tečka) - . (Hlava, Tělo). V praxi se to (naštěstí ;-)) takhle složitě rekurzivně zapisovat nemusí, stačí napsat [a,b,c . . .], resp [Začátek | Tělo], kde začátek je výčet prvků (ne seznam) stojících na začátku definovaného seznamu, a tělo je (rekurzivně) seznam (např. [a,b,c | []]).

Aritmetické výrazy se samy o sobě nevyhodnocují, dokud jim to někdo nepřikáže. Takže např. predikát  $5 * 1 = 5$  by selhal. Vyhodnocení se vynucuje pomocí operátoru *is* (pomocí  $=$  by došlo jen k unifikaci) - není to ale ekvivalent „ $=$ “ z jiných jazyků. Tento operátor se musí použít na nějakou volnou proměnnou a aritmetický výraz, s jehož hodnotou bude tato proměnná dále svázaná (jako např.  $X \text{ is } 5 * 1, X=5$  uspěje).

Důležitý je i *operátor řezu* (značíme vykřičníkem). Tento predikát okamžitě uspěje, ale při tom zakáže backtrackování přes sebe zpět. ( $\text{prvek1}(X, [X|L]) :- ! . \text{prvek1}(X, [_|L]) :- \text{prvek1}(X, L) .$  - je-li prvek nalezen, je zakázán návrat = najde jen první výskyt prvku). Dále je důležitá negace ( $\text{not}(P) :- P, !, \text{fail} . \text{not}(P) .$  - uspěje, pokud se nepodaří cíl  $P$  splnit). Řez tedy umožňuje ovlivňovat efektivitu prologovských programů, definovat vzájemně se vylučující použití jednotlivých klauzulí procedury, definovat negaci atd.

## Haskell

Haskell je standardizovaný funkcionální programovací jazyk používající zkrácené vyhodnocování, pojmenovaný na počest logika Haskell Curryho. Byl vytvořen v 80. letech 20. století. Posledním polooficiálním standardem je Haskell 98, který definuje minimální a přenositelnou verzi jazyka využitelnou k výuce nebo jako základ dalších rozšíření. Jazyk se rychle vyvíjí, především díky svým implementacím Hugs a GHC (viz níže).

Haskell je jazyk dodržující *referenční transparentnost*. To, zjednodušeně řečeno, znamená, že tentýž (pod)výraz má na jakémkoliv místě v programu stejnou hodnotu. Mezi další výhody tohoto jazyka patří přísné *typování proměnných*, které programátorovi může usnadnit odhalování chyb v programu. Haskell plně podporuje práci se soubory i standardními vstupy a výstupy, která je ale poměrně složitá kvůli zachování referenční transparentnosti. Jako takový se Haskell hodí hlavně pro algoritmicky náročné úlohy minimalizující interakci s uživatelem.

### Příklady:

Definice funkce faktoriálu:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Jiná definice faktoriálu (používá funkci *product* ze standardní knihovny Haskellu):

```
fac n = product [1..n]
```

Naivní implementace funkce vracující  $n$ -tý prvek Fibonacciho posloupnosti:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

Elegantní zápis řadicího algoritmu quicksort:

```
qsort [] = []
qsort (pivot:tail) =
  qsort left ++ [pivot] ++ qsort right
  where
    left = [y | y <- tail, y < pivot]
    right = [y | y <- tail, y >= pivot]
```

TODO: popsat strážce (případy, otherwise), seznamy, řetězení, pattern matching u parametrů funkcí, lok. definice (where, let) – patří to sem?

## Lisp

Lisp je funkcionální programovací jazyk s dlouhou historií. Jeho název je zkratka pro List processing (zpracování seznamů). Dnes se stále používá v oboru umělé inteligence. Nic ale nebrání ho použít i pro jiné účely. Používá ho například textový editor Emacs, GIMP či konstrukční program AutoCAD.

Další jazyky od něj odvozené jsou například Tcl, Smalltalk nebo Scheme.

**Syntaxe:** Nejzákladnějším zápisem v Lispu je seznam. Zapisujeme ho jako:

```
(1 2 "ahoj" 13.2)
```

Tento seznam obsahuje čtyři prvky:

- celé číslo 1
- celé číslo 2
- text „ahoj“
- reálné číslo 13,2

Jde tedy o uspořádanou čtveřici. Všimněte si, že závorky nefungují tak jako v matematice, ale pouze označují začátek a konec seznamu. Seznamy jsou v Lispu implementovány jako binární strom degenerovaný na jednosměrně vázaný seznam. Co se seznamem Lisp udělá, záleží na okolnostech.

**Příkazy:** Příkazy píšeme také jako seznam, první prvek seznamu je však název příkazu. Například sčítání provádíme příkazem +, což interpreteru zadáme takto:

```
(+ 1 2 3)
```

Interpreter odpoví 6.

**Ukázka kódu:** Program hello world lze zapsat několika způsoby. Nejjednodušší vypadá takto:

```
(format t "Hello, World!")
```

Funkce se v Lispu definují pomocí klíčového slova defun:

```
(defun hello ()
  (format t "Hello, World!")
)
(hello)
```

Na prvních dvou řádcích je definice funkce hello, na třetím řádku je tato funkce svým jménem zavolána. Funkcím lze předávat i argumenty. V následujícím příkladu je ukázka funkce fact, která vypočítá faktoriál zadaného čísla:

```
(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))
  )
)
```

Pro výpočet faktoriálu čísla 6 předáme tuto hodnotu jako argument funkci fact:

```
(fact 6)
```

Návratovou hodnotou funkce bude hodnota 720.

## Logické programování

TODO (není součástí otázek pro obor Programování)

## 4.4 Struktura překladače, lexikální, syntaktická analýza

Zdroj: poznámky a slidy z přednášek Principy překladačů Dr. J. Yaghoba

### Překladače

#### Definice (*Překladač*)

Formální definice: *překladač* je zobrazení  $L_{in} \rightarrow L_{out}$  pro nějaké dva jazyky  $L_{in}, L_{out}$ , vstupní generovaný gramatikou  $G_{in}$ , výstupní generovaný gramatikou  $G_{out}$  nebo přijímaný automatem  $A_{out}$ . Je to takové zobrazení, kde  $\forall w \in L_{in} \exists w' \in L_{out}$ . Pro  $w \notin L_{in}$  zobrazení neexistuje.

Neformálně jde o stroj, který nějaký zdrojový kód (v nějakém zdrojovém jazyce) převádí na cílový kód (v cílovém jazyce) a případně vypisuje chybová hlášení.

Definice neříká nic o třídách jazyků a gramatik, ve kterých překladač operuje. Běžné programovací jazyky jsou „plus minus“ bezkontextové – nebo se na bezkontextové převádějí, aby byly rozpoznatelné něčím prakticky implementovatelným (tedy zásobníkovým automatem, Turingovy stroje jsou poněkud složitě).

### Příklady

Příklady použití překladačů:

- (překvapivě) překlad programů, psaných v nějakém vyšším programovacím jazyce, do strojového kódu cílové platformy
- syntax-highlighting (většinou lexikálně řízený)
- pretty printer
- statické kontroly programu (hledání chyb bez spouštění programů)
- interpretery (např. skriptovacích jazyků, run-time moduly pro interpretované jazyky jako je Java)
- databázové stroje, dotazovací jazyky

## Překlad programu

Program (pro jednoduchost jediný modul) se

1. ze zdrojového kódu v nějakém programovacím jazyce *preprocesorem* (což je taky překladač, upravující zdrojový kód na textové úrovni) převede na textový soubor (přípravený pro další překlad),
2. *překladačem* se převede dál do assemblerového kódu (jde o kód v jiném jazyce, mnohem bližším cílové architektuře – jde o textový popis instrukcí procesoru),
3. *assemblerem* se převádí na „object-file“ – modul, ve kterém už jazyk odpovídá strojovému kódu cílové CPU,
4. nakonec *linker*, resp. *loader* připojí další informace a vytvoří finální spustitelný kód.

## Fáze překladače

Tradičně se překladače dělí na dvě fáze – *front-end* a *back-end*. První z nich je zaměřená hlavně na analýzu zdrojového kódu po lexikální a syntaktické stránce a její převod do nějakého mezikódu, tj. přípravu pro back-end. Úkolem back-endu je pak z předpřipravené formy vygenerovat finální kód v cílovém jazyce.

První fáze se dále dělí na tyto části:

1. *lexikální analýza* – převádí vstupní text do binární formy, na sled identifikátorů a konstant; hodnoty objektů ukládá do spec. tabulek
2. *syntaktická analýza* – abstraktní část, nezajímá se o hodnoty a význam elementů jazyka, úkolem je rozpoznat, zda vstupní slovo (vstup) patří do jazyka; v dnešních překladačích staví tzv. „syntaktický strom“ kódu
3. *sémantická analýza* – zkoumá sémantiku (význam, smysl) elementů jazyka (např. u sčítání proměnných kontrola typů, používání definovaných proměnných atd.)
4. *generování mezikódu* – úzce svázané se sémantickou analýzou, načítá hodnoty lexikálních elementů z tabulek a vytváří binární formu kódu, v ideálním případě nezávislou na vstupním ani výstupním jazyce
5. *optimalizace nad mezikódem* – díky překladu do nějakého abstraktního mezikódu lze nad ním potom provádět různé obecné (teoreticky dokázané) optimalizace, aby byl výsledný kód ekvivalentní s původním, ale rychlejší při provádění cílovým strojem

Backend má na starosti hlavně

1. *generování kódu* – vytváří už kód pro konkrétní cílový stroj / architekturu / CPU.
2. *optimalizace nízkourovňového kódu* – optimalizace, zaměřené na vlastnosti konkrétních CPU a cílový jazyk (tj. takové, které nad obecným mezikódem s vysokou abstrakcí provést nejde)

Všechny fáze překladače (většinou, když se pomínou třeba staší verze GCC a podobně) sdílejí jednotné *tabulky symbolů* – hodnot lexikálních elementů a jiných věcí a obsluhu chyb. Překladač musí rozpoznat všechny chyby, ale bez velké časové režie, navíc nesmí mít falešné popluchy. Taky by neměl vyrábět chyby sám ;-).

V dřívějších překladačích se vstupní kód procházel několikrát, protože nebylo technicky možné ho udržet celý v paměti. Dnes je potřeba většinou jen jeden přechod, ale někdy je nutných víc (např. dopředné skoky v assembleru – nevím ještě jak daleko skáču).

### **Poznámka** (*Syntax-driven compilation*)

Nejdůležitější částí dnešních překladačů bývá syntaktická analýza; provádí se často najednou se sémantickou analýzou a generováním mezikódu – vše mívá na starosti jediný zásobníkový automat. Navíc si často sám vyvolává lexikální analýzu, ta je jím tedy řízená, takže se taková technika označuje *syntaxí řízený překlad*.

### **Automatické generování (částí) překladače**

Protože dnešní programovací jazyky jsou relativně složité (gramatiky které je generují mají řádově stovky prepisovacích pravidel), konstrukce automatů přijímajících takové jazyky „ručně“ je příliš náročná. Proto existují nástroje, které generují některé části překladače – generátor lexikálních analyzátorů – „scannerů“ – (popíšu lexikální elementy a struktury a co s nimi dělá a vypadne mi analyzátor jako kód v jazyce C) je např. *Flex*, pro výrobu parserů (syntaktických analyzátorů) z popisu gramatiky slouží např. *Bison*, *Coco/R* nebo *ANTLR*. Některé známé překladače mají ale i tak ručně generované parsery (GCC).

Existují i generátory generátorů kódu (ale jejich méně, protože to je dost složité) – pro popis výstupního CPU dostanu z instrukčního mezikódu kód přímo pro něj. Instrukční mezikód může být pro více architektur úplně stejný. Příkladem tohoto je *Mono JIT Compiler*.

### **Mezikód**

(Vysokoúrovňový) *mezikód* je vlastně jakési rozhraní pro přechod (rozdělení i spolupráci) mezi front-endem a back-endem. Jde o binární reprezentaci zdrojového kódu, má být nezávislý na vstupním i výstupním jazyce. Pokud tomu tak je, je možné např. kombinovat různé back-endy a front-endy, jako tomu je u GCC (více back-endů pro 1 front-end) nebo .NET (více front-endů). Většinou ale je mezikód o něco posunutý buď více k závislosti na back-endu nebo na front-endu.

Mezikód je možné reprezentovat několika způsoby – např. syntaktickým stromem (vhodné v paměti), postfixovým zápisem (linearizace stromu) nebo tříadresovým kódem (lineární, sekvence příkazů  $x := y \text{ op } z$ ).

### **Graf toku řízení**

Graf toku řízení je graf, vytvářený překladači (větš. pro 1 funkci) za účelem optimalizací a také generování výsledného kódu. Uzly – *základní bloky* – jsou nepřerušované výpočty (bez instrukcí skoků a bez cílů skoků uvnitř bloků), z nichž první instrukce bývá cílem skoku nebo vstupním bodem funkce. Hrany pak reprezentují skoky – pro podmíněné skoky a case příkazy pak z uzlů vede více hran.

### **Lexikální analýza**

**Definice** (*Lexikální analýza*)

Lexikální analýza je část překladače, zodpovědná za rozpoznávání jednotlivých nedělitelných elementů zdrojového jazyka (např. klíčová slova, identifikátory, závorky atd.) a jejich převod na nějakou binární reprezentaci, vhodnou pro syntaktickou analýzu (např. uložení názvů identifikátorů do tabulek symbolů). V zásadě jde o rozpoznávání regulárních výrazů. Historicky šlo o provedení analýzy na celém zdrojovém kódu a přeposlání do další fáze, dnes je většinou ovládaná ze syntaktické analýzy (opakované volání „vrať další element“). Slouží také ke zvětšení „výhledu“ dalších fází (jedním elementem přestává být jeden znak, je jím jeden element vstupního jazyka).

**Definice** (*Token, pattern*)

*Token* je výstup lexikální analýzy – jeden nedělitelný element zdrojového jazyka. Je zároveň vstupem syntaktické analýzy (tam se nazývá *terminál*). Lexikální analýza uvažuje množinu řetězců, které produkuje pro syntaktickou analýzu stejný token (např. díky ignore-caseovosti nebo jako důsledek sloučení všech řetězcových nebo číselných konstant pod stejný token, protože s nimi je dále nakládáno bez ohledu na hodnotu). Množina řetězců, produkujících daný token, se popisuje urč. pravidly – *patternem*, kde se obvykle užívá regulárních výrazů.

**Definice** (*Lexém*)

*Lexém* neboli *lexikální element* je sekvence znaků ve zdrojovém kódu, která (většinou) odpovídá nějakému patternu nějakého tokenu. Např. komentáře ale jako svůj výstup žádný token nemají.

**Definice** (*Literál*)

*Literál* je konstanta ve vstupním jazyce – má svoji hodnotu (atribut), ukládanou do tabulek symbolů.

**Poznámka** (*Atributy tokenů*)

Je-li jeden token rozpoznáván více patterny, nebo je-li to literál, má nějaké další atributy (většinou jenom jeden), které jeho význam upřesňují – např. token „relační operátor“ má zpřesnění „menší nebo rovno“, token „číselný literál“ má zpřesnění „12345“.

**Problémy lex. analýzy**

Mezi některé problémy, které syntaktická analýza musí řešit, patří

- Počítání zarovnání – některé jazyky (Python) mají zarovnání na řádce jako svoji syntaktickou konstrukci
- Identifikátory s mezerami (rozlišit identifikátor od jiné konstrukce, i víceslovné)
- Klíčová slova jako identifikátory (někdy se mohou překrývat)
- Kontextově závislé tokeny – token závisí na jiných informacích (např. **a\*b**; v C – jde o násobení, nebo deklaraci pointerové proměnné), tady je nutné tokeny slučovat pro oba významy ???

## Pozadí lex. analýzy

Na pozadí lexikálního analyzátoru většinou pracuje nějaký konečný automat (protože rozpoznávání regulárních výrazů – hodnotou reg. výrazu je reg. jazyk – je práce pro konečné automaty). Po každém rozpoznáném tokenu je potřeba automat uvést zpět do výchozího stavu.

## Lexikální chyby

Chyba v lexikální analýze nastane tehdy, když konečný automat nemůže pokračovat dál a není v koncovém stavu (např. pokud nalezne neplatný znak, nebo neukončený řetězec na konci řádky apod.). Většina lexikálních analyzátorů (pomineme Turbo Pascal ;-)) by měla být schopna nějakého „rozumného“ zotavení z chyby – vypsat chybu a domyslet chybějící znak nebo neplatný znak ignorovat apod., tj. nezastavit se na první chybě. I logické zotavení může ale scanner úplně rozhodit a ten pak vyhazuje nesmyslné chyby. Je také spousta chyb, které lexikální analýza nepozná a projeví se až u syntaktické analýzy, např. `beign` místo `begin`, chápané jako identifikátor.

## Poznámka (*Bufferování vstupu*)

Syntaktická analýza časově zabere cca 60-80% překladu, takže se pro její urychlení používá bufferování – nečte se po znacích, ale o něco napřed. Problémem pak jsou např. `#include` direktivy (jsou-li ve vstupním jazyce) – v okamžiku vložení jiného souboru je scanner v nějakém stavu apod.; scannery musí mít pak možnost přepínat mezi více vstupními soubory (manipulovat s několika buffery).

## Syntaktická analýza

### Definice (*Syntaktická analýza*)

Syntaktická analýza je část překladače, zodpovědná za:

1. rozhodnutí, zda dané slovo (vstup) patří do zpracovávaného jazyka
2. syntaxí řízený překlad
3. stavbu derivačního stromu (nalezení přepisovacích pravidel ze startovacího neterminálu gramatiky na vstupní posloupnost tokenů – terminálů)

Většina programovacích jazyků je bezkontextová, proto je syntaktická analýza představována zásobníkovým automatem. Syntaktická analýza operuje s gramatikou daného jazyka (snaží se o přepis abstraktních neterminálů na terminály – tokeny jazyka).

### Definice (*Derivační strom*)

Derivační strom je „grafická“ reprezentace slova vstupního jazyka, nebo spíše derivací, které bylo potřeba provést, aby se v gramatice startovací symbol přepsal na dané slovo (posloupnost terminálů). Uzly takového grafu jsou neterminály i terminály gramatiky jazyka (v listech ale jsou jen terminály, ve vnitřních uzlech neterminály). Hrany grafu představují přepsání podle pravidla gramatiky – vedou od neterminálu který se přepisuje, ke všem neterminálům nebo terminálům na

které se přepisuje (mluvíme o bezkontextových gramatikách, takže na levé straně stojí jen jeden neterminál).

Přepsání v gramatice bohužel nemusí být jednoznačné (tj. pro stejnou posloupnost neterminálů existuje více platných derivačních stromů). Příkladem je problém „dangling else“ z jazyků typu Pascal nebo C – mám-li za sebou 2x `if-then` a pak jedno `else`, nemusí být (z gramatiky) jasné, ke kterému `if-then` ono `else` patří. Takové problémy lze (a je nutné) odstranit převodem na jednoznačnou gramatiku (např. přes další neterminál).

### Levá rekurze, levá faktorizace, nebezkontextovost

Levá rekurze v gramatice se objevuje, pokud je v ní neterminál  $A$ , pro který platí  $A \Rightarrow^* A\alpha$  pro nějaké  $\alpha \neq \lambda$ . Tj. přes  $A$  je možné projít kolikrát chci a vytvořit posloupnost  $\alpha\alpha\dots$ . Pokud parser začíná u startovacího neterminálu a hledá derivace na terminály „shora dolů“ (to jeden z druhů scannerů dělá), neví jakou hloubku rekurze má použít. Proto je nutné i levou rekurzi, stejně jako nejednoznačnosti, z gramatiky napřed odstranit její úpravou (zde opět pomůže přechod přes nový neterminál).

Problémem je i levá faktorizace – případ, kdy se v gramatice vyskytnou pravidla jako  $A \rightarrow \alpha\beta$  a zároveň  $A \rightarrow \alpha\gamma$ . I ten je možné řešit úpravou gramatiky (přenos rozhodnutí na pozdější dobu, kdy bude známo, který ze symbolů  $\beta, \gamma$  si vybrat).

Může se také i pro běžné konstrukce z programovacích jazyků stát, že nevyhovují bezkontextovým gramatikám – např. kontrola deklarace identifikátoru před použitím, kontrola počtu parametrů funkce apod. Zde syntaktická analýza bezkontextovým způsobem nestačí a tyto případy je třeba řešit jinak.

### Definice (Názvoslovní gramatik, FIRST a FOLLOW)

Gramatiky se v teorii překladačů označují dvěma až třemi znaky a číslem v závorce, obecně ve tvaru  $PXY(k)$ , kde:

- $X$  je směr čtení vstupu ( $V$  našem případě vždy  $L$ , tj. zleva doprava),
- $Y$  jsou druhy derivace ( $L$  – levé,  $R$  – pravé derivace),
- $P$  označuje prefix (ještě jemnější dělení na třídy u některých gramatik) a
- $k$  představuje *výhled* (lookahead), každý parser totiž vidí jen na jeden nebo několik tokenů dopředu a další neuvažuje. Obvykle je to celé číslo, většinou 1, ale také 0 nebo obecně  $k$ .

Příklady:  $LL(1)$ ,  $LR(0)$ ,  $LR(1)$ ,  $LL(k)$ ,  $SLR(1)$ ,  $LALR(1)$

Množiny *FIRST* a *FOLLOW* představují množinu použitelných neterminálů na urč. místech (začátky řetězců derivovaných z nějakého pravidla, resp. řetězce které mohou následovat po nějakém neterminálu) a používají se pro konstrukci parserových automatů pro nějakou gramatiku.

TODO: formalizovat FIRST a FOLLOW, není to moc složité?

### Definice (Analýza shora dolů)

Analýza shora dolů je technika parserů, kdy se parser snaží najít nejlevější derivaci pro vstupní řetězec. Pokouší se tedy zkonstruovat derivační strom pro daný vstup počínaje kořenem a přidáváním uzlů do stromu – rozhoduje se, podle kterého pravidla gramatiky přepíše. Pravidlo pro odstranění nejednoznačnosti je provádění

jen levých derivací, proto pak automatům vadí levá rekurze a musí se odstraňovat. Techniky pro nalezení přepisovacího pravidla jsou:

- *Rekurzivní sestup* pomocí procedur – pro každý neterminál existuje jedna procedura, která se rozhodne, které pravidlo použije na základě výhledu. Pro rozhodování se sestavují množiny FIRST a FOLLOW každého neterminálu. Potom musí zkontrolovat, jestli pravá strana tohoto pravidla odpovídá vstupu (přičemž výskyt neterminálu na pravé straně znamená zavolání jemu příslušné procedury).
- *Nerekurzivní analýza s predikcí* – je implementováno automatem s explicitním zásobníkem: ten má *parsovací tabulku*, která se liší podle gramatiky (sama práce automatu je vždy stejná) – jsou v ní řádky odpovídající neterminálům a sloupce terminálům, v políčkách jsou přepisovací pravidla nebo chyby. Na zásobník automatu se ukládají symboly gramatiky a ze vstupu se čtou (lineárně terminály). V každém kroku se automat rozhodne podle vstupu a vrcholu zásobníku – je-li tam terminál, vyhodí se a ukazatel vstupu se posune (nebo se skončí); je-li na zásobníku neterminál, rozhoduje se podle tabulky (položka určená vstupem a neterminálem, buďto se použije přepisovací pravidlo nebo skončí chybou). Konstrukce tabulky je opět závislá na množinách FIRST a FOLLOW.

