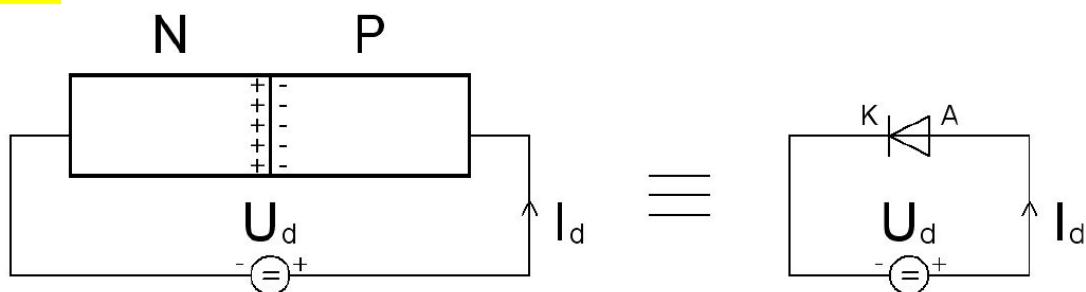


1. Princip činnosti polovodičových prvků (dioda, bipolární a unipolární tranzistor ve spínacím režimu, realizace logických členů NAND a NOR v technologii CMOS).

- elektrony tečou od - k +, proud značíme, že teče od + k -.
- Základem jsou polokovy, hlavně **křemík**.
- Existují 2 typy polovodičových materiálů založených na příměsích:
 - **N** (negativní) - Při vazbě atomů vzniká v polokovu **volný elektron**, který je schopný vést proud - elektronová vodivost. Dopování **fosforem**.
 - majoritní: elektrony
 - minoritní: díry
 - **P** (pozitivní) - Při vazbě vzniká "díra" (**místo kde chybí elektron**) pro vedení proudu. Dopování **borem**.
 - majoritní: díry
 - minoritní: elektrony
- Oba typy polovodičových materiálů **nemají na venek elektrický náboj** (uvnitř materiálů je stejný počet protonů jako elektronů), pokud se spojí na mikroskopické úrovni, dojde k přesunu části (maličko) elektronů z **N** do **P**. V **P** vznikne záporný náboj a v **N** kladný náboj.
- V místě dotyku vzniká **potenciálová bariéra** (~0.7 V u křemíku).
- Přechod v propustném směru (anoda = kladná, katoda = záporná)