Analýza shora dolů je používána v parserech jednoduchých jazyků ( $LL(1)$  gramatiky s řešením konfliktů zvětšením výhledu na  $k$  terminálů) – v generátorech parserů ANTLR a Coco/R, například.

**Definice** (*Analýza zdola nahoru, LR automat*)

Parsery s analýzou zdola nahoru se pokoušejí najít pozpátku nejpravější derivaci pro vstupní řetězec – zkonstruovat derivační strom pro daný vstup počínaje listy a stavěním zesponu až po kořen stromu. V jednom redukčním kroku je tak podřetězec odpovídající pravé straně pravidla gramatiky nahrazen neterminálem z levé strany pravidla. Analýza zdola nahoru se používá ve např. v generátoru parserů Bison – je schopná vytvořit parsery pro  $LALR(1)$ ,  $GLR(1)$  gramatiky, které jsou oproti  $LL(1)$  parserům „silnější“ (Třída rozpoznávaných jazyků  $LR(1)$  je vlastní nadmnožina  $LL(1)$ ), všechny běžné programovací jazyky zapsatelné bezkontextovou gramatikou sem patří. Navíc se dá implementovat zhruba stejně efektivně jako metoda shora dolů.

V analýze zdola nahoru se používá nějaký zásobníkový automat (*LR automat*) čtoucí ze vstupu, parametrizovaný tabulkami *ACTION* a *GOTO*. Na zásobníku se pak uchovávají stavy a symboly gramatiky (nebo jen stavy). Vrchol zásobníku představuje aktuální stav. V počáteční konfiguraci je pointer vstupu nastavený na začátek a na zásobníku je počáteční stav. V každém kroku podle stavu a tokenu na vstupu adresují tabulku *ACTION* a získám akci k provedení:

- *SHIFT*  $s$  – posune vstup o 1 terminál, který přidá na zásobník spolu s novým stavem  $s$ .
- *REDUCE*  $A \rightarrow \alpha$  – zruší ze zásobníku tolik dvojic stavů a symbolů, jak dlouhé je  $\alpha$ , na zásobník dá  $A$  a stav, který najde v tabulce *GOTO* na pozici odpovídající neterminálu  $A$  a aktuálnímu stavu

- *ACCEPT* – generuje nějaký výstup, slovo je úspěšně rozpoznáno
- *ERROR* – zahlásí chybu

V LR automatech v klidu projdou i gramatiky s levou rekurzí. Obecně se v nich používají nějaké  $LR(k)$  gramatiky, většinou „rozšířené“ – doplněné o „tečky“, ukazatele pozice v pravidlech, které pomáhají s rozpoznáním konce vstupu. Ke konstrukci tabulek ACTION a GOTO jsou opět potřeba množiny FIRST a FOLLOW, nyní rozšířené na  $k$  symbolů.

TODO: přidat popis LR(1) a LALR(1) gramatik?

## 4.5 Interpretované jazyky, virtuální stroje

### Interpretovaný jazyk

Zdrojový jazyk se nepřekládá do kódu skutečného procesoru, ale do kódu nějakého abstraktního stroje... Interpret přeložený do kódu skutečného stroje simuluje zvolený abstraktní stroj.

Důvodem může být např. málo prostoru pro překladač (8-bity a BASIC), nebo přenositelnost – stejný abstraktní stroj může běžet na různých OS i různých architekturách CPU (AS/400, Java).

Tato metoda má i problémy:

- Problém s rychlostí - dá se řešit pomocí JIT (Just-In-Time compilation): Pokud interpret narazí na kód abstraktního stroje, který ještě není přeložen, okamžitě ho přeloží na kód cílového stroje a uloží si ho vedle do své cache
- Problémy s přenositelností - nevhodné změny v chování abstraktního stroje mohou přivodit problémy s přenositelností (Java)
- Jak zvolit abstraktní stroj - aby pokryl chování všech zdrojových jazyků (např .NET)

### Použití dynamické paměti v interpretovaných jazycích

Pokud je dynamická paměť podporována, pak výhradně s garbage collectorem, protože:

- Ukazatele jsou pod kontrolou
- Snadnější programování
- Rychlejší práce s dynamickou pamětí (program obvykle nepotřebuje tolik paměti, aby GC vůbec musel zasahovat, takže se pouze souvisle alokuje; simulátor abstraktního ale obvykle zabere více paměti, než by musel)

### Zo „starých“ textů :-)

In computer programming an emphinterpreted language is a programming language whose implementation often takes the form of an interpreter. Theoretically, any

language may be compiled or interpreted, so this designation is applied purely because of common implementation practice and not some underlying property of a language.

Many languages have been implemented using both compilers and interpreters, including Lisp, C, BASIC, and Python. While Java and C# are translated to a form that is intended to be interpreted, just-in-time compilation is often used to generate machine code.

An *interpreter* is a computer program that essentially compiles and executes (interprets) another computer program "on-the-fly" at runtime.

In computer science the term "interpreter" is sometimes used instead of the term emulator. There are software interpreters and hardware interpreters. We will denote interpreter as a software interpreter. It can also refer to a program that performs compilation as well as emulation. Most interpreters available today generally compile source code when the code is first encountered during program execution, rather than in a separate phase prior to execution.

An interpreter has a number of advantages over a compiler, including:

- because it can be tailored to a specific programming language making it simpler to implement and more compact (BASIC was supported on many early home computers for this reason).
- it allows program implementation to be independent of the characteristics of the host cpu (the Java interpreter is a good example of this).

The main disadvantage of interpreters is that when a program is interpreted, it runs slower than if it had been compiled. The difference in speeds could be tiny or great; often an order of magnitude and sometimes more.

### **Bytecode interpreter**

There is a spectrum of possibilities between interpreting and compiling, depending on the amount of analysis performed before the program is executed. For example, Emacs Lisp is compiled to bytecode, which is a highly compressed and optimized representation of the Lisp source, but is not machine code (and therefore not tied to any particular hardware). This "compiled" code is then interpreted by a bytecode interpreter (itself written in C). The compiled code in this case is machine code for a virtual machine, which is implemented not in hardware, but in the bytecode interpreter. The same approach is used with the Forth code used in Open Firmware systems: the source language is compiled into "F code" (a bytecode), which is then interpreted by an architecture-independent virtual machine.

### **Just-in-time compilation**

Just-in-time compilation, or JIT, refers to a technique where bytecode is compiled to native machine code at runtime; giving the high execution speed of running native code at the cost of increased startup-time as the bytecode is compiled. It has gained attention in recent years, which further blurs the distinction between interpreters, byte-code interpreters and compilation. JIT is available for both the .NET and Java platforms. The JIT technique is a few decades old, appearing in languages such as Smalltalk in the 1980s.

In computer science, a *virtual machine* is software that creates a virtualized environment between the computer platform and its operating system, so that the end user can operate software on an abstract machine.

The original meaning of virtual machine, sometimes called a hardware virtual machine, is that of a number of discrete identical execution environments on a single computer, each of which runs an operating system (OS).

Another meaning of virtual machine is a piece of computer software that isolates the application being used by the user from the computer.

A Java Virtual Machine (JVM) is a set of computer software programs and data structures which implements a specific virtual machine model. This model accepts a form of computer intermediate language, commonly referred to as Java bytecode, which conceptually represents the instruction set of a stack-oriented, capability architecture. This code is most often generated by Java language compilers, although the JVM can also be targeted by compilers of other languages. JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM standard published by Sun (and related contractual obligations).

## 4.6 Pojmy a principy objektového návrhu

TODO: tohle je tupý copy & paste z Wikipedie, předělat/přeložit

**Definice** (*Objektový návrh*)

Object oriented design is part of OO methodology and it forces programmers to think in terms of objects, rather than procedures, when they plan their code. An object contains encapsulated data and procedures grouped together to represent an entity. The 'object interface', how the object can be interacted, is also defined. An object oriented program is described by the interaction of these objects. Object Oriented Design is the discipline of defining the objects and their interactions to solve a business problem that was identified and documented during object oriented analysis.

### Uvažované aspekty pro objektový návrh (prerekvizity)

- *Conceptual model (must have)*: Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.
- *Use case (must have)*: Use case is description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems.
- *System Sequence Diagram (should have)*: System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.
- *User interface documentations (if applicable)*: Document that shows and describes the look and feel of the end product's user interface. This is not man-

datory to have, but helps to visualize the end-product and such helps the designer.

- *Relational data model (if applicable)*: A data model is an abstract model that describes how data is represented and used. If not object database is used, usually the relational data model should be created before the design can start. How the relational to object mapping is done is included to the OO design.

### **Poznámka**

Objektový návrh počítá s vlastnostmi objektového programování, podporovanými objektově-orientovanými jazyky. Jsou to zejména:

- zapouzdření, objekty
- abstrakce, skrytí informací
- dědičnost
- vnější interface
- polymorfismus

### **Postup při objektovém návrhu / Designing concepts**

- Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.
- Identifying attributes.
- Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.
- Define application framework (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.
- Identify persisted objects/data (if applicable): Identify objects that have to be persisted. If relational database is used design the object relation mapping.
- Identify, define remote objects (if applicable)

### **Výstup, výsledek objektového návrhu / Output (deliverables) of object oriented design**

- Class diagram: Class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.

- Sequence diagram: Expend the System Sequence Diagram to add specific objects that handle the system events. Usually we create sequence diagram for important and complex system events, not for simple or trivial ones. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

## 4.7 Generické programování a knihovny

Základní myšlenkou, která se skrývá za pojmem generické programování, je rozdělení kódu programu na algoritmus a datové typy takovým způsobem, aby bylo možné zápis kódu algoritmu chápat jako obecný, bez ohledu na to, nad jakými datovými typy pracuje. Konkrétní kód algoritmu se z něj stává dosazením datového typu.

U kompilovaných jazyků dochází k rozvinutí kódu v době překladu. Typickým příkladem jazyka, který podporuje tuto formu generického programování, je jazyk C++. Mechanismem, který zde generické programování umožňuje, jsou takzvané šablony (templates).

### **Definice** (*Generické programování*)

Generic programming is a style of computer programming where algorithms are written in an extended grammar and are made adaptable by specifying variable parts that are then somehow instantiated later by the compiler with respect to the base grammar. Specifically, the extended grammar raises a non-variable element or implicit construct in the base grammar to a variable or constant and allows generic code to be used, usually implementing common software patterns that are already expressible in the base language.

### **Poznámka** (*Metaprogramování*)

It differs from normal programming in that it somehow invokes within the language a metaprogramming facility. Because it happens as an extension of the language, new semantics are introduced and the language is enriched in this process. It is closely related to metaprogramming, but does not involve the generation of source code (none, at least, that is visible to the user of the language). It is different from programming with macros as well, as the latter refers to textual search-and-replace and is not part of the grammar of the language but implemented by a pre-processor. One exception to this is the macro facility in Common Lisp, in which macros operate on parse trees rather than text.

### **Příklad**

*Třída parametrizovaná typem (kontejner)* — As an example of the benefits of generic programming, when creating containers of objects it is common to write specific implementations for each datatype contained, even if the code is virtually identical except for different datatypes. Instead, a possible declaration using generic programming could be to define a template class (using the C++ idiom):

```
template<typename T>
```

```

class List
{
    /* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;

```

Above T represents the type to be instantiated. The list generated is then treated as a list of whichever type is specified. These "containers-of-type-T", commonly called generics, are a generic programming technique that allows the defining of a class that takes and contains different datatypes (not to be confused with polymorphism, which is the algorithmic usage of exchangeable sub-classes) as long as certain contracts such as subtypes and signature are kept. Although the example above is the most common use of generic programming, and some languages implement only this aspect of it, generic programming as a concept is not limited to generics.

### Příklad

*Typově nezávislá funkce* — Another applicaton is type-independent functions as in the Swap example below:

```

template<typename T>
void Swap(T & a, T & b) //"&" passes parameters by reference
{
    T temp = b;
    b = a;
    a = temp;
}

string hello = "world!", world = "Hello, ";
Swap( world, hello );
cout << hello << world << endl; //Output is "Hello, world!"

```

### Použití v porgramovacích jazycích

The template construct of C++ used in the examples above is widely cited as the generic programming construct that popularized the notion among programmers and language designers and provides full support for all generic programming idioms. D also offers fully generic-capable templates based on the C++ precedent but with a simplified syntax. Java has provided generic programming facilities syntactically based on C++'s since the introduction of J2SE 5.0 and implements the generics, or "containers-of-type-T", subset of generic programming.

### Knihovna

Knihovna (angl. library) je v programování funkční logický celek, který poskytuje služby pro programy. Většinou se jedná o sbírku procedur, funkcí a datových typů,

či při objektově orientovaném přístupu o sadu tříd, uložených v jednom diskovém souboru.

Knihovna poskytuje aplikační programátorské rozhraní (zvané API), které poskytuje funkce této knihovny. Existuje mnoho knihoven pro různé účely, např. pro využívání služeb operačního systému, grafické funkce, řízení periférií, vědeckotechnické výpočty atp.

**Typy knihoven:** Z technického hlediska je možné rozdělit knihovny dle způsobu propojení s programem, který je bude využívat:

- statická knihovna (static linking library) - spojuje se s programem ve při překlada, většinou přípona souboru .lib
- dynamická knihovna (dynamic linking library) - spojuje se s programem až při spuštění programu, většinou spojení knihovny s programem zajišťuje operační systém

Každý typ knihovny má své výhody a nevýhody. Program spojený se statickou knihovnou je přenositelnější, není třeba zajišťovat dostupnost požadovaných dynamických knihoven. Programy spojené s dynamickou knihovnou jsou menší (ve spustitelném souboru se nachází jen výkonný kód programu), výhodou je možnost jednoduše zaměnit požadovanou dynamickou knihovnu za její novější verzi. Kód v dynamicky linkované knihovně je také za běhu sdílen mezi všemi aplikacemi které jej používají, což šetří operační paměť.

Typickou příponou souboru obsahující dynamickou knihovnu je .dll v operačním systému Microsoft Windows a .so v různých Unixech a v Linuxu.

Příklady různých knihoven: standardní knihovna jazyka C, standardní knihovna šablon (Standard Template Library=STL) v jazyce C++, grafické knihovny jako DirectX, OpenGL, SDL apod., matematická knihovna LINPACK pro řešení soustav lineárních rovnic...

## 4.8 Návrhové vzory

Návrhový vzor je pojmenované a popsané řešení typického problému. Princip existují už dlouho: v architektuře – např. barokní styl, literatura – tragický hrdina, romantická novela...

V software se mnohé postupy „vynalézají“ stále znovu – návrhové vzory mají potom pro typickou situaci popisovat:

- jak a kdy mají být objekty vytvářeny
- jaké vztahy a struktury mají obsahovat třídy
- jaké chování mají mít třídy, jak mají spolupracovat objekty

Návrhový vzor (design pattern) je tedy obecně znovupoužitelné řešení problémů často se vyskytujících při návrhu softwaru. Nejedná se o hotový design, který by se dal transformovat přímo na kód – je to víceméně jen popis nebo šablona, jak řešit nějaký problém vyskytující se ve více různých situacích. Objektově-orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami nebo objekty – bez specifikace konkrétních konečných tříd nebo objektů. Algoritmy nejsou považovány za návrhové vzory, protože řeší spíše výpočetní problémy než designové.

Ne všechny softwarové vzory (software patterns) jsou návrhové. Návrhové vzory řeší problémy na úrovni návrhu softwaru (software desing). Jiné druhy vzorů (jako např. architekturní vzory (architectural patterns)) popisují problémy a řešení, které se zaměřují na jiné úrovně.

Základní prvky návrhových vzorů:

- *Název* – co nejvíce vystihující podstatu, usnadnění komunikace – společný slovník
- *Problém* – obecná situace kterou má NV řešit
- *Podmínky* – popis okolností ovlivňujících použití NV a kontextu vhodném pro použití; některé okolnosti mohou být využity při řešení, jiné naopak jsou v konfliktu
- *Řešení* – soubor pravidel a vztahů popisujících jak dosáhnout řešení problému; nejen statická struktura, ale i dynamika chování
- *Souvislosti a důsledky* – detailní vysvětlení použití, implementace a principu fungování; způsob práce s NV v praxi
- *Příklady* – definice konkrétního problému, vstupní podmínky, popis implementace a výsledek
- *Související vzory* – použití jednoho NV nepředstavuje typicky ucelené řešení – řetězec NV

Následující se vyučuje na predmete Návrhové vzory, ktorý je odporúčaný až pre nmgr štúdium:

Kategorie základních NV:

	Creational <i>Tvořivé vzory</i>	Structural <i>Strukturální vzory</i>	Behavioral <i>Vzory chování</i>
Třída	Factory Method	Adapter	Interpreter Template Method
Objekt	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Návrhové vzory je možno klasifikovat dle problémů které řeší. Příklady klasifikace vzorů podle řešených problémů:

- *Fundamental patterns*: ??? nevyhodíme to, je to jen na wiki a nepopsane
- *Creation patterns*: vytváření objektů
- *Structural patterns*: jak jsou třídy a objekty složený do větších struktur
- *Behavioral patterns*: rozdělení funkčnosti a zodpovědnosti mezi objekty; komunikace mezi objekty; umožňuje zaměřit se při návrhu na propojení tříd, ne na běhové detaily
- *Concurrency patterns* ??? tohle taky

## Příklady

Creational patterns:

- Factory method (zajišťuje rozhodnutí o typu vytvářeného objektu při polymorfismu)
- Prototype (jak klonovat objekty)
- Singleton (jak omezit objekt jen na 1 instanci)

Structural patterns:

- Adapter (konverze rozhraní objektů)
- Bridge (oddělení rozhraní a implementace třídy)
- Composite (jak složit více objektů do jednoho s jednotným přístupem)
- Proxy (jak zajistit přístup k jinému objektu přes můj objekt)

- Decorator (jak změnit vlastnosti třídy nebo zajistit rozšířenou funkčnost bez oddělování)

Behavioral patterns:

- Chain of responsibility (jak určit kdo vykoná akci, když z venku přijde požadavek)
- Command (odstínění klienta od zpracování požadavku – klient neurčuje kdy a jak se to provede)
- Iterator (projití prvků pole bez znalosti jejich implementace)
- Visitor (navštívení všech objektů nějaké struktury a práce s nimi, aby všechny nemusely implementovat stejné metody (a měnit se při jejich změně))
- Template (jak mohou odvozené třídy ovlivňovat algoritmy bázové třídy)

TODO: rozšířit, doplnit, opravit :-)

## 5 Architektura počítačů a operačních systémů

### Požadavky

- Architektury počítače
- Procesory, multiprocesory
- Sběrnice, protokoly
- Vstupní a výstupní zařízení
- Architektury OS
- Vztah OS a HW, obsluha přerušení
- Procesy, vlákna, plánování
- Synchronizační primitiva, vzájemné vyloučení
- Zablokování a zotavení z něj
- Organizace paměti, alokační algoritmy
- Principy virtuální paměti, stránkování, algoritmy pro výměnu stránek, výpadek stránky, stránkovací tabulky, segmentace
- Systémy souborů, adresářové struktury
- Bezpečnost, autentifikace, autorizace, přístupová práva
- Druhy útoků a obrana proti nim
- Kryptografické algoritmy a protokoly

### 5.1 Architektury počítače

#### Definice (*Architektura počítača*)

Architektura počítača popisuje „všetko, čo by mal vedieť ten, ktorý programuje v assembleri / tvorí operačný systém“. Teda:

- z akých častí – štruktúra počítača, usporadanie

- význam častí – funkcia časti, ich vnútorná štruktúra
- ako spolu časti komunikujú – riadenie komunikácie
- ako sa jednotlivé časti ovládajú, aká je ich funkčnosť navonok

### Definice (*Víceúrovňová organizace počítače*)

- Mikroprogramová úroveň (priamo technické vybavenie počítača)
- Strojový jazyk počítače (virtuálny stroj nad obvodovým riešením; vybavenie – popis architektúry a organizácie)
- Úroveň operačného systému (doplnenie predchádzajúcej úrovne o súbor makroinštrukcií a novú organizáciu pamäti)
- Úroveň assembleru (najnižšia úroveň ľudsky orientovaného jazyka)
- Úroveň vyšších programovacích jazykú (obecné alebo problémovo orientované; prvá nestrojovo orientovaná úroveň)
- Úroveň aplikačných programů

### Je teda potrebné definovať

- Inštrukčný súbor (definícia prechodovej funkcie medzi stavmi počítača, formát inštrukcie, spôsob zápisu, možnosti adresovania operandov)
- Registrový model (rozlišovanie registrov procesoru: podľa voľby, pomocou určenia registru – explicitný/implicitný register; podľa funkcie registru – riadiaci register/register operandu)
- Definície špecializovaných jednotek (jednotka na výpočet vo floatoch; fetch/decode/execute jednotky)
- Paralelizmus (rozklad na úlohy, ktoré sa dajú spracovať súčasne – granularita (programy, podprogramy, inštrukcie...))
- Stupeň predikcie (schopnosť pripraviť sa na očakávanú udalosť (načítanie inštrukcie, nastavenie prenosu dát) – explicitná predikcia, štatistika, heuristiky, adaptívna predikcia)
- Datové štruktúry a reprezentáciu dát (spôsob uloženia dát v počítači, mappovacie funkcie medzi reálnym svetom a vnútorným uložením, minimálna a maximálna veľkosť adresovateľnej jednotky)
- Adresové konvencie (ako sa pristupuje k dátovým štruktúram – *segment+offset* alebo *lineárna adresácia*; veľkosť pamäti a jej šírka, „povolené“ miesta)
- Řízení (spolupráca procesoru a ostatných jednotiek, interakcia s okolím, prerušenia – vnútorne/vonkajšie)
- Vstupy a výstupy (metódy prenosu dát medzi procesorom a ostatnými jednotkami/počítačom a okolím; zahrňuje definície dátových štruktúr, identifikácia zdroja/cieľa, dátových ciest, protokoly, reakcie na chyby).
- Šíre datových cest
- Stupeň sdílení (na úrovni obvodov – zdieľanie obvodov procesoru a IO; na úrovni jednotiek – zdieľanie ALU viacerými procesormi)

## Základní dvě architektury počítačů

### Von Neumannova

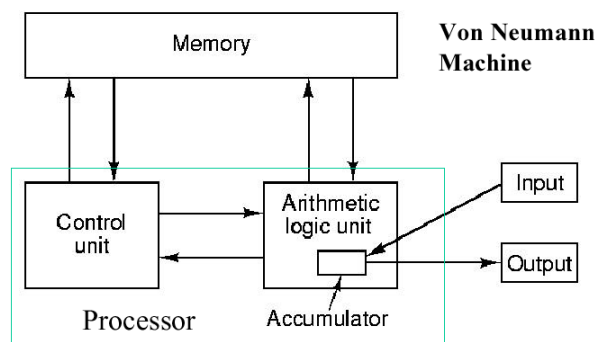
- Počítač se skládá z řídicí jednotky, ALU, paměti a I/O jednotek
- Štruktúra počítača sa nemení typom úlohy (tj. počítač je programovaný obsahem paměti).
- Program se nejprve zavede do paměti, z ní se postupně popořadě vybírají instrukce (a následující krok závisí na předchozím), pořadí lze změnit instrukcemi skoku.
- Do jedné paměti, dělené na buňky stejné velikosti, se ukládají i zpracovávaná data. Data jsou reprezentovaná binárně.
- V každém okamžiku je vykonávána jen jedna činnost. Je to architektura SISD (viz Flynnova taxonomie).

Je pevně daná instrukční sada. Strojová instrukce obsahuje operační znak, který určuje druh operace, počet parametrů atd., a operandovou část – umístění jednotlivých operandů. Vykonat jednu instrukci znamená:

- (fetch) načítat instrukci z paměti do procesoru
- (decode) zjistit o akú instrukciu ide
- (load) pripraviť zdrojové operandy
- (execute) vykonať operáciu
- (store) uložiť cieľové operandy

Při vykonávání programu jsou potřebné různé registry – nejdůležitější jsou: PC (Program Counter, obsahuje adresu následující instrukce), IR (Instruction Register, adresa právě vykonávané instrukce), SP (Stack Pointer, ukazatel na vrchol zásobníku), MAR (memory access register – adresa do operační paměti), MBR (memory buffer register, data čítána/zapisována do paměti).

Struktura jednoprosesorového počítače podle Von Neumanna:



### Harvardská

Vytvořena až po Von Neumannově, liší se hlavně tím, že program se ukládá do jiné paměti než data (tzn. jsou 2 „druhy paměti“ – instrukcí a dat). Příklady jsou DSP procesory a mikrokontrolery (např. AVR od Atmelu, a PIC – mají paměť na program a data a RISC instrukční sadu; výhoda oddělených pamětí je, že můžou

mít různou bitovou hloubku – 8 bitové data, ale 12-, 14- či 16- bitové instrukce (např. ARM musí občas použít více než jednu instrukci na zpracování obsahu plné velikosti)).

Oproti Von Neumannově nehrozí nebezpečí přepsání programu sebou samým, ale kvůli většímu počtu paměťových sběrnic je náročnější na výrobu. Paměť navíc nelze dělit podle potřeby (rozdělení je už dané).

## 5.2 Procesory, multiprocesory

### Definice (*Processor*)

Procesor (CPU – central processing unit) je ústřední výkonnou jednotkou počítače, která čte z paměti instrukce a na jejich základě vykonává program.

Základními součástmi procesora sú:

- řadič nebo řídicí jednotka, která řídí tok programu, tj. načítání instrukcí, jejich dekódování, načítání operandů instrukcí z operační paměti a ukládání výsledků zpracování instrukcí
- sada registrů k uchování operandů a mezivýsledků.
- jedna nebo více aritmeticko-logických jednotek (ALU), které provádí s daty aritmetické a logické operace.
- některé procesory obsahují jednu nebo několik jednotek plovoucí čárky (FPU), které provádí operace v plovoucí řádové čárce.

### Poznámka

Súčasné procesory navyše často obsahujú ďalšie rozsiahle funkčné bloky (cache, rôzne periférie) – ktoré z „ortodoxného hladiska“ nie sú priamo súčasťou *jadra procesoru*. Niektoré procesory môžu obsahovať viac jadier (+logiku slúžiacu k ich vzájomnému prepojeniu). Ďalším trendom je SoC (System on Chip), kde sa na čipe procesora nachádzajú aj ďalšie subsystemy napr. na spracovanie zvuku, grafiky alebo pripojenie externých periférií (takéto riešenia sa využívajú väčšinou v PDA, domácej elektronike, mobiloch atď.).

### Dělení podle instrukční sady

Podľa inštrukčnej sady je možné procesory rozdeliť na:

- **CISC** (Complex Instruction Set Computer): poskytuje rozsiahlu inštrukčnú sadu spolu s rôznymi variantami inštrukcií. Jedna inštrukcia napr. môže vykonať veľa low-level operácií (načítanie z pamäti, vykonať aritmetickú operáciu a výsledok uložiť). Takéto inštrukcie zjednodušovali zápis programov (inštrukcie boli bližšie vyšším programovacím jazykom) a zmenšovali veľkosť programu a počet prístupov do pamäti – čo bolo v 60tych rokoch dôležité. Avšak nie vždy je vykonanie jednej zložitej operácie rýchlejšie ako vykonanie viac menej zložitých miesto toho (napr. kvôli zložitému dekódovaniu a použitiu mikrokódu na volanie jednoduchých „podinštrukcií“). Príkladmi CISC architektúr procesorov sú System/360, Motorola 68000 a Intel x86. V súčasnosti napr. x86 rozkladá zložité inštrukcie na „micro-operations“ ktoré môžu byť pipeline-ou

spracované paralelne a vyšší výkon je tak dosahovaný na väčšom rozsahu inštrukcií. Vďaka tomu sú súčasné x86 procesory minimálne rovnako výkonné ako ozajstné RISC architektúry.

- **RISC** (Reduced Instruction Set Computer): design CPU ktorý uprednostňuje jednoduchšiu inštrukčnú sadu a menšiu zložitosť adresovacích modelov – vďaka čomu je možné dosiahnuť lacnejšiu implementáciu, väčšiu úroveň paralelizmu a účinnejšie kompilátory. Dôvodom vzniku bolo aj nevyužívanie celej CISC inštrukčnej sady a uprednostňovania len obmedzenej podmnožiny (designéri procesorov potom optimalizovali len tieto podmnožiny a tak sa zvyšné inštrukcie používali ešte menej...). Kvôli väčšiemu počtu inštrukcií však musia RISC procesory častejšie pristupovať k pamäti... Príkladmi RISC procesorov sú napr. SPARC a ARM. V architektúrach typu **Post-RISC** jde o spojenie RISCových vlastností s technikami zvýšenia výkonu, jako je out-of-order vykonávaní a paralelizmus.
- **VLIW**: Very Long Instruction Word or VLIW refers to a CPU architecture designed to take advantage of instruction level parallelism (ILP). A processor that executes every instruction one after the other (i.e. a non-pipelined scalar architecture) may use processor resources inefficiently, potentially leading to poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (this is pipelining), or even executing multiple instructions entirely simultaneously as in superscalar architectures. The VLIW approach, on the other hand, executes operation in parallel based on a fixed schedule determined when programs are compiled. Since determining the order of execution of operations (including which operations can execute simultaneously) is handled by the compiler, the processor does not need the scheduling hardware that the three techniques described above require. As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs.
- **EPIC**: (Někdy označován za poddruh VLIW) Explicitly Parallel Instruction Computing (EPIC) is a computing paradigm that began to be researched in the 1990s. This paradigm is also called Independence architectures. It was used by Intel and HP in the development of Intel's IA-64 architecture, and has been implemented in Intel's Itanium and Itanium 2 line of server processors. The goal of EPIC was to increase the ability of microprocessors to execute software instructions in parallel, by using the compiler, rather than complex on-die circuitry, to identify and leverage opportunities for parallel execution. This would allow performance to be scaled more rapidly in future processor designs, without resorting to ever-higher clock frequencies, which have since become problematic due to associated power and cooling issues.

TODO: asi opravit, možná zpřesnit VLIW a EPIC a určitě přeložit

Řekneme, že procesor má *ortogonální instrukční sadu*, pokud žádná instrukce nepředpokládá implicitně použití některých registrů. To umožňuje jednodušší práci algoritmům přidělování registrů v překladačích. Příkladem neortogonální instrukční sady je i x86.

## Další dělení

Ďalej je možné procesory rozdeliť podľa dĺžky operandov v bitoch (8, 16, 32, 64...), ktorý je procesor schopný spracovať v jednom kroku. V embedded zariadeniach sa najčastejšie používajú 4- a 8-bitové procesory. V PDA, mobiloch a videohráčoch 8 resp. 16 bitové. 32 a viac bitov využívajú napr. osobné počítače a laserové tlačiarne.

Dôležitou vlastnosťou je aj taktovacia frekvencia jadra, MIPS (millions of instructions per second) a jeho rýchlosť. V súčasnosti je ťažké dávať do súvislosti výkon procesorov s ich frekvenciou (resp. MIPS) – kým Pentium zvládne na výpočet vo floatoch, jednoduchý 8-bitový PIC na to potrebuje oveľa viac taktov. Ďalším „problémom“ je superskalarita procesorov, ktorá im umožňuje vykonať viacero nezávislých inštrukcií počas jedného taktu.

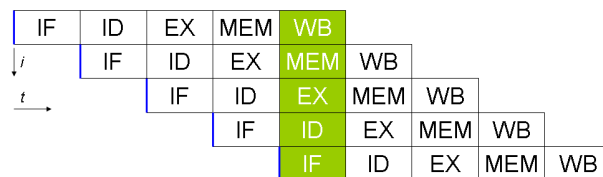
## Techniky pro zvýšení výkonu

Zvyšovať výkon (procesorov) je možné viacerými spôsobmi. Najjednoduchším (a najpomalším) typom je Subskalárny CPU (načíta a spracúva len jednu inštrukciu naraz – preto musí celý procesor čakať kým vykonávanie inštrukcie skončí; je tak zdržovaný dlhšie trvajúcimi inštrukciami).

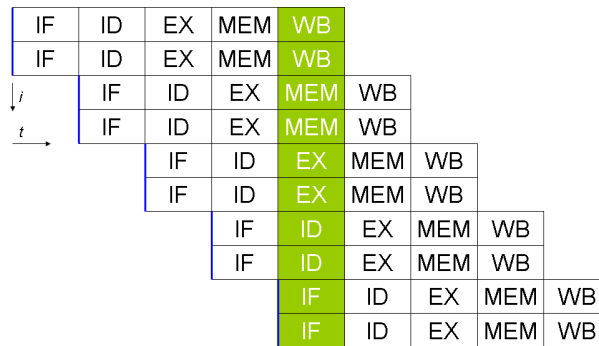


Pokusy o dosiahnutie skalárneho a lepšieho výkonu vyústili do designov ktoré sa správajú menej lineárne a viac paralelne. Čo sa týka paralelizmu v procesoroch, používajú sa dva druhy pojmov na ich klasifikáciu – *Instruction level parallelism* (zvyšovanie rýchlosti vykonávania inštrukcií v procesore a teda zväčšovanie využitia prostriedkov na čípe) a *Thread level parallelism* (zväčšovanie počtu vlákien, ktoré dokáže CPU vykonávať naraz).

- **pipeline:** Zlepšenie je možné dosiahnuť pomocou „instruction pipelining“-u, ktoré je použité vo väčšine moderných procesorov. Umožňuje vykonanie viac ako jednej inštrukcie v jednom kroku vďaka rozloženiu spracovávania inštrukcie na viac menších krokov:



- **superskalarita:** Ďalšia možnosť je použitie superscalar designu, ktorý obsahuje dlhú inštrukčnú pipeline a viacero identických execution jednotiek.



- **Out of order execution**

1. Načtení instrukce, případně její rozdrobení na mikroinstrukce
2. Zařazení do vyčkávací stanice (instruction pool)
3. Instrukce čeká na všechny svoje operandy
4. Instrukce se vykoná ve své výkonné jednotce (je vybírána z instruction poolu nezávisle na ostatních)
5. Výsledky se uchovají ve frontě (reorder buffer)
6. Až se všechny starší instrukce zapíší do registrů, zapíše se výsledek této instrukce (opětovné řazení)

- **Predikce skoků** – hluboké pipeline mají problém, pokud podmíněný skok není proveden; dynamická predikce skoků (historie CPU – vzory nějaké hloubky) vs. statická (bez nápovědy – skok vpřed se neprovede, skok vzad se provede; s nápovědou – překladač odhaduje pravděpodobnost skoku)
- **Spekulativní vykonávání** – vykonávání kódu, který nemusí být zapotřebí; významná disproporce mezi rychlostí CPU a paměti; typické využití je značné předsunutí čtecích operací; CPU provádí i odsouvání zápisových operací
- **Data parallelism**: SIMD instrukcie (napr. multimediálne instrukcie), vektorové procesory...

## Multiprocessory

TODO: jde o copy & paste z Wiki ... předělat česky/slovensky

### Definice (*Multiprocessor*)

O *multiprocessoru* mluvíme, pokud je použito dvou nebo více procesorů (CPU) v rámci jednoho počítačového systému. Termín je také používán mluvíme-li o schopnosti systému využívat více procesorů a/nebo schopnosti rozdělovat úlohy mezi jednotlivými procesory.

### Vztah k datům a instrukcím

In multiprocessing, the processors can be used to execute a single sequence of instructions in multiple contexts (single-instruction, multiple-data or SIMD, often used in vector processing), multiple sequences of instructions in a single context (multiple-instruction, single-data or MISD, used for redundancy in fail-safe systems and sometimes applied to describe pipelined processors or hyperthreading),

or multiple sequences of instructions in multiple contexts (multiple-instruction, multiple-data or MIMD).

### Symetrie

In a multiprocessing system, all CPUs may be equal, or some may be reserved for special purposes. A combination of hardware and operating-system software design considerations determine the symmetry (or lack thereof) in a given system. For example, hardware or software considerations may require that only one CPU respond to all hardware interrupts, whereas all other work in the system may be distributed equally among CPUs; or execution of kernel-mode code may be restricted to only one processor (either a specific processor, or only one processor at a time), whereas user-mode code may be executed in any combination of processors. Multiprocessing systems are often easier to design if such restrictions are imposed, but they tend to be less efficient than systems in which all CPUs are utilized equally.

Systems that treat all CPUs equally are called symmetric multiprocessing (SMP) systems. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including asymmetric multiprocessing (ASMP), non-uniform memory access (NUMA) multiprocessing, and clustered multiprocessing (qq.v.).

### Těsnost spojení multiprocessorů

- **Tightly-coupled** multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory (SMP or UMA), or may participate in a memory hierarchy with both local and shared memory (NUMA). The IBM p690 Regatta is an example of a high end SMP system. Intel Xeon processors dominated the multiprocessor market for business PCs and were the only x86 option till the release of AMD's Opteron range of processors in 2004. Both ranges of processors had their own onboard cache but provided access to shared memory; the Xeon processors via a common pipe and the Opteron processors via independent pathways to the system RAM.
- **Chip multiprocessors**, also known as multi-core computing, involves more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. Mainframe systems with multiple processors are often tightly-coupled.
- **Loosely-coupled multiprocessor** systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system (Gigabit Ethernet is common). A Linux Beowulf cluster is an example of a loosely-coupled system.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly; nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

**SMP** (Symmetric Multiprocessing): viac procesorov so zdieľanou operačnou pamäťou (nutné mechanizmy na zabránenie nesprávnych náhľadov na pamäť a migráciu procesov medzi procesormi). SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

### 5.3 Sběrnice, protokoly

- **Struktura sběrnice:** datové linky, adresové linky, řídicí linky
- **Synchronní přenos** (vznik události je dán hodinovým signálem) vs. **asynchronní přenos** (vznik události je určen předcházející událostí – napr. signalizací začiatku dát)
- **Parametry sběrnice:**
  - *datová šířka* – počet přenášených bitů v jednom okamžiku,
  - *kapacita* – počet bitů přenesených za čas,
  - *rychlost* – kapacita sběrnice normovaná k jednotce informace.
- **Řízení požadavků:**
  - *centrální* – náhodné, dle pořadí vzniku požadavků, prioritní,
  - *distribuované* – kolizní (CSMA/CD), token bus, prioritní linka (daisy chain).
- **Přenos dat po sběrnici** může probíhat buď za účasti procesoru (zdroj → CPU → cíl), nebo bez. Bez procesoru to může být např.:
  - dávkový režim – domluva mezi CPU a řadičem na době obsazení sběrnice (napr. pomocí zdvihnutia „lock flagu“ na zbernici)
  - kradení cyklů – řadič na dobu přenosu „uspí“ procesor (nelze uspat na dlouho, je to technicky náročnější)
  - transparentní režim – řadič rozezná, kdy procesor nepoužívá sběrnici, obvykle nelze větší přenosy najednou
  - DMA (Direct Memory Access) – speciální jednotka pro provádění přenosů dat (mezi zařízeními a pamětí)

Jednou z technik, používaných k přenosu dat po sběrnici řadiči DMA, je *scatter-gather*. Znamená to, že v rámci jednoho přenosu se zpracovává víc ne nutně souvislých bloků dat.

- *scatter* – DMA řadič v rámci 1 přenosu uloží z 1 místa data na několik různých míst (např hlavičky TCP/IP - jinak zbytečné kopírování)
- *gather* – např. při stránkování paměti - načítání stránek, které fyzicky na disku nemusí být u sebe, složení na 1 místo do paměti.

Příklady sběrnice:

- ISA, EISA
- ATA, ATAPI – UltraDMA, Serial-ATA (SATA)

- SCSI (Small Computer System Interface)
- PCI, PCI-X, PCI Express
- AGP (Advanced Graphics Port)
- USB (Universal Serial Bus)
- FireWire (IEEE 1394)
- RS485
- $I^2C$

Příbuzné sběrnic:

- IrDA
- Bluetooth
- Wi-Fi, WiMAX

## 5.4 Vstupní a výstupní zařízení

K I/O zařízením je možné přistupovat dvěma způsoby: pomocí **portů** (speciální adresový port CPU) nebo **pamětovým mapováním** (namapování do fyzické paměti).

Zařízení mají různé charakteristiky:

- **druh** – blokové (disk, síťová karta), znakové (klávesnice, myš)
- **přístup** – sekvenční (datová páska), náhodný (hdd, cd)
- **komunikace** – synchronní (pracuje s daty na žádost – disk), asynchronní („nevyžádaná“ data – síťová karta)
- **sdílení** – sdílené (preemptivní, lze odebrat – síťová karta (po multiplexu OS)), vyhrazené (nepreemptivní – tiskárna, sdílení se realizuje přes *spooling*). Reálně se rozdíly stírají.
- **rychlost** (od několika Bps po GBps)
- **směr dat** – R/W, R/O (CD-ROM), W/O (tiskárna)

Přenos dat mezi zařízením a CPU/paměti:

- **polling** – aktivní čekání na změnu zařízení, přenos provádí CPU
- **přerušování** – asynchronní přerušování od zařízení, přenos provádí CPU
- **DMA (Direct Memory Access)** – zařízení si samo řídí přístup na sběrnici a přenáší data z/do paměti; po skončení přenosu přerušování (oznámění o dokončení)

Cíle I/O software:

- **Nezávislost zařízení** – programy nemusí dopředu vědět, s jakým přesně zařízením budou pracovat – je jedno jestli pracují se souborem na pevném disku, disketě nebo na CD-ROM
- **Jednotné pojmenování** (na UNIXu /dev)
- **Připojení (mount)** – časté u vyměnitelných zařízení (disketa); možné i u pevných zařízení (disk); nutné pro správnou funkci cache OS
- **Obsluha chyb** – v mnoha případech oprava bez vědomí uživatele (velmi často způsobeno právě uživatelem)

## 5.5 Architektury OS

### Klasická struktura – monolitická

Nejstarší, už IBM 360, Unix, Win., všechny služby uvnitř, prováděny ve chráněném módu, jádro poměrně velké, „údajně“ nejrychlejší. Program zavolá službu OS, přes tabulku se zjistí adresa přísl. fce, ta se zavolá a vrátí výsledek. Nevýhoda: horší údržba – je-li v programu chyba, může poškodit zbývající části systému, rozšiřování za běhu je komplikované.

### Virtuální stroje

Původní nápad : Virtual Machine pro IBM360 – oddělit multitasking od OS jako ext. stroj. Nad HW byla další vrstva – „Virtual Machine“ – měla plánovat, vyrábí pro procesy iluzi holého HW; dneska např. VMWare dělá to samé. Pro IBM360 se dalo použít v kombinaci s CMS (jednoúlohový) i původního OS360 (rychlejší než OS360 na holém HW).

Dnes: definuji abstraktní stroj, pro něj překládám programy (.NET, Java) → přenositelnost, kompatibilita (IBM AS400 – desítky let), problém – pomalé.

### Mikrojádro

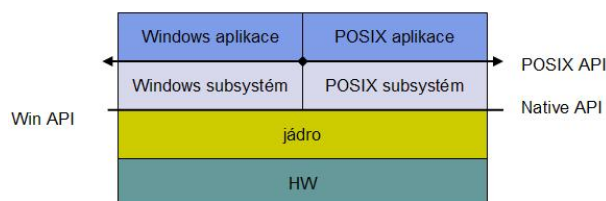
snaha aby část běžící v kernel módu byla co nejmenší (třeba jen cca 10 KB), nejnovější, experimentální, často pro Distribuované OS (dnes už nepoužívané), hodně procesů & komunikace (klient/server), mikrojádro řeší jenom komunikaci.

Filesystém apod. jsou procesy – aplikace jim posílají přes jádro požadavky. Jediný komerční OS – Chorus (ústředny). výhoda: když něco spadne, nepoškodí to zbytek, moduly jdou měnit za běhu, komunikace jde snadno rozšířit na komunikaci po síti.

### Architektura WinNT

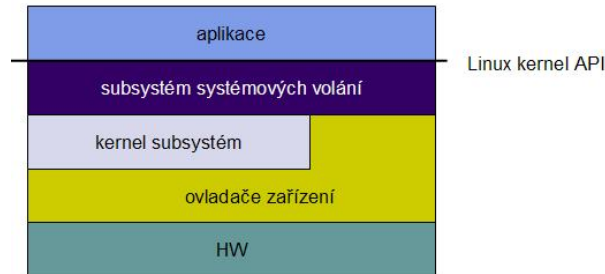
Jádru je poměrně malé (cca 1MB), schopné (pro vyšší vrstvy jsou některé schopnosti skryté), na jeho vzniku se podíleli schopní Unixáři. Byla zde snaha o malou velikost, přenositelnost. Jádro je neutrální vzhledem k vyšším vrstvám, nad ním lze vybudovat různé systémy (Windows subsystém, POSIX, OS/2).

Rozhraní OS a uživ. programů zajišťuje WinAPI, nad ním se nacházejí různé DLL, mezi kernelem a HW je „hardware abstraction layer“, tj. kernel lze jednoduše upravit pro jiné architektury (Alpha, IA-64). Grafické drivery jediné mají přímý přístup k HW (kvůli výkonu), části API (USER, GDI) jsou implementované v jádře, přechod mezi user a kernel režimem zajišťuje ntdll.dll (a je tedy využíván všemi programy). Veškeré služby a aplikace běží v user módu nad jádrem.



## Architektura Linuxu

- Na úrovni SW – přenositelnost; abstrakce HW.
- nad HW – kernel, nad ním systémová volání, hodně podobné Windows.



## 5.6 Vztah OS a HW, obsluha přerušení

### Zjištění změny stavu I/O zařízení:

- *asynchronní přerušení* – zašle zařízení
- *polling* – periodická kontrola stavu zařízení

### Druhy přerušení:

- *synchronní* – záměrně (instrukce TRAP – vstup do OS), výjimky (nesprávné chování procesu) – zpracuje se okamžitě
- *asynchronní* – vnější událost (např. příchod dat) – zpracuje se po dokončení aktuální instrukce

### Obsluha přerušení:

- OS se ujme řízení
- uloží se stav CPU (obsah registrů, čítač, ...)
- analyzuje se přerušení, vyvolá se příslušná obsluha (pokud není přerušení blokováno)
- obslouží se přerušení (např. se zavolá obslužná procedura)
- obnoví se stav CPU a aplikace pokračuje, popř. může dojít k přeplánování

### I/O software (vrstvy):

- uživatelský I/O software
- I/O nezávislý subsystém
- ovladače zařízení
- obsluha přerušení

## Cíle I/O software:

- nezávislost zařízení – programy nemusí vědět, s jakým přesně pracují
- jednotné pojmenování (/dev)
- připojení (mount) – vyměnitelná zařízení
- obsluha chyb

## 5.7 Procesy, vlákna, plánování

### Procesy a vlákna

Systémové volání je interface mezi OS (kernel-space) a uživatelskými programy (user-space).

#### Definice (*Proces*)

*Proces* je instancie vykonávaného programu. Kým program je len súbor inštrukcií, proces je vlastný „výkon“ týchto inštrukcií. Proces má vlastný adresný priestor (pamäť), prostriedky, práva a napr. aj ID (Process ID).

Počas života sa môže proces nachádzať v rôznych stavoch:

- *bežiaci* – jeden proces na procesor,
- *blokováný* – pri použití blokujúceho volania – I/O disku atď.,
- *pripravený* – skončilo blokovanie; spotreboval všetok pridelený čas resp. vrátil riadenie systému, čaká na nové pridelenie procesora,
- *zombie* – po ukončení procesu, keď už nepracuje – ale ešte nebol vymazaný.

#### Definice (*Vlákno*)

*Vlákno* je možnosť pre program ako sa „rozdeliť“ na dva alebo viac zároveň (resp. pseudo-zároveň) vykonávaných úloh. Oproti procesu mu nie je pridelená vlastná pamäť – je to len miesto vykonávania inštrukcií v programe. Oproti procesu sú jeho „atribútmi“ len hodnota programového čítača, stav registrov CPU a zásobník.

Oproti Windows/Solaris neobsahuje Linux priamu podporu pre vlákna. Miesto toho podporuje procesy (zhodou okolností :-)) zdieľajúce pamäť. V samotnom jadre linuxu ale vlákna existujú (kthreads).

### Plánovanie

Pridelovanie procesorového času jednotlivým procesom má na starosti *plánovač*. Plánovanie pritom môže byť preemptívne alebo nepreemptívne (kooperatívne – alla Win16).

#### Ciele plánovania (niektoré z nich sú očividne protichodné):

- Spravodlivosť (každý procesor dostane adekvátnu časť času CPU)
- Efektívnosť (plne vyťažený procesor)
- Minimálna doba odpovede
- Průchodnosť (maximálny počet spracovaných procesov)
- Minimálna réžia systému

### Kritériá plánovania:

- Viazanosť procesu na dané CPU a I/O (presun procesu na iný procesor zaberie veľa prostriedkov)
- Proces je dávkový/interaktívny?
- Priorita procesu (statická (nemenná – okrem „renice“) + dynamická, ktorá sa mení v čase kvôli spravodlivosti)
- Ako často proces generuje výpadky stránok (nejaký popis???)
- Kolik skutočného času CPU proces obdržel

### Algoritmy:

- **FIFO**: nepreemptívny, proces opustí procesor až po skončení
- **Round Robin**: preemptívne rozšírenie FIFO, po skončení časového kvanta je proces presunutý na koniec fronty
- **Plánovanie s viacerými frontami**: niekoľko front, procesu z  $i$ -tej fronty je pridelený procesor až keď vo frontách  $1, \dots, i - 1$  nie je pripravený žiadny proces. Ak proces skončil I/O operáciou, je blokovaný a presunutý do fronty  $i - 1$ , ak skončil preemptiou, je pripravený a presunutý do fronty  $i + 1$ .
- **SMP**: fronta CPU čakajúcich na pripravené procesy (aktívne (spotrebováva energiu) vs. pasívne čakanie (špeciálne inštrukcie)), „vzťah“/afinita procesov k CPU
- **TODO**: Plánovanie windows vs. linux???

## 5.8 Synchronizační primitiva, vzájemné vyloučení

### Pojmy

*Race conditions*: výsledok operace závisí na plánování

*Vzájemné vyloučení (mutual exclusion)*: kritickou operaci provádí nejvýše jeden proces. Podmínky vzájemného vyloučení:

1. Žádné dva procesy nemohou být najednou ve stejné kritické sekci
2. Nemohou být učiněny žádné předpoklady o rychlosti nebo počtu CPU
3. Žádný proces mimo kritickou sekci nesmí blokovat jiný proces
4. Žádný proces nesmí čekat nekonečně dlouho v kritické sekci

*Kritická sekce*: část programu, kde se provádí kritická operace

Metody dosažení vzájemného vyloučení: aktivní čekání (busy waiting) a pasivní čekání/blokování.

### Aktivní čekání

*Vlastnosti*: spotřebovává čas procesoru, vhodnější pro předpokládané krátké doby čekání, nespotebovává prostředky OS, rychlejší.

Je možné použít např. *zakázání přerušování* (vhodné pro jádro OS). Používání *zámků* nefunguje:

```

int lock;
void proc(void) {
    for (;;) {
        nekritická_sekce();
        while (lock != 0);
        lock = 1;
        kritická_sekce();
        lock = 0;
    }
}

```

ale *důsledné střídání* ano (to ale porušuje podmínku 3.)

```

int turn = 0;

void p1(void)                void p2(void)
{                              {
    for (;;) {                for (;;) {
        while (turn != 0);    while (turn != 1);
        kritická_sekce();    kritická_sekce();
        turn = 1;            turn = 0;
        nekritická_sekce();  nekritická_sekce();
    }                          }
}                              }

```

*Petersonovo řešení:*

```

#define N 2                    /* počet procesů */

int turn;
int interested[N];           /* kdo má zájem */

void enter_region(int proc) { /* proc: kdo vstupuje */
    int other;

    other = 1-proc;          /* číslo opačného procesu */
    interested[proc] = TRUE; /* mám zájem o vstup */
    turn = proc;            /* nastav příznak */
    while (turn == proc && interested[other] == TRUE);
}

void leave_region(int proc) { /* proc: kdo vystupuje */
    interested[proc] = FALSE; /* už odcházím */
}

```

*Instrukce TSL* (spin-lock) - je nutné aby ji podporoval HW (všechny současné procesory nějakou mají):

```

enter_region:
    tsl    R,lock                ; načti zámek do registru R a
                                ; nastav zámek na 1
    cmp    R,#0                  ; byl zámek nulový?
    jnz    enter_region          ; byl-li nenulový, znova
    ret                                ; návrat k volajícímu - vstup do
                                ; kritické sekce

leave_region:
    mov    lock,#0               ; ulož do zámku 0
    ret                                ; návrat k volajícímu

```

## Pasivní čekání

*Vlastnosti:* proces je ve stavu blokován, vhodné pro delší doby čekání, spotřebovává prostředky OS, pomalejší.

Postup používající Sleep/Wakeup (implementovány OS, atomické operace - sleep uspí volající proces, wakeup probudí udaný proces) nefunguje (vid' Problém producent/konzument).

**Semaforey...** Sémantika:

```

Down(Semaphore s) {
    wait until s > 0, then s := s-1;
    /* must be atomic once s > 0 is detected */
}

```

(pokud je čítač > 0, sníží čítač o 1 a pokračuje dál; pokud je čítač = 0, operace DOWN se zablokuje a proces je přidán do fronty čekající na tomto semaforu)

```

Up(Semaphore s) {
    s := s+1;    /* must be atomic */
}

```

(pokud je fronta neprázdná, vybere libovolný proces a ten probudí za DOWN; jinak zvětší čítač o 1)

```

Init(Semaphore s, Integer v) {
    s := v;
}

```

Je možné „používat“ aj sémantiku, kde sa hodnota vždy zníži/zvýši o 1 (a je možné sa teda dostať do záporných hodnôt semaforu)... Speciálny (binárny) typ semaforu, kde sú povolené len hodnoty 0 a 1 (v Up sa miesto  $s := s + 1$  volá  $s := 1$ ) sa nazýva *mutex* a používa sa na riadenie prístupu k jednej premennej.

## Monitory

Implementovány prekladačom, lze si představit jako třídu C++ (všechny proměnné privátní, funkce mohou být i veřejné), vzájemné vyloučení v jedné instanci (zajištěno synchronizací na vstupu a výstupu do/z veřejných funkcí, synchronizace implementována blokovacím primitivem OS). ???TODO

## Zprávy

Operace SEND a RECEIVE, zablokování odesílatele/příjemce, adresace proces/mailbox, rendez-vous...

## RWL - read-write lock, bariéry...

Ekvivalence primitiv - pomocí jednoho blokovacího primitiva lze implementovat jiné blokovací primitivum.

Rozdíly mezi platformami: Windows - jednotné funkce pro pasivní čekání, čekání na více primitiv, timeouty. Unix - OS implementuje semafor, knihovna pthread.

## Klasické synchronizační problémy

### Problém producent/konzument

Producent vyrábá předměty, konzument ich spotřebuje. Mezi nimi je sklad pevné velikosti (N). Konzument nemá co spotřebovat, pokud je sklad prázdný; producent přestane vyrábět, pokud je sklad plný.

```
int N = 100;
int count = 0;
void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item);
        if(count==N) sleep ();
        enter_item(item);
        count++;
        if(count == 1) wake(consumer);
    }
}
void consumer(void) {
    int item;
    while(TRUE) {
        if(count==0) sleep ();
        remove_item(&item);
        count--;
        if(count==N-1)
            wake(producer);
        consume_item(&item);
    }
}
```

1. Buffer je prázdný, a konzument právě přečetl count, aby zjistil, či je rovný nule
2. Preplánovanie na producenta
3. Producent vytvorí item a zvýši count
4. Producent zjistí, či je count rovný jedné. Zjistí že ano, čo znamená že konzument bol predtým zablokovaný (pretože muselo byť 0), a zavolá wakeup
5. Teraz môže dôjsť k zablokovaniu: konzument sa uspí, pretože si myslí, že nemá čo zobrať; producent bude chvíľu produkovať a dôjde "preplneniu" ⇒ uspí sa; spí producent aj konzument :o)

### Problém obědvajících filosofů

Pět filosofů sedí okolo kulatého stolu. Každý filosof má před sebou talíř špaget a jednu vidličku. Špagety jsou bohužel slizké a je třeba je jíst dvěma vidličkami. Život filosofa sestává z období jídla a období přemýšlení. Když dostane hlad, pokusí se vzít dvě vidličky, když se mu to podaří, nají se a vidličky odloží.

### Problém ospalého holiče

Holič má ve své oficíně křeslo na holení zákazníka a pevný počet sedaček pro čekající zákazníci. Pokud v oficíně nikdo není, holič se posadí a spí. Pokud přijde první zákazník a holič spí, probudí se a posadí si zákazníka do křesla. Pokud přijde zákazník a holič už stříhá a je volné místo v čekárně, posadí se, jinak odejde.

## 5.9 Zablokování a zotavení z něj

Prostředek je cokoliv, k čemu je potřeba hlídat přístup (HW zařízení – tiskárny, cpu; informace – záznamy v DB). Je možné je rozdělit na *odnímatelné* (lze odejmout procesu bez následků – CPU, paměť) a *neodnímatelné* (nelze odejmout bez nebezpečí selhání výpočtu – CD-ROM, tiskárna... tento druh způsobuje problémy).

Práce s prostředky probíhá v několika krocích: *žádost o prostředek* (blokující, právě tady dochází k zablokování), *používání* (např. tisk), *odevzdání* (dobrovolné/při skončení procesu).

Množina procesů je *zablokována*, jestliže každý proces z této množiny čeká na událost, kterou může způsobit pouze jiný proces z této množiny.

### Coffmanovy podmínky

Splnění těchto podmínek je nutné pro zablokování:

1. **Vzájemné vyloučení** – každý prostředek je buď vlastněn právě jedním procesem nebo je volný.
2. **Drž a čekej** – procesy aktuálně vlastní nějaké prostředky mohou žádat o další.
3. **Neodnímatelnost** – přidělené prostředky nemohou být procesům odebrány.
4. **Čekání do kruhu** – existuje kruhový řetěz procesů, kde každý z nich čeká na prostředek vlastněný dalším článkem řetězu.

### Řešení zablokování

- **Přstrosí algoritmus** – Zablokování se ani nedetekuje, ani se mu nezabraňuje, ani se neodstraňuje, Uživatel sám rozhodne o řešení (kill). Nespotřebovává prostředky OS – nemá režii ani neomezuje podmínky provozu. (Nejčastější řešení – Unix, Windows)
- **Detekce a zotavení** – Hledá kružnici v orientovaném grafu (hrany vedou od procesu, který čeká, k procesu, který prostředek vlastní), pokud tam je kružnice, nastalo zablokování a je třeba ho řešit:
  - *Odebrání prostředku* – dohled operátora, pouze na přechodnou dobu
  - *Zabíjení procesů z cyklu* (resp. mimo cyklus vlastní identický prostředek)

- *Rollback* (OS ukládá stav procesů, při zablokování se některé procesy vrátí do předchozího stavu  $\Rightarrow$  ztracena práce... obdoba u DB)
- **Vyhýbání se** – Bezpečný stav (procesy/prostředky nejsou zablokovány, existuje cesta, jak uspokojit všechny požadavky na prostředky spouštěním procesů v jistém pořadí); Vid. bankéřův algoritmus. Nutné je předem znát všechny prostředky, které budou programy potřebovat; OS pak dává prostředky tomu, který je nejbliž svému maximu potřeby a navíc pro který je prostředků dost na dokončení. Dnes se moc nepoužívá.
- **Předcházení (prevence)** – napadení jedné z Coffmanovy podmínek
  1. *Vzájemné vyloučení – spooling* (prostředky spravuje jeden systémový proces, který dohlíží na to, aby jeho stav byl konzistentní (tiskárna) – pozor na místo na disku)
  2. *Drž a čekej* – žádat o všechny prostředky před startem procesu. Nejprve všechno uvolnit a pak znovu žádat o všechny najednou
  3. *Neodnímatelnost* – vede k chaosu
  4. *Čekání do kruhu* – nejvýše jeden prostředek – všechny prostředky jednoznačně očíslovány, procesy mohou žádat o prostředky jen ve vzestupném pořadí
- *Dvojfázové zamykání* – nejprve postupně všechno zamykám (první fáze). Potom se může pracovat se zamčenými prostředky – a na závěr se už jen odemyká (druhá fáze) – vid' transakční zpracování u databází ((striktní/konzervativní) dvoufázové zpracování)

**Bankéřův algoritmus:** Bankéř má klienty a těm slíbil jistou výšku úvěru. Bankéř ví, že ne všichni klienti potřebují plnou výši úvěru najednou. Klienti občas navštíví banku a žádají postupně o prostředky do maximální výšky úvěru. Až klient skončí s obchodem, vrátí bance vypůjčené peníze. Bankéř peníze půjčí pouze tehdy, zůstane-li banka v bezpečném stavu.

## 5.10 Organizace paměti, alokační algoritmy

**Hierarchie paměti** (směrem odshora dolů roste velikost, cena na bajt a rychlost klesá – a naopak...):

- *registry CPU* — 10ky-100vky bajtů (IA-32: obecné registry pár 10tek), IA-64 – až kB (extrém), stejně rychlé jako CPU.
- *cache* — z pohledu aplikací není přímo adresovatelná; dnes řádově MB, rozdělení podle účelu, několik vrstev. L1 cache (cca 10ky kB) – dělené instrukce/data; L2 (cca MB) sdílené instr&data, běží na rychlosti CPU (dřív bývala pomalejší), servery – L3 (cca 10MB). Vyrovnává rozdíl rychlosti CPU a RAM. Využívá lokality programů – cyklení na místě; sekvenčního přístupu k datům. Pokud nenačtu co chci v cache – „cache-miss“, načítá se potřebné z RAM (po blocích), jinak (v 95-7% případů) nastane „cache-hit“, tj. požadovaná data v cache opravdu jsou a do RAM nemusím.
- *hlavní paměť* (RAM) — přímo adresovatelná procesorem, 100MB – GB; pomalejší než CPU; CAS – doba přístupu na urč. místo – nejvíc zdržuje (v 1

sloupci už čte rychle, dat. tok dostatečný), další – latence – doba než data dotečou do CPU – hraje roli vzdálenost (AMD- integrovaný řadič v CPU)

- *pomocná paměť* — není přímo adresovatelná, typicky HDD; náh. přístup, ale pomalejší. 100GB, různé druhy – IDE, SATA, SCSI; nejvíc zdržuje přístupová doba (čas seeku) cca 2-10ms; obvykle sektor – 512 B; roli hraje i rychlost otáčení (4200 – 15000 RPM) – taky řádově ms.
- *zálohovací paměť* — nejpomalejší, z teorie největší, dnes ale neplatí; typicky – pásky; pro větší kapacitu – autoloaery ; sekvenční přístup; dnes – kvůli rychlosti často zálohování RAIDem.

**Správce paměti:** část OS, která spravuje paměťovou hierarchii se nazývá správce paměti (memory manager):

- udržuje informace o volné/plné části paměti
- stará se o přidělování paměti
- a „výměnu paměti s diskem“

### Přiřazení adresy

- při překladu (je již známo umístění procesu, generuje se absolutní kód, PS: statické linkování)
- při zavádění (OS rozhodne o umístění – generuje se kód s relokacemi, PS: dynamické linkování)
- za běhu (proces se může stěhovat i za běhu, relokační registr)

**Overlay** – Proces potřebuje více paměti než je skutečně k dispozici. Programátor tedy rozdělí program na nezávislé části (které s v paměti podle potřeby vyměňují) a část nezbytnou pro všechny části...

**Výměna (swapping)** – dělá se, protože proces musí být v hlavní paměti, aby jeho instrukce mohly být vykonávány procesorem... Jde o výměnu obsahu paměti mezi hlavní a záložní.

**Překlad adresy** – nutný, protože proces pracuje v logickém (virtuálním) adresovém prostoru, ale HW pracuje s fyzickým adresovým prostorem...

**Spojité přidělování** – přidělení jednoho bloku / více paměťových oddílů (*pevně* – paměť pevně rozdělena na části pro různé velikosti bloků/*volně* – v libovolné části volné paměti může být alokován libovolně veliký blok)

**Informace o obsazení paměti** – bitová mapa / spojový seznam volných bloků (spojování uvolněného bloku se sousedy)

### Alokační algoritmy:

- *First-fit* – první volný dostatečné velikosti – rychlý, občas ale rozdělí velkou díru
- *Next-fit* – další volný dostatečné velikosti – jako First-fit, ale rychlejší
- *Best-fit* – nejmenší volný dostatečné velikosti – pomalý (prohledává celý seznam), zanechává malinké díry (ale nechává velké díry vcelku)
- *Worst-fit* – největší volný – pomalý (prohledává celý seznam), rozdělí velké díry

- *Buddy systém* – paměť rozdělena na bloky o velikosti  $2^n$ , bloky stejné velikosti v seznamu, při přidělení zaokrouhlit na nejbližší  $2^n$ , pokud není volný, rozštípnou se větší bloky na příslušné menší velikosti, při uvolnění paměti se slučují sousední bloky (buddy)

### Fragmentace paměti:

- *externí* – volný prostor rozdělen na malé kousky, pravidlo 50% – po nějaké době běhu programu bude cca 50% paměti fragmentováno a u toho to zůstává
- *interní* – nevyužití celého přiděleného prostoru
- *sesypání* – pouze při přiřazení adresy za běhu, nebo segmentaci – nelze při statickém přidělení adresy

## 5.11 Principy virtuální paměti, stránkování, algoritmy pro výměnu stránek, výpadek stránky, stránkovací tabulky, segmentace

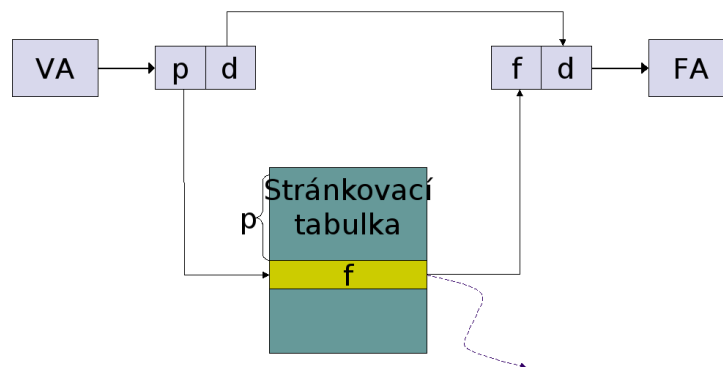
### Virtuální paměť

- procesy pracují s virtuální adresou
- mapování adresy na fyzickou - mapovací tabulky
- obraz virtuální paměti (VAP) částečně v RAM a částečně na disku
- dříve iluze větší paměti, dnes hlavně ochrana přístupu
- stránkování / segmentace

### Stránkování

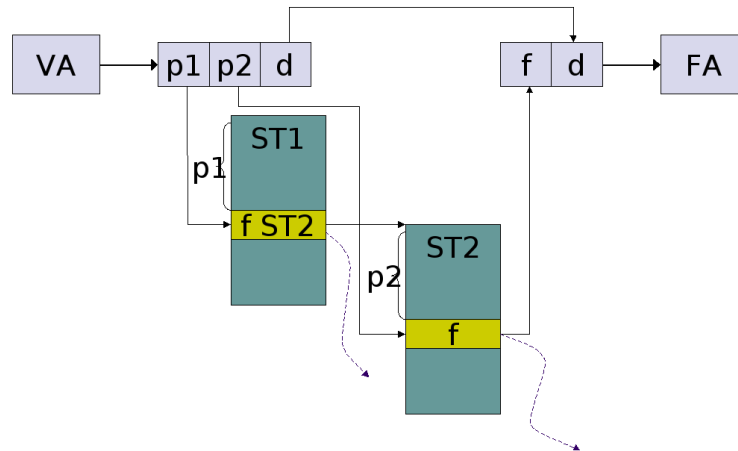
podporované všemi velkými CPU a OS, jednorozměrný VAP

- VAP rozdělen na stránky (velikost je mocnina 2), FAP na rámce (úseky stejné délky)
- převod stránkovací tabulkou, příznak existence mapování (výpadek stránky → synchronní přerušování)



- umožňuje *oddělené VAP* i *sdílenou paměť* - mapování virtuální stránky 2 procesů na jednu fyzickou

- víceúrovňové stránkování (např. kvůli velikosti)

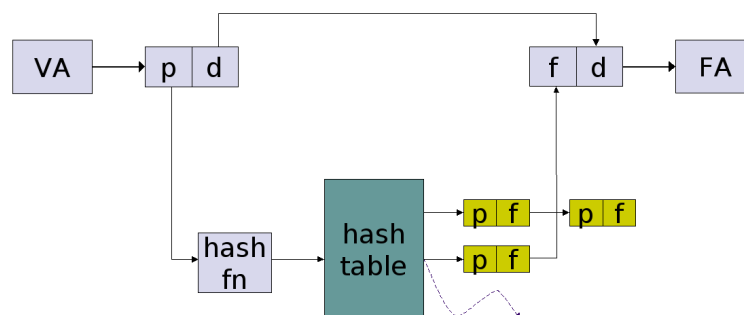


- TLB (Translation Lookaside Buffer) - asociativní paměť sloužící na rychlé vyhledání mapování virtuální stránky na fyzickou, využívá lokalitu chování programů

Platné	Stránka	Dirty	Ochrana	Rámec
1	125	1	RW	654
1	44	0	RW	132
1	485	1	RW	12
0				
1	55	0	R X	142
1	111	0	R X	54
1	108	1	RW	54
1	88	0	R X	543

...nulaúrovňové stránkování - používá pouze TLB, řízeno také OS (oblíbené u 64-bitových CPU - UltraSPARC III)

- inverzní stránkování (např. když FAP je menší než VAP, 64-bitové CPU - IA-64)



Akce vykonávané při výpadku stránky:

- výjimka procesoru
- uložit stav CPU (kontext)

- zjistit VA
- kontrola platnosti adresy a práv
- nalezení volného rámce
- zrušit mapování na nalezený rámec
- pokud je vyhazovaný rámec vyhazován, spustit ukládání na disk
- načíst z disku požadovanou stránku do rámce
- zavést mapování
- obnovit kontext

Při implementaci stránkování je nutno brát v úvahu:

- *znovuspuštění instrukce* — je potřeba aby procesor po výpadku zkusil přístup do paměti znova. dnes umí všechny CPU, např. 68xxx - problémy (přerušeni v půlce instrukce)
- *sdílení stránek* — jednomu rámci odp. víc stránek → pokud s ním něco dělám, týká se to všech stránek! musím vše ost. odmapovat. musím si pamatovat mapování pro každý rámec - obrácené tabulky.
- *odstranění položky z TLB při rušení mapování* — nestačí změnit tabulky, musí se vyhodit i z TLB (kde to může, ale nemusí být). problém - u multiprocesorů má každá CPU vlastní TLB, tabulky jsou sdílené → CPU při rušení mapování musí poslat interrupt s rozkazem ke smazání všem (i sobě), počkat na potvrzení akce od všech.

### Algoritmy pro výměnu stránek

- **Optimální stránka** (v okamžiku výpadku stránky vybírám stránku, na níž se přistoupí za největší počet instrukcí) - nelze implementovat
- **NRU** (Not Recently Used) - každá stránka má příznaky Accessed a Dirty (typicky implementovatelné v HW, možno simulovat SW); jednou za čas se smažou všechna A; při výpadku rozdělím stránky podle A,D a vyberu stránku z nejnižší neprázdné třídy:

	A	D
0	0	0
1	0	1
2	1	0
3	1	1

- **FIFO** (vykazuje anomálie - Belady (zvětšení počtu výpadků stránky, když zvýšíme počet stránek v paměti)), druhá šance (úprava FIFO; pokud A=1, zařadím na konec FIFO... nevykazuje anomálie)
- **Hodiny** - modifikace druhé šance: kruhový zoznam stránek + iterátor na ukazující na nejstarší stránku v zoznamu. Při výpadku (a neexistenci volého rámce) se zjistí, jestli má \*iterator nastavený příznak Accessed. Jestli ne, tato stránka bude nahrazena - v opačném případě se Accessed příznak zruší a iterator++. Toto se opakuje, dokud nedojde k výměně...

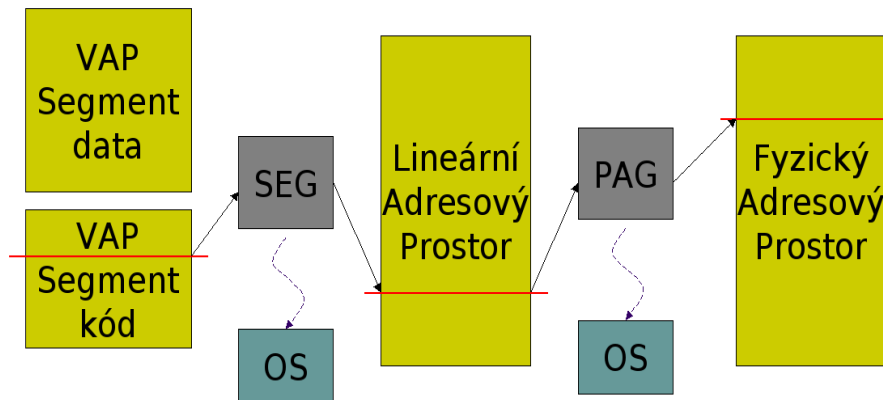
- **LRU** (Least Recently Used) - často používané stránky v posledním krátkém časovém úseku budou znovu použity, čítač použití stránek, možné implementovat v HW
- **NFU** (Not Frequently Used) - SW simulace LRU, SW čítač ke každé stránce; jednou za čas projdu všechna A a přičtu je k odpovídajícím čítačům; vybírám stránku s nejnižším čítačem; nezapomíná - je možná modifikace se stárnutím čítače

## Segmentace

dnes pouze Intel IA-32, dvojrozměrný VAP

- rozdělení programu na segmenty (napr. podle částí s různými vlastnostmi - kód, data, zásobníky...), různé délky segmentů, které mohou měnit svoji délku za běhu
- VAP dvourozměrný (segment, offset), FAP jednorozměrný (vyzerá jako při spojitém přidělování paměti)
- segmentová převodní tabulka (VA se skládá ze dvou částí S:D, v tabulce se najde adresa segmentu S...k této adrese se poté přičte D, co je umístění adresy v FA), příznak existence mapování
- při výpadku je nutné měnit celý segment (ty mohou být velké), je možné segmenty sesypat - ale nelze mít segment větší než FAP

Segmentaci je možné kombinovat se stránkováním (odstraňuje nevýhody segmentace, neprovádí se výpadky segmentů):



## 5.12 Systémy souborů, adresářové struktury

### Definice (soubor)

*Soubor* je pojmenovaná množina souvisejících informací, která leží v pomocné paměti (na disku).

*Soubor* je abstrakce, která umožňuje uložit informaci na disk a později ji přečíst. Abstrakce odstiňuje uživatele od podrobností práce s disky.

## Soubory

- pojmenování souboru (umožňuje uživateli přístup k jeho datům; přesná pravidla pojmenování určuje OS - malá vs. velká písmenka, speciální znaky, délka jména, přípony a jejich význam)
- atributy souborů (opět určuje OS) - jméno, typ, umístění, velikost, ochrana, časy, vlastník, ...
- struktura souborů - sekvence bajtů / sekvence záznamů / strom
- typy souborů - běžné soubory, adresáře (systémové soubory vytvářející strukturu souborového systému), speciální soubory (znakové/blokové, soft linky)
- přístup
  - **sekvenční** – pohyb pouze vpřed, OS může přednačítat
  - **náhodný** – možno měnit aktuální pozici
  - **paměťově mapované soubory** – pojmenovaná virtuální paměť, práce se souborem instrukcemi pro práci s pamětí, ušetří se kopírování paměti; mají i problémy (přesná velikost souboru, zvětšování souboru, velikost souborů)
- volné místo na disku - bitmapa / spojový seznam volných bloků

## Uložení souborů

Soubory se ukládají na disk po blocích

- souvislá alokace - souvislý sled bloků
- spojovaná alokace - blok odkazuje na další
- indexová alokace - inode (UNIX)

## Adresáře

- zvláštní typ souboru
- operace nad adresáři - hledání souboru / vypsání adresáře / přejmenování, vytvoření, smazání souboru
- kořen, aktuální adresář, absolutní/relativní cesta
- hierarchická struktura
  - *strom* – jednoznačné pojmenování (cesta)
  - *DAG* – víceznačné pojmenování, ale nejsou cykly
  - *obecný graf* – cykly vytváří problém při prohledávání
- implementace adresářů - záznamy pevné velikosti, spojový seznam, B-stromy

## Co musí filesystém umět?

musí splňovat 3 věci: *správu souborů* (kde jsou, jak velké), *správu adresářů* (převod jméno ↔ id) (někdy to dělá jiný prostředek, dnes větš. umí FS sám), *správu volného místa*. někdy mohou být i další (odolnost proti výpadkům)

Velikost bloků – blok = nejmenší jednotka pro práci s diskem; disk pracuje s min. 1 sektorem (typicky 512 B) - někdy by pak bylo moc bloků → OS sdružuje

několik sektorů lineárně vedle sebe = 1 blok. velikost: velké = rychlejší práce, ale vnitřní fragmentace (průměrný soubor má cca pár KB), malé = malá vnitřní fragmentace, větší režie na info o volném místě/ umístění souboru (zabírá víc bloků!), navíc fragmentace souborů → zpomalení. dnes má blok cca 2-4KB.

## Linky

- **Hard link** – Na jedna data souboru se odkazuje z různých položek v adresářích
- **Soft link** – Speciální soubor, který obsahuje jméno souboru

## Příklady

- FAT – [http://en.wikipedia.org/wiki/File\\_Allocation\\_Table](http://en.wikipedia.org/wiki/File_Allocation_Table)
- NTFS – charakteristika, MFT (Master File Table), run list  
<http://www.digit-life.com/articles/ntfs/>  
<http://www.pcguides.com/ref/hdd/file/ntfs/archSector-c.html>
- ext2/ext3 – struktura, inode, žurnál  
<http://www.science.unitn.it/~fiorella/guidelinux/tlk/node95.html>  
<http://www.linux-security.cn/ebooks/ulk3-html/0596005652/understandlk-CHP-18.html>

## Plánování pohybu hlav disků

- FCFS (First-Come, First-Served) - žádné plánování, fronta požadavků, jeden za druhým
- SSTF (Shortest Seek Time First) - krajní žádosti mohou "hladovět"
- LOOK (výtah), C-LOOK (circular LOOK) - pohyb jen jedním směrem, na konci otočka

## RAID (Redundant Array of Inexpensive Disks)

- JBOD (Just a Bunch of Disks)
- RAID 0 – striping, žádná redundance
- RAID 1 – mirroring, redundance
- RAID 0+1 – mirroring a striping
- RAID 2 – 7-bitový paritní Hammingův kód
- RAID 3 – 1 paritní disk, po bitech na disky
- RAID 4 – 1 paritní disk a striping
- RAID 5 – distribuovaná parita a striping
- RAID 6 – distribuovaná parita – dvojitá P+Q, striping

## 5.13 Bezpečnost, autentifikace, autorizace, přístupová práva

### Definice

- **Ochrana** – prostředky OS mohou pracovat pouze autorizované procesy
- **Autorizace** – zjištění oprávněnosti požadavku
- **Bezpečnost** – zabraňuje neautorizovaný přístup do systému
- **Právo** – povolení/zakázání vykonávat nějakou operaci
- **Doména ochrany** – množina párů (objekt:práva)
  - **ACL (Access Control List)** – ke každému objektu seznam práv pro uživatele/skupiny
  - **C-list (Capability List)** – ke každému uživateli/skupině seznam práv pro objekty

### Autentifikace

Identifikace něčím, co uživatel ví, má nebo je.

- **Hesla**
  - slovníkový útok (80–90% hesel je jednoduchých), hrubá síla
  - vynucování délky a složitosti hesla
- **Model otázka/odpověď**
- **Fyzický objekt** – smartcards, USB klíče
- **Biometrika** – otisky prstů, rohovka, hlas

TODO: autorizace, přístupová práva

## 5.14 Druhy útoků a obrana proti nim

### Vnitřní útoky

- **Trojský kůň** – zdánlivě neškodný program obsahuje „zlý“ kód
- **Login spoofing** – falešná „logovací“ obrazovka
- **Logická bomba** – zaměstnanec vpraví kus kódu do systému, který musí být pravidelně informován o tom, že zaměstnanec je stále zaměstnancem
- **Zadní dvířka (trap door, back door)** – kód při nějaké podmínce přeskočí normální kontroly
- **Přetečení vyrovnávací paměti (buffer overflow)**
  - ve velkém množství kódu nejsou dělány kontroly na přetečení polí pevné velikosti
  - při přetečení se typicky přepíše část zásobníku a lze tam umístit adresu kódu i samotný kód, který se vykoná při návratu z funkce

## Vnější útoky

- **Virus** – vytvoří se nakažený „žádaný“ soubor
- **Internetový červ (worm)** – samoreplikující se program (červ), využívá nějaké chyby systému
- **Mobilní kód** – applety, agenti...

## Útočníci

Útočníkem může být buď náhodný uživatel, vnitřní pracovník, zločinec (zvenčí) nebo špión (vojenský, komerční). Cíle útoků jsou na důvěrnost – zjištění obsahu, nebo celistvost – změna obsahu, případně dostupnost služby – Denial of service. Ke ztrátě dat může dojít i v důsledku chyby hardware, software, lidské chyby nebo Božího zásahu.

## Obrana

jsou to spíš banality, ale nic víc po nás nechtějí???

- proti trojanům, backdoorům, logical bomb – omezení přístupových práv, metoda „least privilege“
- proti login-spoofu – „secure attention key“, tj. takové to „Začněte stisknutím Ctrl-Alt-Del“
- proti buffer overflow – jediné patche
- proti virům – antivirus ;-), anti-spyware
- proti červům – firewall, patche (útoky jsou většinou proti známým a opraveným chybám aplikací, proti druhému typu, tzv. „zero-day attack“ je jedinou obranou firewall)
- proti problémům s applety a skripty – sandboxing (běh v omezeném prostředí bez možnosti přístupu k počítači)
- proti všemu – backupy ;-)

## 5.15 Kryptografické algoritmy a protokoly

### Cíle kryptografie

- důvěrnost dat
- celistvost dat
- autentifikace – od koho jsou data
- nepopíratelnost – když jednou něco potvrdím, nemohu to popřít.

**Definice** (*Kryptografický systém*)

**Kryptografický systém** obsahuje:

- prostor zpráv – *plaintext*,
- prostor šifrovaných zpráv – *ciphertext*,
- prostory šifrovacích a dešifrovacích *klíčů*,

- efektivní algoritmus pro *generování klíčů*,
- efektivní algoritmus pro *šifrování*,
- efektivní algoritmus pro *dešifrování*.

### Definice (označení)

$C$  – šifra,  $P$  – otevřený text,  $K$  – klíč,

$E$  – šifrovací algoritmus,  $D$  – dešifrovací algoritmus.

*Šifrování*:  $C = E(P)$ , resp.  $C = E(K, P)$

*Dešifrování*:  $P = D(C)$ , resp.  $P = D(K, C)$

### Kerchoffovy principy dobrého krypt. systému

- $E$  a  $D$  neobs. tajnou část
- $E$  distribuuje rozumné zprávy rovnoměrně po  $C$
- se správným klíčem jsou  $E$  &  $D$  efektivní
- bez správného klíče je dešifrování minimálně NP-úplné.

### dělení kryptografických systémů

- symetrické krypt. systémy :  $k = k'$
- asymetrické :  $k \neq k'$  ( veřejný a tajný klíč ).

### Model útočníka podle Doleva a Yao

- může získat jakoukoliv zprávu jdoucí po síti, může zahájit komunikaci s jiným uživatelem, může se stát příjemcem zpráv od kohokoliv, může zasílat zprávy komukoliv & vydávat se za jiného uživatele,
- nemůže uhádnout náh. číslo z dost velké množiny, bez klíče nemůže dešifrovat zprávu & nemůže vytvořit platnou šifrovanou zprávu (vzhledem k šifr. alg.).

### Kryptografické protokoly

- **Arbitrované protokoly** – rozhodčí dělá skoro všechno.
- **Rozhodované protokoly** – rozhodčí je dobrý jenom při sporu aby rozhodl.
- **Samozabezpečovací protokoly** – není žádná třetí strana.

### Anonymní platby

Problém kreditních karet spočívá v sledovatelnosti toku peněz. Hledáme protokol pro tvorbu autentizovaných ale nesledovatelných zpráv.

## Časové známky

Nejjednodušší metodou je zasílat kopie zpráv důvěryhodnému arbitrovi, problémy s množstvím uchovávaných dat lze vyřešit použitím hašovacích funkcí.

Používají se spojené (linked) aby odesílatel spolu s arbitrem nemohli podvádět.

1. Odesílatel  $S$  zašle arbitrovi  $A$  hashkod zprávy  $H_n$ .
2.  $A$  vrátí odesílateli  $T_n = S_K(n, S, H_n, Tm_n; Id_{n-1}, H_{n-1}, T_{n-1}, H(Id_{n-1}, H_{n-1}, T_{n-1}))$  kde  $n$  je pořadí zprávy,  $Tm_n$  čas podpisu zprávy,  $Id_{n-1} \dots$  jsou informace o předešlé zprávě, kterou arbitr vyřizoval.
3. Po vyřízení následující zprávy arbitr zašle odesílateli identifikaci následujícího odesílatele

Chce-li někdo ověřit časovou známku zprávy, kontaktuje odesílatele  $Id_{n-1}$  a  $Id_{n+1}$  a pomocí nich ověří platnost  $T_n$

## Digitální podpisy

Musí být nefalšovatelné, autentické, neměnitelné, „nerecyklovatelné“.

**Symetrické systémy:** Nechť odesílatel  $S$  zasílá příjemci  $R$  zprávu  $M$

1.  $S$  zašle arbitrovi  $A$  zprávu  $\mathbf{E}(M, K_S)$ .
2. Arbitr verifikuje odesílatele a příjemci  $R$  zašle  $\mathbf{E}((M, S, \mathbf{E}(M, K_S)), K_R)$
3. Příjemce uchová  $M$  a  $\mathbf{E}(M, K_S)$  pro účely případného dokazování přijetí.

**Asymetrické systémy:** Stačí provést  $\mathbf{E}(\mathbf{D}(M, K_S), K_R)$

## Důkazy s nulovou znalostí

- dokazovatel nesmí podvádět - pokud důkaz nezná, jeho šance přesvědčit arbitra je mizivá
- ověřovatel nesmí podvádět - o důkazu smí zjistit jenom to, ze ho dokazovatel zná. V žádném případě nesmí být schopen důkaz zrekonstruovat a sám provést.
- ověřovatel se nesmí dozvědět nic, co by nebyl schopen zjistit bez pomoci dokazovatele.

Není-li splněna poslední podmínka mluvíme o *důkazech s minimálním vyzrazením*. Jeden z možných důkazů je založen na problematice Hamiltonovských kružnic v grafu.

1. Nechť  $A$  zná Hamiltonovskou kružnici v grafu  $G$ .
2.  $A$  provede náhodnou permutaci očíslování vrcholů  $G$ . Původní graf a vzniklý  $H$  jsou izomorfní.
3. Kopie grafu  $H$  je zaslána entitě  $B$ .
4. Ověřovatel  $B$  položí dokazovateli  $A$  jednu z následujících otázek
  - (a) Dokázat, že  $G$  a  $H$  jsou izomorfní
  - (b) Ukázat Hamiltonovskou kružnici v grafu  $H$
5. Opakováním kroku 1. až 4. lze docílit potřebné jistoty.

### Neurčitý obnos (Oblivious transfer)

Protokol umožňuje, aby si adresát vybral z několika nabízených možností aniž by odesílatel předem znal jeho volbu, možné doplnění o následnou vzájemnou kontrolu.

### Podepisování kontraktů (Contract signing)

V každém okamžiku musí být obě smluvní strany vázány stejně moc. Nejjednodušším řešením je arbitrovaný protokol, kde obě strany předají centrální autoritě své podepsané kopie a tato třetí strana zajistí výměnu po obdržení obou kopií.

### Elektronická potvrzovaná pošta (digital certified mail)

Chceme, aby adresát mohl přečíst naši zprávu až poté, co získáme potvrzení o tom, že ji obdržel (elektronický doporučený dopis).

### Bezpečné volby

- volit smí pouze oprávnění voliči,
- každý smí hlasovat nejvýše jednou,
- nikdo nesmí vědět, kdo jak volil,
- nikdo nesmí měnit volbu jiných,
- každý hlas musí být započítán.

Nejjednodušší možnost je použít protokol se dvěma centrálními autoritami. Používá registrační autoritu  $RA$  provádějící registraci voličů a sčítací autoritu  $SA$ , která sčítá hlasovací lístky a zveřejňuje výsledky voleb.

1. Všichni voliči zašlou  $RA$  žádost o validační číslo.
2.  $RA$  zašle každému voliči náhodně zvolené validační číslo  $L$  a zároveň si poznamená kdo jaké číslo dostal.
3.  $RA$  zašle seznam validačních čísel  $SA$ .
4. Každý z voličů si náhodně vybere svoje identifikační číslo  $Id$  a  $SA$  zašle zprávu  $(L, Id, v)$  kde  $v$  je jeho volba.
5.  $SA$  porovná  $L$  se seznamem validačních čísel z kroku 3. Odpovídající číslo škrtně a voličovo  $Id$  přidá do seznamu asociovaného s voleným kandidátem.
6. Po skončení voleb  $SA$  zveřejní výsledky a seznamy identifikačních čísel spojené se jmény kandidátů.

### Útoky na protokoly

- *přehraní zpráv* –  $M$  odposlouchá všechny zprávy a pak totéž udělá sám
- *muž uprostřed* (man-in-the-middle)
- *paralelní spojení* – několik běhů protokolů prováděných současně pod řízením  $M$
- *odražení* –  $A$  zahájí komunikaci,  $M$  zachytí zprávu, upraví ji, aby nebyl poznat původní  $A$  a pošle ji zpět  $A$
- *prokládání* – Několik běhů protokolu prováděných současně pod řízením  $M$ , zprávy z jednoho se použijí u dalšího, atd.

- *chyba typu* – Nedodržení přesného sémantického významu zprávy
- *vypuštění jména* – Pokud v protokolu není poznat, kdo za to může
- *chybné použití šifrovací služby* – Špatný algoritmus použitý na nevhodném místě

## Kryptografické algoritmy

### Definice (*Substitution-box – S-box*)

- krabička která z  $m$  bitů vstupu dělá  $n$  bitů výstupu.
- někdy je použita pevná tabulka. Např. u DES
- někdy je výstup s-boxu závislý na klíči. Např. u Blowfish, Twofish
- v blokových šifrách je to často s-box kdo zamlžuje vztah mezi plaintextem a šifrou.
- dost často na něm závisí jak je šifra napadnutelná  $\Rightarrow$  musí se volit dost obezřetně

### Symetrické

- vysoká datová propustnost
- klíče na obou koncích musí zůstat utajeny  $\Rightarrow$  je třeba často měnit klíče
- potřeba ověřené TTP (Trusted Third Party)

### DES

Vyvinula firma IBM na zakázku NBS počátkem 70. let. Původní název DEA, v USA DEA1. Jako standard přijat 23. 11. 1976 Dodnes používán v komerční sféře, pro vojenské účely není certifikován ani pro ochranu neklasifikovaných informací. Patrně nejrozsáhleji používaný šifrovací algoritmus všech dob.

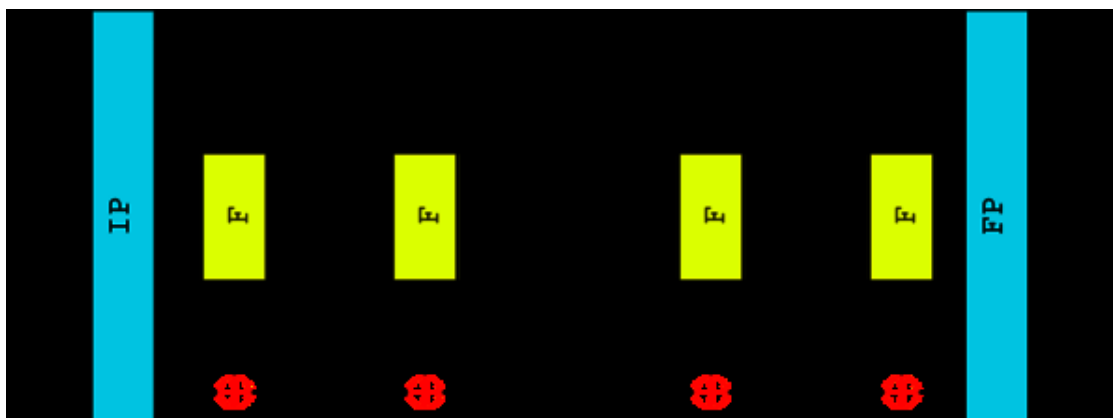
Šifruje 64-bitové bloky otevřeného textu na 64-bitové výstupní bloky, délka klíče 64 bitů.

### Analýza:

- velká slabina je 64-bitový klíč (navíc efektivně pouze 56-bitový). Prolomen za méně než 24 hodin.
- úvodní permutace nemá prakticky žádný vliv
- existence slabých ( $\mathbf{E}(K) = \mathbf{D}(K)$ ) a poloslabých ( $\mathbf{E}(K_1)\mathbf{E}(K_2) = Id.$ ) klíčů
- komplementárnost  $C = \mathbf{E}(K, P) \Leftrightarrow \neg C = \neg \mathbf{E}(\neg K, \neg P)$

### Blowfish

- nástupce systému DES,
- opět Feistelova šifra, délka bloku je 64 bitů, proměnná délka klíče až 448 bitů
- algoritmus provádí 16 cyklů nad vstupem délky 64-bitů



Obr. 5: Struktura hlavní sítě algoritmu DES (zdroj: Wikipedie)

## IDEA

- z roku 1991, vyšel pod názvem IPES.
- IDEA (International Data Encryption Algorithm)
- bloková šifra s délkou bloku 64-bitů a délkou klíče 128-bitu
- algoritmus je patentován
- zajímavé je že pokud bychom algoritmus upravili tak, že bychom všechny řetězce se kterými pracuje zvětšili na dvojnásobek, tak dojde ke ztrátě bezpečnosti.
- algoritmus je považován za bezpečný.

## RC5

- z roku 1994 od R. Rivesta
- používá rotace závislé na datech.
- algoritmus umožňuje nastavit spoustu parametrů:
  - délka šifrovacího klíče (0...255 bajtů)
  - počet kol šifrovacího procesu (0...255)
  - z hodnot 16, 32, 64, ale i vyšších lze zvolit délku slova, algoritmus zpracovává bloky o délce dvojnásobku slova

## Kryptosystém Rijndael

- produkční bloková šifra
- proměnná délka bloku – 16, 24 nebo 32 bajtů
- proměnná délka klíče – 128, 192 nebo 256 bitů

**Analýza:** Po rozsáhlé analýze nenalezena žádná slabina a tak zvolen jako nový standard AES.

## RC4

- proudová šifra od R. Rivesta
- jednoduchý a rychlý algoritmus

**Analýza:** Zatím není známý žádný způsob útoku  $\Rightarrow$  algoritmus považován za bezpečný.

## FISH

- proudová šifra založena na Fibonacciho generátoru pseudonáhodných čísel.
- z fibonacciho generátoru se získá posloupnost a šifrování se provádí například XORováním této posloupnosti s  $P$

## Asymetrické

- šifry s asymetrickým klíčem – RSA, DSA (ElGammal)
- mnohem pomalejší
- není potřeba TTP
- pouze jeden klíč tajný, nemusí se měnit tak často
- o žádném schématu veřejného klíče nebylo dokázáno, že je bezpečné

## RSA

Kryptoschéma je založeno na Eulerově formuli:

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

kde  $\varphi(n)$  je počet čísel z intervalu  $1..n$  která jsou s  $n$  nesoudělná.

**Šifrování:** Je třeba znát číslo  $n$  a malé prvočíslo  $e$ . Otevřený text převedeme do posloupnosti modulo  $n$ . Každý blok  $P_j$  zašifrujeme dle vzorce:

$$C_j \equiv P_j^e \pmod{n}$$

Spojením výsledných bloků vznikne zašifrovaný text.

**Dešifrování:** Je třeba znát číslo  $n$  a číslo  $d$ . Každý z bloků potom dešifrujeme takto:

$$P_j \equiv C_j^d \pmod{n}$$

Pro dešifrovací klíč  $d$  musí platit:

$$ed \equiv 1 \pmod{\varphi(n)}$$

Prvočíslo  $e$  nesmí dělit  $\varphi(n)$ .  $d$  určíme z předchozího vztahu rozšířeným eukleidovým algoritmem.

Veřejný klíč tvoří pár  $(n, e)$ , soukromý klíč pár  $(n, d)$ . Číslo  $n$  musí být velmi velké a nesmí mít malé faktory. Pro reálné použití 100 až 200 bitů. Hranice bezpečnosti 1024 bitů modulu  $n$ , rozumné 1500 bitů, lépe 2048 bitů.

Není známa žádná metoda vedoucí k rozbití algoritmu RSA.

## Merkle-Hellman kryptosystém

- založen na problému batohu
- plaintext je chápán jako posloupnost vah (řešení)
- ciphertext je výsledná hmotnost batohu
- pro superrostoucí posloupnost je problém řešitelný v lineárním čase
- superrostoucí posloupnost je součástí soukromého klíče a tak dešifrování pomocí ní je zvládnutelné lineárně, kdežto bez ní je to NP-úplný problém
- systém byl prolomen! Není tedy považován za bezpečný. Útočník je schopen získat superrostoucí posloupnost a pomocí ní může dešifrovat

## Elgamal kryptosystém

Založen na obtížnosti výpočtu diskrétního logaritmu nad kruhem.

Potřebujeme společný modul  $q$  a číslo  $g$  co nejvyššího řádu. Každý účastník si zvolí tajný klíč  $y_i$  a vypočítá veřejný klíč  $g^{y_i} \bmod q$ .

**Šifrování:** Nechť uživatel  $A$  posílá zprávu  $P$  uživateli  $B$ . Náhodně vybere číslo  $k$  a vypočítá:

$$g^k \bmod q; P \otimes (g^{y_b})^k \bmod q$$

obě čísla zašle  $B$ .

**Dešifrování:** Uživatel  $B$  vypočítá:

$$(g^k)^{y_b} \bmod q$$

a najde inverzní prvek. Z druhého čísla potom snadno získá  $P$ .

Systém je považován za bezpečný. Nevýhodou je nutnost generovat náhodné číslo  $k$  a zdvojnásobení dat během šifrování.

## 6 Síť a internetové technologie

### Požadavky

- Architektura ISO/OSI
- Rodina protokolu TCP/IP (ARP, IPv4, IPv6, ICMP, UDP, TCP) - adresace, routing, fragmentace, spolehlivost, flow control, congestion control, NAT
- Rozhraní BSD sockets
- Spolehlivost - spojované a nespojované protokoly, typy, detekce a oprava chyb
- Bezpečnost - IPSec, principy fungování AH, ESP, transport mode, tunnel mode, firewalls
- Internetové a intranetové protokoly a technologie - DNS, SMTP, FTP, HTTP, NFS, HTML, XML, XSLT a jejich použití.

## 6.1 Architektura ISO/OSI

### Úvod

#### Definice

**Síťový model** je ucelená představa o tom, jak mají být sítě řešeny (obsahuje: počet vrstev, co má která vrstva na starosti; neobsahuje: konkrétní představu jak která vrstva plní své úkoly - tedy konkrétní protokoly). Příkladem je *referenční model ISO/OSI* (konkrétní protokoly vznikaly samostatně a dodatečně).

**Síťová architektura** navíc obsahuje konkrétní protokoly - napr. *rodina protokolů TCP/IP*.

Referenčný model ISO/OSI (International Standards Organization / Open Systems Interconnection) bol pokusom vytvoriť univerzálnu sieťovú architektúru - ale skončil ako sieťový model (bez protokolov). Pochádza zo „sveta spojov“ - organizácie ISO, a bol „oficiálnym riešením“, presadzovaným „orgánmi štátu“; dnes už prakticky odpísaný - prehral v súboji s TCP/IP. ISO/OSI bol reakciou na vznik proprietárnych a uzavretých sietí. Pôvodne mal model popisovať chovanie otvorených systémov vo vnútri aj medzi sebou, ale bolo od toho upustené a nakoniec z modelu ostal len sieťový model (popis funkcionality vrstiev) a konkrétne protokoly pre RM ISO/OSI boli vyvíjané samostatne (a dodatočne zaraďované do rámca ISO/OSI).

Model vznikol maximalistickým spôsobom - obsahoval všetko čo by mohlo byť v budúcnosti potrebné. Vďaka rozsiahlosti štandardu sa implementovali len jeho niektoré podmnožiny - ktoré neboli (vždy) kompatibilné. Vznikol GOSIP (Government OSI Profile) určujúci podmnožinu modelu, ktorú malo mať implementované všetko štátne sieťové vybavenie. Naproti tomu všetkému TCP/IP vzniklo naopak - najprv navrhnutím jednoduchého riešenia, potom postupným obohacovaním o nové vlastnosti (tie boli zahrnuté až po preukázaní „životaschopnosti“).

### 7 vrstev

Kritériá pri návrhu vrstiev boli napr.: rovnomerná vyťaženosť vrstiev, čo najmenšie dátové toky medzi vrstvami, možnosť prevziať už existujúce štandardy (X.25), odlišné funkcie mali patriť do odlišných vrstiev, funkcie na rovnakom stupni abstrakcie mali patriť do rovnakej vrstvy. Niektoré vrstvy z finálneho návrhu sa používajú málo (relačná a prezentačná), niektoré zase príliš (linková - rozpadla sa na 2 podvrstvy LLC+MAC).

aplikační vrstva prezentační vrstva relační vrstva	vrstvy orientované na podporu aplikací
transportní vrstva	příspěvobovací vrstva
síťová vrstva linková vrstva fyzická vrstva	vrstvy orientované na přenos dat

**Fyzická vrstva** sa zaoberá prenosom bitov (kódovanie, modulácia, synchronizácia...) a ponúka teda služby typu pošli a príjmi bit (pričom neinterpretuje význam

týchto dát). Pracuje sa tu s veličinami ako je *šírka pásma, modulačná a prenosová rýchlosť*.

**Linková vrstva** prenáša vždy celé bloky dát (rámce/frames), používa pritom fyzickú vrstvu a prenos vždy funguje len k priamym susedom. Môže pracovať spoľahlivo či nespoľahlivo, prípadne poskytovať QoS/best effort. Ďalej zabezpečuje riadenie toku - zaistenie toho, aby vysielajúci nezahltil príjemcu. Delí sa na dve podvrstvy - MAC (prístup k zdieľanému médiu - rieši konflikty pri viacnásobnom prístupe k médiu) a LLC (ostatné úlohy).

**Sieťová vrstva** prenáša pakety (packets) - fakticky ich vkladá do linkových rámcov. Zaručuje doručenie paketov až ku konečnému adresátovi (tj. zabezpečuje smerovanie). Môže používať rôzne algoritmy smerovania - ne/adaptívne, izolované, distribuované, centralizované... (v architektúre TCP/IP je to IP vrstva)

**Transportná vrstva** zabezpečuje komunikáciu medzi koncovými účastníkmi (end-to-end) a môže meniť nespoľahlivý charakter komunikácie na spoľahlivý, menej spoľahlivý na viac spoľahlivý, nespojovaný prenos na spojovaný... Príkladom sú napr. TCP a UDP. Ďalšou úlohou je rozlišovanie jednotlivých entít (na rozdiel od napr. sieťovej vrstvy) v rámci uzlov - procesy, demony, úlohy (rozlišuje sa zväčša nepriamo - napr. v TCP/IP pomocou portov).

**Relačná vrstva** zaisťuje vedenie relácií - šifrovanie, synchronizáciu, podporu transakcií. Je to najkritizovanejšia vrstva v ISO/OSI modeli, v TCP/IP úplne chýba.

**Prezentačná vrstva** slúži na konverziu dát, aby obe strany interpretovali dáta rovnako (napr. reálne čísla, rôzne kódovanie textov). Ďalej má na starosti konverziu dát do formátu, ktorý je možné preniesť: napr. linearizácia viacrozmerných polí, dátových štruktúr; konverzia viacbajtových položiek na jednotlivé byty (little vs. big endian). *Poznámka:* Zápis čísla 1234H v Big endian je [12:34:-:-] (sun, motorola), v Little endian [-:-:34:12] (intel, amd, ethernet).

**Aplikačná vrstva** mala pôvodne obsahovať aplikácie - ale tých je veľa a nebolo možné ich štandardizovať. Teraz teda obsahuje len „jadro“ aplikácií - tie, ktoré malo zmysel štandardizovať (email a pod.). Ostatné časti aplikácií (GUI) boli vysunuté nad aplikačnú vrstvu.

## Kritika

Model ISO/OSI:

- je príliš zložitý, ťažkopádny a obtiažne implementovateľný
- je príliš maximalistický
- nerešpektuje požiadavky a realitu bežnej praxe
- počítal skôr s rozľahlými sieťami ako s lokálnymi
- niektoré činnosti (funkcie) zbytočne opakuje na každej vrstve
- jednoznačne uprednostňuje spoľahlivé a spojované prenosové služby (ale tie sú spojené s veľkou réžiou ⇒ spoľahlivosť si efektívnejšie zabezpečia koncové uzly)

Možnosť nespoľahlivého/nespojovaného spojenia bolo pridané do štandardu až dodatočne, napriek tomu bol porazený architektúrou TCP/IP. Používajú sa však

niektoré prevzaté prokoly - X.400 (elektronická pošta), X.500 (adresárové služby - odľahčením vznikol úspešný protokol LDAP).

## 6.2 Rodina protokolů TCP/IP (ARP, IPv4, IPv6, ICMP, UDP, TCP) – adresace, routing, fragmentace, spolehlivost, flow control, congestion control, NAT

ISO/OSI	TCP/IP
aplikační vrstva prezentační vrstva relační vrstva	aplikační vrstva
transportní vrstva	transportní vrstva
síťová vrstva	síťová vrstva (též IP vrstva)
linková vrstva fyzická vrstva	vrstva síťového rozhraní

Obvyklé označenie je *TCP/IP protocol suite* (súčasťou je viac ako 100 protokolov). Architektúra vznikla postupne (v akademickom prostredí, neskôr sa rozšírila aj do komerčnej sféry) – najprv vznikli protokoly, potom vrstvy – a od vzniku sa toho zmenilo len málo (zmeny sú aditívne). Je to najpoužívanejšia sieťová technológia (IP over everything, everything over IP). Prístup autorov bol, na rozdiel od ISO/OSI, od jednoduchšieho k zložitejšiemu – najprv sa vytvárajú jednoduché riešenia, ktoré sa postupne obohacujú. Až sa riešenie prakticky overí (2 nezávislé implementácie), vznikne štandard. TCP/IP predpokladá že siete sú typu nespojované, nespoľahlivé a best effort. Všetka inteligencia je sústredená do koncových uzlov, sieť je „hlúpa“ ale rýchla.

TCP/IP bol pôvodne určený pre ARPAnet – nemohol mať teda žiadnu centrálnu časť a musel byť robustný voči chybám (nespoľahlivé/nespojované prenosy). Dôraz sa kládol aj na "internetworking". Nebolo však požadované zabezpečenie, mobilita ani kvalita služieb.

TCP/IP nedefinuje rôzne siete (čo sa hardvérových vlastností týka) a technológie vo vrstve sieťového rozhrania – iba sa snaží nad nimi prevádzkovať protokol IP (okrem SLIP a PPP pre dvojbodové spoje). V sieťovej vrstve je IP protokol, v transportnej jednotné transportné protokoly (TCP a UDP), v aplikačnej potom jednotné základy aplikácií (email, prenos súborov, remote login...).

### Adresace, IPv4, IPv6

Data se v IP síti posílají po blocích nazývaných datagramy. Jednotlivé datagramy putují sítí zcela nezávisle, na začátku komunikace není potřeba navazovat spojení či jinak „připravovat cestu“ datům, přestože spolu třeba příslušné stroje nikdy předtím nekomunikovaly.

IP protokol v doručování datagramů poskytuje nespolehlivou službu, označuje se také jako best effort – „nejlepší úsilí“; tj. všechny stroje na trase se datagram snaží podle svých možností poslat blíže k cíli, ale nezaručují prakticky nic. Datagram vůbec nemusí dorazit, může být naopak doručen několikrát a neručí se ani za

pořadí doručených paketů. Pokud aplikace potřebuje spolehlivost, je potřeba ji implementovat v jiné vrstvě síťové architektury, typicky protokoly bezprostředně nad IP (viz TCP).

Pokud by síť často ztrácela pakety, měnila jejich pořadí nebo je poškozovala, výkon sítě pozorovaný uživatelem by byl malý. Na druhou stranu příležitostná chyba nemívá pozorovatelný efekt, navíc se obvykle používá vyšší vrstva, která ji automaticky opraví.

V **IPv4** je *adresou* 32bitové číslo, zapisované po jednotlivých bajtech, oddělených tečkami. Takových čísel existuje celkem  $2^{32}$ . Určitá část adres je ovšem rezervována pro vnitřní potřeby protokolu a nemohou být přiděleny. Dále pak praktické důvody vedou k tomu, že adresy je nutno přidělovat hierarchicky, takže celý adresní prostor není možné využít beze zbytku. To vede k tomu, že v současnosti je již znatelný nedostatek IP adres, který řeší různými způsoby: dynamickým přidělováním (tzn. např. každý uživatel dial-up připojení dostane dočasnou IP adresu ve chvíli, kdy se připojí, ale jakmile se odpojí, je jeho IP adresa přidělena někomu jinému; při příštím připojení pak může tentýž uživatel dostat úplně jinou adresu), překladem adres (NAT) a podobně. Ke správě tohoto přidělování slouží specializované síťové protokoly, jako např. DHCP.

Původný koncept adres počítal so štruktúrou adresy IPv4 v tvare *sieť:počítač*, kde bolo delenie častí pevne dané. Neskôr sa to ale ukázalo ako príliš hrubé delenie a lokálna časť adresy (v rámci jednej podsiete) môže mať dnes promenlivú dĺžku. Obecne platí, že medzi adresami v rovnakej podsieti (majú rovnakú sieťovú časť) je možné dopravovať dáta priamo – dotyční účastníci sú prepojení jedným ethernetom alebo inou lokálnou sieťou. V opačnom prípade sa dáta dopravujú *smerničkami/routermi*. Hranicu v adrese medzi adresou siete a počítača určuje dnes maska podsiete. Jedná sa o 32 bitovú hodnotu, ktorá obsahuje jednotky tam, kde je v adrese určená sieť.

**Adresovanie sietí** bolo v prvopočiatkoch internetu vyriešené staticky – prvých 8 bitov adresy určovalo sieť, zvyšok jednotlivé počítače (existovať tak mohlo max. 256 sietí). S nástupom lokálnych sietí bolo tento systém potrebné zmeniť – zaviedli sa *triedy IP adres*. Existovalo 5 tried (A(začiatok 0, hodnoty prvého bajtu 0-127, maska 255.0.0.0), B(10, 128-191, 255.255.0.0), C(110, 192-223, 255.255.255.0), D(1110, 224-239, určené na multicast) a E(1111, 240-255, určené ako rezerva)). Postupom času sa ale aj toto rozdelenie ukázalo ako nepružné a bol zavedený CIDR (Classless Inter-Domain Routing) systém v ktorom je možné hranicu medzi adresou siete a lokálnou časťou adresy umiestniť ľubovoľne (označuje sa potom ako kombinácia prefixu a dĺžky vo forme 192.168.0.0/24, kde 24 znamená že adresu tvorí prvých 24 bitov – jiný zápis je pomocí už zmiňované masky podsítě, tj. 192.168.0.0 s maskou 255.255.255.0).

Medzi adresami existujú niektoré tzv. **vyhradené adresy**, ktoré majú špeciálny význam.

- Adresa s (binárnymi) nulami v časti určujúcej počítač (192.168.0.0 (/24)) znamená „táto sieť“, resp. „táto stanica“.
- Adresa s jednotkami v časti určujúcej počítač (192.168.0.255 (/24)) znamená broadcast – všesmerové vysielanie.
- Adresy 10.0.0.0 – 10.255.255.255, 172.16.0.0 – 172.31.255.255 a 192.168.0.0 –

192.168.255.255 sa používajú na adresovanie interných sietí – smerovače tieto adresy nesmie smerovať ďalej do internetu.

**IPv6** je trvalejším riešením nedostatku adries – zatiaľ sa ale rozširuje veľmi pozvoľna. Adresa v IPv6 má dĺžku 128 bitov (oproti 32), čo znamená cca.  $6 \times 10^{23}$  IP adries na  $1m^2$  zemského povrchu – umožňuje teda, aby každé zariadenie na zemi malo vlastnú jednoznačnú adresu. Adresa IPv6 sa zapisuje ako osem skupín po štyroch hexadecimálnych číslach (napr. 2001:0718:1c01:0016:0214:22ff:fec9:0ca5) – pričom úvodné nuly v číslach je možné vynechať. Ak po sebe nasleduje niekoľko nulových skupín, je možné použiť len znaky :: – napr. ::1 miesto 0000:0000:.....:0001. Toto je možné použiť len raz v zápise adresy. RFC 4291 zavádza 3 typy adries:

- **individuálne / unicast** – identifikujú práve jedno rozhranie
- **skupinové / multicast** – určuje skupinu zariadení, ktorým sa má správa dopraviť
- **výberové / anycast** – určuje tiež skupinu zariadení, dáta sa však doručia len jednému z členov (najbližšiemu)

IPv6 neobsahuje všesmerové (broadcast) adresy. Byly nahrazeny obecnějším modelem skupinových adres a pro potřeby doručení dat všem zařízením připojeným k určité síti slouží speciální skupinové adresy (např. ff02::1 označuje všechny uzly na dané lince).

IPv6 zavádí také koncepci dosahu (scope) adres. Adresa je jednoznačná vždy jen v rámci svého dosahu. Nejčastější dosah je pochopitelně globální, kdy adresa je jednoznačná v celém Internetu. Kromě toho se často používá dosah linkový, definující jednoznačnou adresu v rámci jedné linky (lokální síť, např. Ethernetu). Propracovanou strukturu dosahů mají skupinové adresy (viz níže).

Adresní prostor je rozdělen následovně:

prefix	význam
::/128	neurčená
::1/128	smyčka (loopback)
ff00::/8	skupinové
fe80::/10	individuální lokální linkové
ostatní	individuální globální

Výběrové adresy nemají rezervovanu svou vlastní část adresního prostoru. Jsou promíchány s individuálními a je otázkou lokální konfigurace, aby uzel poznal, zda se jedná o individuální či výběrovou adresu.

Strukturu globálních individuálních IPv6 adres definuje RFC 3587. Je velmi jednoduchá a de facto odpovídá (až na rozměry jednotlivých částí) výše uvedené struktúře IPv4 adresy.

n bitů	64-n bitů	64 bitů
globální směrovací prefix	adresa podsítě	adresa rozhraní

Globální směrovací prefix je de facto totéž co adresa sítě, následuje adresa podsítě a počítače (přesněji síťového rozhraní). V praxi je adresa podsítě až na výjimky 16bitová a globální prefix 48bitový. Ten je pak přidělován obvyklou hierarchií, jejíž stávající pravidla jsou:

- první dva bajty obsahují hodnotu 2001 (psáno v šestnáctkové soustavě)
- další dva bajty přiděluje regionální registrátor (RIR)
- další dva bajty přiděluje lokální registrátor (LIR)

Reálná struktura globální individuální adresy tedy vypadá následovně:

16 bitů 2001	16 bitů přiděluje RIR	16 bitů přiděluje LIR	16 bitů adresa podsítě	64 bitů adresa rozhraní
-----------------	--------------------------	--------------------------	---------------------------	----------------------------

Adresa rozhraní by pak měla obsahovat modifikovaný EUI-64 identifikátor. Ten získáte z MAC adresy jednoduchým postupem: invertuje se druhý bit MAC adresy a doprostřed se vloží dva bajty obsahující hodnotu fffe. Z ethernetové adresy 00:14:22:c9:0c:a5 tak vznikne identifikátor 0214:22ff:fec9:0ca5.

Adresy začínající hodnotou ff sú tzv. "skupinové adresy" – čtyři následující bity v nej obsahují příznaky, další čtyři potom dosah (napr. interface-local, link-local, admin-local, site-local, organization-local, global...)

IPv6 dále podporuje QoS a bezpečnost (IPsec).

## Routing

Pojmem **směrování** (routing, routování) je označováno hledání cest v počítačových sítích. Jeho úkolem je dopravit datový paket určenému adresátovi, pokud možno co nejefektivnější cestou. Síťová infrastruktura mezi odesílatelem a adresátem paketu může být velmi složitá. Směrování se proto zpravidla nezabývá celou cestou paketu, ale řeší vždy jen jeden krok – komu data předat jako dalšímu (tzv. „distribuované směrování“). Ten pak rozhoduje, co s paketem udělat dál.

V prípade, že je cieľová stanica packetu v rovnakej sieti ako je odosielateľ, o doručenie sa postará linková vrstva. V opačnom prípade musí odosielateľ určiť najvhodnejší odchodzí smer a poslať datagram smerovaču vo zvolenom smere.

Základní datovou strukturou pro směrování je směrovací tabulka (routing table). Představuje vlastně onu sadu ukazatelů, podle kterých se rozhoduje, co udělat s kterým paketem. Směrovací tabulka je složena ze záznamů obsahujících:

- cílovou adresu, které se dotýčný záznam týká. Může se jednat o adresu individuálního počítače, častěji však je cíl definován prefixem, tedy začátkem adresy. Prefix mívá podobu 147.230.0.0/16. Hodnota před lomítkem je adresa cíle, hodnota za lomítkem pak určuje počet významných bitů adresy. Uvedenému prefixu tedy vyhovuje každá adresa, která má v počátečních 16 bitech (čili prvních dvou bajtech) hodnotu 147.230.
- akci určující, co provést s datagramy, jejichž adresa vyhovuje prefixu. Akce mohou být dvou typů: doručit přímo adresátovi (pokud je dotýčný stroj s adresátem přímo spojen) nebo předat některému ze sousedů (jestliže je adresát vzdálen).

Směrovací rozhodnutí pak probíhá samostatně pro každý procházející datagram. Vezme se jeho cílová adresa a porovná se směrovací tabulkou následovně:

- Z tabulky se vyberou všechny vyhovující záznamy (jejichž prefix vyhovuje cílové adrese datagramu).

- Z vybraných záznamů se použije ten s nejdelším prefixem. Toto pravidlo vyjadřuje přirozený princip, že konkrétnější záznamy (jejichž prefix je delší, tedy přesnější; speciálním případem je *host-specific route*) mají přednost před obecnějšími (co může být např. i *default route*; ps: *agregace*).

Zajímavou otázkou je, jak vznikne a jak je udržována směrovací tabulka. Tento proces mají obecně na starosti směrovací algoritmy. Když jsou pak pro určitý algoritmus definována přesná pravidla komunikace a formáty zpráv nesoucích směrovací informace, vznikne směrovací protokol (routing protocol). Směrovací algoritmy můžeme rozdělit do dvou základních skupin: na statické a dynamické. Často se také mluví o statickém a dynamickém směrování, které je důsledkem činnosti příslušných protokolů.

Při **statickém (též neadaptivním) směrování** se směrovací tabulka nijak nemění. Je dána konfigurací počítače a případné změny je třeba v ní provést ručně. Tato varianta vypadá jako nepříliš atraktivní, ve skutečnosti ale drtivá většina zařízení v Internetu směřuje staticky.

**Dynamické (adaptivní) směrování** průběžně reaguje na změny v síťové topologii a přizpůsobuje jim směrovací tabulky. Na vytváření tabulek existuje několik algoritmů – routovacích protokolů (vector-distance/link-state) – RIP, BGP, OSPF.

### Distribuované směrování

V distribuovaném směrování může výpočet cesty (směru předání paketu) provádět buď každý uzel nezávisle, nebo mohou uzly kooperovat (distribuovaný výpočet). Rozlišuje se také četnost aktualizace informací. Dva základní algoritmy distribuovaného směrování jsou:

- *vector distance* – každý uzel si udržuje tabulku vzdáleností, přímí sousedé si vyměňují informace o cestách ke všem uzlům, tj. jde o distribuovaný výpočet, přenáší se dost informací. Trpí problémem „count-to-infinity“ – tj. když 1 uzel přestane existovat, postupně si jeho sousedé mezi sebou přehazují vzdálenost, postupně o 1 zvětšovanou (do nekonečna). Řeší se pomocí technik „split horizon“ (neinzeruj vzdálenost zpět) a „poisoned reverse“ (inzeruj zpět nekonečno), někde ale přesto selhává.
- *link state* – každý uzel hledá změny svých sousedů a pokud k nějaké dojde, pošle floodem informaci do celé sítě. Výpočet vzdáleností dělá každý uzel sám.

Tyto algoritmy se používají u některých známých směrovacích protokolů:

- *RIP* (Routing Information Protocol) – protokol z BSD Unixu, typu vector distance. Počítá s max. 16 přeskoky, změny se updatují 2x za minutu. Informace ve směrovací tabulce může zahrnovat max. 25 sítí, používá split horizon & poisoned reverse. Hodí se ale jen pro malé sítě.
- *OSPF* (Open Shortest Path First) – jde o protokol typu link state, uzly si počítají vzdálenosti do všech sítí Dijkstrovým algoritmem. Pro zjišťování změn se posílají pakety „HELLO“ a „ECHO“. Má lepší škálovatelnost, hodí se pro větší sítě.

## Hierarchické směrování, autonomní systémy

Hierarchické směrování znamená rozdělení sítě do oblastí (*areas*) a směrování mezi nimi jen přes vstupní body. Je vhodné pro velké, složitě propojené nebo různým způsobem spravované sítě. Nad oblastmi se vytvoří propojení – *backbone area* (páteřní systém), přes které se směrování mezi oblastmi provádí. Celému tomuto (*areas + backbone area*) se říká *autonomní systém*. Detailní směrovací informace neopouštějí jednotlivé oblasti.

Pro směrování v rámci jedné oblasti i mezi oblastmi v rámci jednoho autonomního systému slouží jeden z tzv. *interior gateway protocols*, může být použit např. OSPF nebo RIP, případně další jako IGRP (*interior gateway routing protocol*, typu *vector distance*) nebo EIGRP (*enhanced IGRP*, hybrid mezi *vector distance* a *link state*). Mezi jednotlivými autonomními systémy (přes *AS boundary routers*) se směruje pomocí *exterior gateway protocolu*, jedním z nich je např. *Border Gateway Protocol* (BGP).

Díky existenci autonomních systémů jde např. při peeringu stanovit, který provoz půjde přes peering a který výše po upstreamu do páteřních sítí.

## Fragmentace

**Maximum transmission unit** (MTU) je maximální velikost paketu, který je možné přenést z jednoho síťového zařízení na druhé. Obvyklá hodnota MTU v případě Ethernetu je cca 1500 bajtů, nicméně mezi některými místy počítačové sítě (spojených například modemem nebo sériovou linkou) může být maximální délka přeneseného paketu nižší. Hodnotu MTU lze zjistit prostřednictvím protokolu ICMP. Při posílání paketů přes několik síťových zařízení je samozřejmě důležité nalézt nejmenší MTU na dané cestě. Hodnota MTU je omezena zdola na 576 bajtů.

U přenosového protokolu TCP je při směrování paketu do přenosového kanálu s nižším MTU než je délka paketu, provedena **fragmentace paketu**. U protokolu UDP není fragmentace paketu podporována a paket je v takovém případě zahozen.

Pokud dorazí na směrovač paket o velikosti větší, než kterou je přenosová trasa schopna přenést (např. při přechodu z Token Ringu používajícího 4 kByte pakety na Ethernet používajícího maximálně 1,5 kByte pakety), musí směrovač zajistit tzv. fragmentaci, neboli rozebrání paketu na menší části a cílový uzel musí zajistit opětovné složení, neboli defragmentaci.

Fragmenty procházejí přes síť jako samostatné datagramy. Aby byl koncový uzel schopen fragmenty složit do originálního datagramu, musí být fragmenty příslušně označeny. Toto označování se provádí v příslušných polích IP hlavičky.

Pokud nesmí být datagram fragmentován, je označen v příslušném místě IP hlavičky příznakem „Don't Fragment“. Jestliže takto označený paket dorazí na směrovač, který by jej měl poslat prostředím s nižším MTU a tudíž je nutnost provést fragmentaci, provede směrovač jeho zrušení a informuje odesílatele chybovou zprávou ICMP.

Aby byl cílový uzel schopen složit originální datagram, musí mít dostatečný buffer do něhož jsou jednotlivé fragmenty ukládány na příslušnou pozici danou offsetem. Složení je dokončeno v okamžiku, kdy je vyplněn celý datagram začínající fragmentem s nulovým offsetem (*identification a fragmentation offset* v hlavičce) a končící

segmentem s příznakem „More Data Flag“ (resp. More Fragments) nastaveným na False.

V IPv4 je možné fragmentované pakety ďalej deliť; naproti tomu v IPv6 musí fragmentáciu zabezpečiť odosielateľ – nevyhovujúce pakety sa zahadzujú.

## Spolehlivost, Flow control, Congestion control

Keďže TCP/IP funguje nad obecné nespojovanými a nespoľahlivými médiami, **spoľahlivosť** ktorú TCP poskytuje nie je „skutočná“, ale len „softvérovo emulovaná“ – medziľahlé uzly o spojení nič nevedia, fungujú nespojovane (pre komunikáciu sa používa sieťová vrstva, transportná „existuje“ iba medzi koncovými uzlami). Je teda nutné ošetriť napr. nespoľahlivosť infraštruktúry (strácanie dát, duplicity – pričom stratiť sa môže aj žiadosť o vytvorenie pripojenia, potvrdenie...) a reboot uzlov (uzol stratí históriu, je potrebné ošetriť existujúce spojenia...).

Používa sa celá rada techník, kde základom je kontinuálne potvrdzovanie: príjemca posiela kladné potvrdenia; odosielateľ po každom odoslaní spúšťa časovač a ak mu do vypršania nepríde potvrdenie, posiela dáta znovu. Potvrdzovanie nie je samostatné ale vkladá sa do paketov cestujúcich opačným smerom – *piggybacking*.

TCP priebežne kontroluje „dobu obrátky“ a vyhodnocuje vážený priemer a rozptyl dôb obrátky. Čakaciu dobu (na potvrdenie) potom vypočítava ako funkciu tohto váženého priemeru a rozptylu. Výsledný efekt je potom ten, že čakacia doba je tesne nad strednou dobou obrátky. V prípade konštantnej doby obrátky sa čakacia doba približuje strednej dobe obrátky; ak kolíše, čakacia doba sa zväčšuje.

Dáta v TCP sa prijímajú/posielajú po jednotlivých byteoch – interne sa však bufferujú a posielajú až po naplnení buffera (pričom aplikácia si môže vyžiadať okamžité odoslanie – operácia PUSH). TCP si potrebuje označovať jednotlivé byty v rámci prúdu (keďže nepracuje s blokmi) – napr. kvôli potvrdzovaniu; používa sa na to 32-bitová pozícia v bytovom prúde (začína sa od náhodne zvoleného čísla).

TCP sa snaží **riadiť tok dát** – aby odosielateľ nezahľcoval príjemcu a kvôli tomu nedochádzalo k strate dát. Podstata riešenia je tzv. *metóda okienka*. Okienko udáva veľkosť voľných bufferov na strane prijímajúceho a odosielateľ môže posielateľ dáta až do „zaplnenia“ okienka. Príjemca spolu s každým potvrdením posiela aj svoju ponuku – údaj o veľkosti okienka (window advertisement), ktorý hovorí koľko ešte dát je schopný prijať (naviac k práve potvrdeným). Znovu – používa sa metóda kontinuálneho potvrdzovania.

Väčšina strát prenášaných dát ide skôr na vrub zahlteniu ako chybám HW a transportné protokoly môžu nevhodným chovaním zhoršovať dôsledky. TCP každú stratu dát chápe ako dôsledok zahltenia – nasadzuje **opatrenia proti zahlteniu** (congestion control). Po strate paketu ho pošle znovu ale neposiela ďalšie a čaká na potvrdenie (tj. prechod z kontinuálneho potvrdzovania na jednotlivé ⇒ vysíla menej dát ako mu umožňuje okienko). Ak príde potvrdenie včas, zdvojnásobí množstvo odosielaných dát – a tak pokračuje kým nenarazí na aktuálnu veľkosť okienka (postupne sa tak vracia na kontinuálne potvrdzovanie).

Dôležitou vlastnosťou je aj korektné chovanie pri naväzovaní a rušení spojenia (v prostredí, kde môže dôjsť k spomaleniu, strate, duplicite...) – používa sa tzv. 3-fázový handshake. Vytvorenie spojenia prebieha nasledovne:

1. Klient pošle serveru SYN paket (v pakete je nastavený příznak SYN) spolu s náhodným *sequence number* (X).
2. Server tento paket přijme, zaznamená si *sequence number* (X) a pošle zpět paket SYN-ACK. Tento paket obsahuje pole Acknowledgement, které označuje další číslo (*sequence number*), které tento host očekává (X+1). Tento host rovno vytvoří spätnú session s vlastným sekvenčným čísлом (Y).
3. Klient odpovie so sekvenčným čísлом (X+1) a jednoduchým Acknowledgement čísлом (Y+1) – čo je sekvenčné číslo servera+1.

Pak už spojení považováno za navázané. Rušení spojení funguje podobne, posílají se pakety FIN (finish), FIN+ACK a ACK. Pokud více než nějaký určitý počet pokusů o odeslání (po spočítaných time-outech) jednoho z 3-way handshake paketů selže (druhá strana neodešle to, co mělo následovat), spojení se považuje za přerušené (i u navazování, i u rušení).

## NAT

TODO: přeložit ty copy & paste z Wiki

Network address translation (zkráceně NAT, česky překlad síťových adres) je funkce síťového routeru pro změnu IP adres paketů procházejících zařízením, kdy se zdrojová nebo cílová IP adresa převádí mezi různými rozsahy. Nejběžnější formou je tzv. maškaráda (maskování), kdy router IP adresy z nějakého rozsahu mění na svoji IP adresu a naopak – tím umožňuje, aby počítače ve vnitřní síti (LAN) vystupovaly v Internetu pod jedinou IP adresou. Router si drží po celou dobu spojení v paměti tabulku překladu adres.

Překlad síťových adres je funkce, která umožňuje překládání adres. Což znamená, že adresy z lokální sítě přeloží na jedinečnou adresu, která slouží pro vstup do jiné sítě (např. Internetu), adresu překládanou si uloží do tabulky pod náhodným portem, při odpovědi si v tabulce vyhledá port a pošle pakety na IP adresu přiřazenou k danému portu. NAT je vlastně jednoduchým proxy serverem (na síťové vrstvě).

## Komunikace

Klient odešle požadavek na komunikace, směrovač se podívá do tabulky a zjistí, zdali se jedná o adresu lokální, nebo adresu venkovní. V případě venkovní adresy si do tabulky uloží číslo náhodného portu, pod kterým bude vysílat a k němu si přiřadí IP adresu. Během přeposílání „ven“ a změny adresy v paketu musí NAT také přepočítat CRC checksum TCP i IP (aby pakety nebyly zahazovány kvůli špatnému CRC, protože změněná adresa je jejich součástí).

Výhodami NAT sú umožnenie pripojenie viacerých počítačov do internetu cez jednu zdieľanú verejnú IP adresu, a zvýšenie bezpečnosti počítačov za NATom (aj keď je to security through obscurity a nie je dobré postaviť bezpečnosť iba na NATe). Nevýhodami potom sú nefungujúce protokoly (napr. aktívne FTP) – čo je zrejmé z fungovania NATu.

## NAT Traversal

NAT traversal refers to an algorithm for the common problem in TCP/IP networking of establishing connections between hosts in private TCP/IP networks that use NAT devices.

This problem is typically faced by developers of client-to-client networking applications, especially in peer-to-peer and VoIP activities. NAT-T is commonly used by IPsec VPN clients in order to have ESP packets go through NAT.

Many techniques exist, but no technique works in every situation since NAT behavior is not standardized. Many techniques require a public server on a well-known globally-reachable IP address. Some methods use the server only when establishing the connection (such as STUN), while others are based on relaying all the data through it (such as TURN), which adds bandwidth costs and increases latency, detrimental to conversational VoIP applications.

## Druhy uspořádání NATu

- *Static NAT*: A type of NAT in which a private IP address is mapped to a public IP address, where the public address is always the same IP address (i.e., it has a static address). This allows an internal host, such as a Web server, to have an unregistered (private) IP address and still be reachable over the Internet.
- *Dynamic NAT*— A type of NAT in which a private IP address is mapped to a public IP address drawing from a pool of registered (public) IP addresses. Typically, the NAT router in a network will keep a table of registered IP addresses, and when a private IP address requests access to the Internet, the router chooses an IP address from the table that is not at the time being used by another private IP address. Dynamic NAT helps to secure a network as it masks the internal configuration of a private network and makes it difficult for someone outside the network to monitor individual usage patterns. Another advantage of dynamic NAT is that it allows a private network to use private IP addresses that are invalid on the Internet but useful as internal addresses.
- *PAT* — PAT (NAT overloading) je další variantou NATu. U této varianty NATu se více inside local adres mapuje na jednu inside global adresu na různých portech. Tedy máme jednu veřejnou adresu a vnitřní síť oadresovanou inside local adresami. Překladová tabulka je rozšířena o dvě položky: inside local port – port, ze kterého byl paket odeslán a inside global port – číslo portu, na který je paket odesláný ze zdrojového portu počítače mapován. Výhodou je, že se tak připojuje více počítačů přes jednu IP adresu.

## ARP

**Address Resolution Protocol (ARP)** se v počítačových sítích s IP protokolem používá k získání ethernetové (MAC) adresy sousedního stroje z jeho IP adresy. Používá se v situaci, kdy je třeba odeslat IP datagram na adresu ležící ve stejné podsíti jako odesílatel. Data se tedy mají poslat přímo adresátovi, u něhož však odesílatel zná pouze IP adresu. Pro odeslání prostřednictvím např. Ethernetu ale potřebuje znát cílovou ethernetovou adresu.

Proto vysílající odešle ARP dotaz (ARP request) obsahující hledanou IP adresu a údaje o sobě (vlastní IP adresu a MAC adresu). Tento dotaz se posílá linkovým broadcastem – na MAC adresu identifikující všechny účastníky dané lokální sítě (v případě Ethernetu na ff:ff:ff:ff:ff:ff). ARP dotaz nepřekročí hranice dané podsítě, ale všechna k ní připojená zařízení dotaz obdrží a jako optimalizační krok si zapíše údaje o jeho odesílateli (IP adresu a odpovídající MAC adresu) do své ARP cache. Vlastník hledané IP adresy pak odešle tazateli ARP odpověď (ARP reply) obsahující vlastní IP adresu a MAC adresu. Tu si tazatel zapíše do ARP cache a může odeslat datagram.

Informace o MAC adresách odpovídajících jednotlivým IP adresám se ukládají do ARP cache, kde jsou uloženy do vypršení své platnosti. Není tedy třeba hledat MAC adresu před odesláním každého datagramu – jednou získaná informace se využívá opakovaně. V řadě operačních systémů (Linux, Windows XP) lze obsah ARP cache zobrazit a ovlivňovat příkazem arp.

Alternativou pro počítač bez ARP protokolu je používat tabulku přiřazení MAC adres IP adresám definovanou jiným způsobem, například pevně konfigurovanou. Tento přístup se používá především v prostředí se zvýšenými nároky na bezpečnost, protože v ARP se dá podvádět – místo skutečného vlastníka hledané IP adresy může odpovědět někdo jiný a stáhnout tak k sobě jeho data.

ARP je definováno v RFC 826. Používá se pouze pro IPv4. Novější verze IP protokolu (IPv6) používá podobný mechanismus nazvaný Neighbor Discovery Protocol (NDP, „objevování sousedů“).

Ačkoliv se ARP v praxi používá téměř výhradně pro překlad IP adres na MAC adresy, nebyl původně vytvořen pouze pro IP síť. ARP se může použít pro překlad MAC adres mnoha různých protokolů na síťové vrstvě. ARP byl také uzpůsoben tak, aby vyhodnocoval jiné typy adres fyzické vrstvy: například ATMARP se používá k vyhodnocení ATM NSAP adres v protokolu Classical IP over ATM.

## ICMP

**ICMP protokol (anglicky Internet Control Message Protocol)** je jeden z jádrových protokolů ze sady protokolů internetu. Používají ho operační systémy počítačů v síti pro odesílání chybových zpráv – například pro oznámení, že požadovaná služba není dostupná nebo že potřebný počítač nebo router není dosažitelný.

ICMP se svým účelem liší od TCP a UDP protokolů tím, že se obvykle nepoužívá síťovými aplikacemi přímo. Jedinou výjimkou je nástroj ping, který posílá ICMP zprávy „Echo Request“ (a očekává příjem zprávy „Echo Response“) aby určil, zda je cílový počítač dosažitelný a jak dlouho paketům trvá, než se dostanou k cíli a zpět.

ICMP protokol je součástí sady protokolů internetu definovaná v RFC 792. ICMP zprávy se typicky generují při chybách v IP datagramech (specifikováno v RFC 1122) nebo pro diagnostické nebo routovací účely. Verze ICMP pro IPv4 je známá jako ICMPv4. IPv6 používá obdobný protokol: ICMPv6.

ICMP zprávy se konstruují nad IP vrstvou; obvykle z IP datagramu, který ICMP reakci vyvolal. IP vrstva patřičnou ICMP zprávu zapouzdří novou IP hlavičkou (aby se ICMP zpráva dostala zpět k původnímu odesílateli) a obvyklým způsobem vzniklý datagram odešle. Například každý stroj (jako třeba mezilehlé routery), který

forwarduje IP datagram, musí v IP hlavičce dekrementovat políčko TTL („time to live“, „zbývající doba života“) o jedničku. Jestliže TTL klesne na 0 (a datagram není určen stroji provádějícímu dekrementaci), router přijatý paket zahodí a původnímu odesílateli datagramu pošle ICMP zprávu „Time to live exceeded in transit“ („během přenosu vypršela doba života“).

Každá ICMP zpráva je zapouzdřená přímo v jediném IP datagramu, a tak (jako u UDP) ICMP nezaručuje doručení. Ačkoli ICMP zprávy jsou obsažené ve standardních IP datagramech, ICMP zprávy se zpracovávají odlišně od normálního zpracování prokolů nad IP. V mnoha případech je nutné prozkoumat obsah ICMP zprávy a doručit patřičnou chybovou zprávu aplikaci, která vyslala původní IP paket, který způsobil odeslání ICMP zprávy k původci.

Mnoho běžně používaných síťových diagnostických utilit je založeno na ICMP zprávách. Příkaz traceroute je implementován odesláním UDP datagramů se speciálně nastavenou životností v TTL políčku IP hlavičky a očekáváním ICMP odezvy „Time to live exceeded in transit“ nebo „Destination unreachable“. Příbuzná utilita ping je implementována použitím ICMP zpráv „Echo“ a „Echo reply“.

#### **Nejpoužívanější ICMP datagramy:**

- *Echo*: požadavek na odpověď, každý prvek v síti pracující na IP vrstvě by na tuto výzvu měl reagovat. Často to z různých důvodů není dodržováno.
- *Echo Reply*: odpověď na požadavek
- *Destination Unreachable*: informace o nedostupnosti cíle, obsahuje další upřesňující informaci
  - Net Unreachable: nedostupná cílová síť, reakce směrovače na požadavek komunikovat se sítí, do které nezná cestu
  - Host Unreachable: nedostupný cílový stroj
  - Protocol Unreachable: informace o nemožnosti použít vybraný protokol
  - Port Unreachable: informace o nemožnosti připojit se na vybraný port
- *Redirect*: přesměrování, používá se především pokud ze sítě vede k cíli lepší cesta než přes defaultní bránu. Stanice většinou nepoužívají směrovací protokoly a proto jsou informovány touto cestou. Funguje tak, že stanice pošle datagram své, většinou defaultní, bráně, ta jej přepošle správným směrem a zároveň informuje stanici o lepší cestě.
  - Redirect Datagram for the Network: informuje o přesměrování datagramů do celé sítě
  - Redirect Datagram for the Host: informuje o přesměrování datagramů pro jediný stroj
- *Time Exceeded*: vypršel časový limit
  - Time to Live exceeded in Transit: během přenosu došlo ke snížení TTL na 0 aniž byl datagram doručen
  - Fragment Reassembly Time Exceeded: nepodařilo se sestavit jednotlivé fragmenty v časovém limitu (např pokud dojde ke ztrátě části datagramů)

Ostatní datagramy jsou používány spíše vzácně, někdy je používání ICMP zneemožněno zcela špatným nastavením firewallu.

## UDP, TCP

UDP – nespoľahlivý nespojovaný prenos datagramov... pridáva len porty

TCP – porty+spoľahlivý spojovaný prenos streamov...

...ďalšie info viď kapitolu o BSD Sockets :-)

## 6.3 Rozhraní BSD Sockets

### Úvod

**Berkeley (BSD) sockets** je rozhranie (API) na vyvíjanie aplikácií ktoré používajú medziprocesovú komunikáciu (napr. v rámci siete). De facto je to štandardná abstrakcia pre sieťové sockety. Primárnym jazykom tohto API je C, pre väčšinu ostatných však existujú podobné rozhrania.

BSD sockets je API umožňujúce komunikáciu medzi dvomi hostmi alebo procesmi na jednom počítači, používajúc koncepciu internetových socketov. Toto rozhranie je implicitné pre TCP/IP a je teda jednou zo základných technológií internetu. Programátori môžu využívať rozhrania socketov na troch úrovniach, najzákladnejšou z nich sú RAW sockety (aj keď túto úroveň sa využijú zväčša len na počítačoch implementujúcich technológie týkajúce sa už priamo internetu).

### Hlavičkové súbory

Berkeley sockets používajú viaceré hlavičkové súbory, okrem iného:

- **sys/socket.h** Core BSD socket functions and data structures.
- **netinet/in.h** AF\_INET and AF\_INET6 address families. Widely used on the Internet, these include IP addresses and TCP and UDP port numbers.
- **sys/un.h** AF\_UNIX address family. Used for local communication between programs running on the same computer. Not used on networks.
- **arpa/inet.h** Functions for manipulating numeric IP addresses.
- **netdb.h** Functions for translating protocol names and host names into numeric addresses. Searches local data as well as DNS.

## TCP

TCP poskytuje koncept spojenia. Proces vytvorí TCP socket pomocou volania `socket()` s parametrom `PF_INET(6)` a `SOCK_STREAM`.

### Server

Vytvorenie jednoduchého TCP servera vyžaduje nasledujúce kroky:

- Vytvorenie TCP socketu (pomocou volania `socket()`)
- Pripojenie socketu na port, kde bude načúvať (`bind()`); parametrami je `sockaddr_in` štruktúra, v ktorej sa nastavuje `sin_family` (`AF_INET`-IPv4, `AF_INET6`-IPv6) a `sin_port`)
- Pripravenie socketu na načúvanie na porte (`listen()`).

- Akceptovanie prichodzích pripojení pomocou *accept()*. Táto funkcia blokuje volajúceho do príchodu pripojenia a vracia identifikátor prichodzieho spojenia, ktorý sa môže ďalej použiť. *accept()* je hneď možné volať na pôvodný identifikátor socketu na čakanie na ďalšie spojenia.
- Komunikácia s klientom pomocou *send()*, *recv()* alebo *read()* a *write()*
- Keď už socket nie je potrebný, je možné ho zavrieť pomocou *close()*.

## Klient

Vytvorenie TCP klienta vyžaduje nasledujúce kroky:

- Vytvorenie TCP socketu (pomocou volania *socket()*)
- Pripojenie k serveru pomocou *connect()* (znovu sa používa štruktúra *sockaddr\_in*, vyplní sa *sin\_family*, *sin\_port* (ako pri serveri) + *sin\_addr* (adresa servera))
- Komunikácia so serverom pomocou *send()*, *recv()* alebo *read()* a *write()*
- Keď už socket nie je potrebný, je možné ho zavrieť pomocou *close()*.

## UDP

UDP je protokol bez spojenia (connectionless) a bez garancie doručenia správ. UDP balíky môžu (okrem správneho počtu/poradia) doraziť mimo poradia, môžu byť duplikované alebo nedoraziť ani raz. Vďaka minimálnym garanciám má UDP oproti TCP oveľa menšiu réžiu. Keďže tento protokol nevytvára spojenia, dáta sa prenášajú v datagramoch.

Adresovací priestor UDP (porty UDP) je úplne nezávislý na priestore portov TCP.

## Server

Keďže sa nevytvárajú spojenia, po vytvorení socketu (ako pri TCP pomocou *socket()+bind()*) už aplikácia (server) rovno čaká prichodzie datagramy pomocou funkcie *recvfrom()*. Na konci sa socket zatvára pomocou *close()*.

## Klient

U klienta je tiež oproti spojovanej verzii zjednodušenie - stačí vyrobiť socket (pomocou *socket()*) a potom už iba posielat datagramy pomocou *sendto()*. Na konci sa socket zatvára pomocou *close()*.

## Najdôležitejšie funkcie

- **int socket(int domain, int type, int protocol)**
  - *domain* (PF\_INET — PF\_INET6)
  - *type* (SOCK\_STREAM, SOCK\_DGRAM, SOCK\_SEQPACKET (spoľahlivé zoradené balíky), SOCK\_RAW (raw protokoly nad sieťovou vrstvou))
  - *protocol* (väčšinou IPPROTO\_IP, ďalšie sú v *netinet/in.h*)

- **struct hostent \*gethostbyname(const char \*name)**  
**struct hostent \*gethostbyaddr(const void \*addr, int len, int type)**
  - Vracia pointer na hostent štruktúru, ktorá popisuje internetového hosta zadaného pomocou mena alebo adresy (obsahuje buď informácie od name servera, alebo z lokálneho /etc/hosts súboru)...
- **int connect(int sockfd, const struct sockaddr \*serv\_addr, socklen\_t addrlen)**
- **int bind(int sockfd, struct sockaddr \*my\_addr, socklen\_t addrlen)**
- **int listen(int sockfd, int backlog)**
  - *backlog* určuje maximálne koľko pripojení môže vo fronte čakať na akceptovanie...
- **int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen)**  
do *cliaddr* sa vyplnia informácie o klientovi...

### Blokujúce a neblokujúce volania

BSD sockety môžu fungovať v dvoch módoch - blokujúcich a neblokujúcich. V blokujúcom móde funkcie nevrátia riadenie programu, kým nie sú spracované všetky dáta - čo môže spôsobiť rôzne problémy (program „zamrzne“, keď socket načúva; alebo keď socket čaká na dáta, ktoré neprichádzajú). Typicky sa nastavuje neblokujúci mód pomocou *fcntl()* alebo *ioctl()*

## 6.4 Spolehlivosť - spojované a nespojované protokoly, typy, detekcie a oprava chýb

### Spolehlivosť

#### Spolehlivosť:

- môže byť zajištená na ktorejkoľvek vrstve (kromě fyzické)
- TCP/IP rieši na transportní (TCP), ISO/OSI očakáva spolehlivosť na všetkých (počínajúc linkovou)
- väčší režie, zpožďenie pri chybách

#### Nespoľehlivá komunikácia:

- menší režie, lepší odezva
- výhodné pre audio/video prenosy, kde lze tolerovať ztráty

### Spojované a nespojované protokoly

**Spojovaná komunikácia:** stavová, virtuálna okruhy, navazovanie a ukončenie spojenia. Viz TCP.

**Nespojovaná komunikácia:** zasílanie správ, datagramy (UDP), nestavová, bez navazovania a ukončovania. Viz UDP.

## Detekce a oprava chyb

- schopnost poznat, že došlo k nějaké chybě při přenosu
- Hammingovy kódy - příliš velká redundance, nepoužívané
- potvrzování (ACK) - viz TCP/IP
  - příjemce si znovu nechá zaslat poškozená/nedoručená data
  - podmínkou existence zpětného kanálu (alespoň half-duplex)
  - jednotlivé vs. kontinuální
  - kladné (ACK) a záporné (NAK)
  - samostatné vs. nesamostatné (piggybacking)
  - metoda okénka
  - selektivní opakování vs. opakování s návratem
- parita - příčná, podélná
- kontrolní součty
- cyklické redundantní součty (CRC)
- druhy chyb: pozměněná data, shluky chyb, výpadky dat
- při chybě nutno vyžádat si celý rámeček znovu

## 6.5 Bezpečnost – IPSec, principy fungování AH, ESP, transport mode, tunnel mode, firewalls

### IPSec

- Není to pouze jeden protokol ale soustava vzájemně provázaných opatření a dílčích protokolů pro zabezpečení komunikace pomocí IP protokolu, funguje na síťové vrstvě – není závislý na protokolech vyšších vrstev jako je TCP a UDP (např. SSL protokol pracuje na transportní vrstvě)
- Podporováno jak v IPv4 (podpora nepovinná) i v IPv6 (podpora povinná)
- Zajišťuje důvěrnost (šifruje přenášená data) a integritu (data nejsou při přenosu změněna)
- několik desítek RFC dokumentů
- autentifikace – ověření původu dat (odesílatele)
- kryptování – šifrování komunikace (mimo IP hlavičky)
- může být implementováno na bráně (security gateway, lokální síť je považována za bezpečnou) nebo na koncových zařízeních
- **SA (Security Association)**
  - point-to-point bezpečnostní spoj (návrh uvažuje i o jiných variantách)
  - pro každý směr a každý protokol nutné mít vlastní SA spoj

IPsec módy:

- **transport mode**
  - IP hlavička nechráněná (jeden z důvodů je užívání systému NAT), tělo paketu šifrováno (data vyšších protokolů)
  - použitelné jen na koncových stanicích

- **tunnel mode**

- pakety jsou celé (včetně hlavičky) zašifrovány a vloženy do dalšího paketu, na druhé straně rozbaleny
- povinné pro security gateways, volitelné pro koncové stanice
- ve vnější IP hlavičce se jako příjemce uvádí security gateway na hranici cílové sítě

IPsec protokoly:

- **AH (Authentication Header)**

- komunikující strany se dohodnou na klíči
- k datům se připojuje hash
- chrání také před replay attack
- provádí autentizaci a kontrolu změny dat, neprovádí šifrování

- **ESP (Encapsulating Security Payload)**

- provádí autentizaci a také šifruje obsah
- pro šifrování používá 3DES, Blowfish aj. (původně DES, již není považováno za bezpečné)

Dohoda klíčů:

- před použitím protokolu AH či ESP si musí strany dohodnout klíče
- manuální konfigurace
- automatická konfigurace – IKE (Internet Key Exchange) protokol

## Firewally

- sledování a filtrování komunikace na síti
  - blokování – zabraňuje neoprávněnému přístupu
  - prostupnost – propouštění povoleného toku
- paketové filtry – např. na routeru
- stavový firewall (stateful) – sleduje vztahy mezi pakety, ohlíží se na historii
- na různých vrstvách
  - síťová – pouze dle zdrojových a cílových adres a protokolu
  - transportní – také podle portů
  - aplikační – dle obsahu (dat)
- demilitarizovaná zóna (DMZ):
  - jiné řešení bezpečnosti
  - přístup ven pouze přes specializovaná zařízení (proxy, brány), nelze přímo
    - platí pro oba směry

## 6.6 Internetové a intranetové protokoly a technologie – DNS, SMTP, FTP, HTTP, NFS, HTML, XML, XSLT a jejich použití

### DNS (Domain Name System)

- Řešení které umožňuje používat symbolická jména místo číselných adres realizované pomocí DNS serverů, distribuované řešení – výpadek i více serverů nevyřadí službu z provozu
- hierarchický (stromový) prostor jmen. Kořenem je tzv. kořenová doména — tečka, pod ní se v hierarchii nachází domény nejvyšší úrovně (com, edu, cz, uk, ...), strom je možné rozdělit do zón a její správu svěřit někomu dalšímu – právě možnost delegování pravomocí a distribuovaná správa tvoří klíč. vlastnosti DNS a stojí za jeho úspěchem
- DNS servery
  - Primární – zde data vznikají, zde se musí také provádět změny, každá doména obsahuje právě jeden
  - Sekundární -- automatická kopie primárního, průběžně si aktualizuje data, slouží jednak jako záloha pro případ výpadku prim. serveru a také pro rozkládání zátěže, každá doména musí mít alespoň jeden sekundární server
  - Pomocný (caching only) -- slouží jako vyrovnávací paměť pro snížení zátěže celého systému, uchovává si odpovědi a poskytuje je při opakování dotazů dokud jim nevyprší životnost
- Odpovědi z primárního a ze sekundárních serverů jsou autoritativní – platné. Z pomocného serveru je odpověď neautoritativní – klient může požádat o autoritativní odpověď.
- soubor hosts
- doména, zóna, delegace, TLD, ccTLD, gTLD
- syntaxe jmen, FQDN (plně kvalifikované...)
- IDN
- iterativní dotaz, rekurzivní dotaz, cachování, resolvery, TTL, autoritativní odpověď
- Resource Records (RR) – jednotka informace v DNS
  - formát: [name type class TTL rlength rdata]
  - class – dnes vždy IN (internet)
  - TTL – time to live (doba platnosti záznamu)
  - types – A (IPv4 adresa), NS (hostname nameserveru), MX (mailserver), PTR (domain name pointer – reverse DNS), AAAA (IPv6 adresa), SPF, TXT, SRV, ...
  - rlength – délka dat, rdata – vlastní data
- DNS protokol – TCP/UDP, truncation

## SMTP (Simple Mail Transfer Protocol)

- Internetový protokol určený pro přenos zpráv elektronické pošty mezi stanicemi. Protokol zajišťuje doručení pošty pomocí přímého spojení mezi odesílatelem a adresátem; zpráva je doručena do tzv. poštovní schránky adresáta, ke které potom může uživatel kdykoli (offline) přistupovat a vybírat zpráva pomocí protokolů POP3 popř. IMAP.
- Funguje nad protokolem TCP, používá port 25
- Pro netextové přenosy je využíván standart MIME (řeší problém národních abeced, formátování, příloh . . .) – rozšíření formátu zpráv – Quoted-Printable, Base64, mime-type
- SMTP – protokol pro přenos zpráv
- RFC822 – formát zpráv – 7-bitová data
- POP3, IMAP – stahování zpráv ze schránky
- struktura zprávy – hlavička + tělo + přílohy
- e-mailové adresy, MX záznamy, priority

## FTP (File Transfer Protocol)

- Jeden ze sady protokolů TCP/IP, netransparentní řešení – uživatel si uvědomuje že se soubor nachází na vzdáleném počítači, typicky se vzdálený soubor celý přenesou na místní počítač a zde se s ním pracuje ; v rámci TCP/IP existuje zjednodušená verze – TFTP
- Dva režimy – textový a binární ; vychází z modelu klient/server, klient je typicky aplikační program, server je obvykle systémový proces (démon,. . .)
- Jednotný formát pro potřeby přenosu dat, veškeré konverze provádí koncové uzly
- Používají se 2 různá spojení – řídicí (přenos příkazů – FTP má vlastní řídicí jazyk) a datové(přenos souborů), řídicí navazuje klient, datové navazuje server (okrem passive módu)
- příkazy – řízení přístupu, nastavení parametrů, výkonné příkazy
- TFTP

## NFS (Network File System)

- Jeden ze sady protokolů TCP/IP, slouží pro transparentní sdílení souborů (uživatel/aplikace si neuvědomuje že se soubor nachází na vzdáleném počítači), typicky se vzdálený soubor chová „tváří“ jako místní soubor a také se s ním tak pracuje
- Je použitelný na různých platformách
- Bezstavový protokol (v4 už je ale stavový), díky tomu je velmi robustní – to je důvod jeho úspěšnosti
- Využívá protokoly RPC (vzdálené volání procedur) a XDR (definuje jednotný způsob reprezentace přenášených dat nezávislý na konkrétní architektuře příjemce a odesílatele)
- mount server

## HTTP (Hyper-Text Transfer Protocol)

- Protokol původně určený pro výměnu hypertextových dokumentů ve formátu HTML, v současné době užíván i pro přenos dalších informací – pomocí rozšíření MIME umí přenášet jakýkoli soubor (podobně jako SMTP)
- Existuje bezpečnější verze HTTPS, umožňuje přenášená data šifrovat
- Funguje systémem dotaz-odpověď, při zaslání více dotazů není možné rozpoznat zda spolu souvisí – HTTP je bezstavový protokol (nepříjemná vlastnost pro implementaci složitějších procesů přes HTTP) – proto byl rozšířen o HTTP cookies (umožňují uchovávat info o stavu na počítači uživatele)
- verze 0.9 – bez hlaviček, minimální možnosti
- verze 1.0 – rozšiřující hlavičky, podpora MIME
- verze 1.1 – virtuální servery, jedno spojení pro více přenosů (keep-alive), komprimace dat
- GET, HEAD, POST
- cookies
- cachování

## HTML (Hyper-Text Markup Language)

- značkový jazyk pro hypertext, definuje obsah, nikoliv vzhled
- CSS
- statické a dynamické HTML dokumenty – CGI, ISAPI, NSAPI, ASP, PHP
- skripty, Java, ActiveX

## XML (eXtensible Markup Language)

- Značkový jazyk vyvinut a standardizován konsorciem W3C, umožňuje snadné vytváření konkrétních značkových jazyků pro různé účely
- Určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů, umožňuje popsat strukturu dokumentu z hlediska věcného obsahu, nezabývá se sám o sobě vzhledem dokumentu, vzhled dokumentu se definuje připojeným stylem
- Pomocí různých stylů je možné provést transformaci do jiného typu dokumentu, nebo jiné XML struktury (výsledkem může být např. HTML, PostScript, . . .)

## XSLT (eXtensible Style Sheet Language Transformations)

- Transformace sloužící pro převod dat ve formátu XML do lib. jiného požadovaného formátu (nejčastěji HTML, jiného XML, ale také PDF, či RTF, . . .), struktura výstupu není definována přímo standardem – je závislá na procesoru XSLT (program který provede transformaci)
- K provedení transformace jsou třeba 2 soubory:
  - Soubor, který obsahuje zdrojová data, která budou transformována. Struktura tohoto souboru vyjma obecných vlastností XML není blíže specifikována.

- Soubor, obsahující vzorec pro transformaci, napsaný v jazyce XSL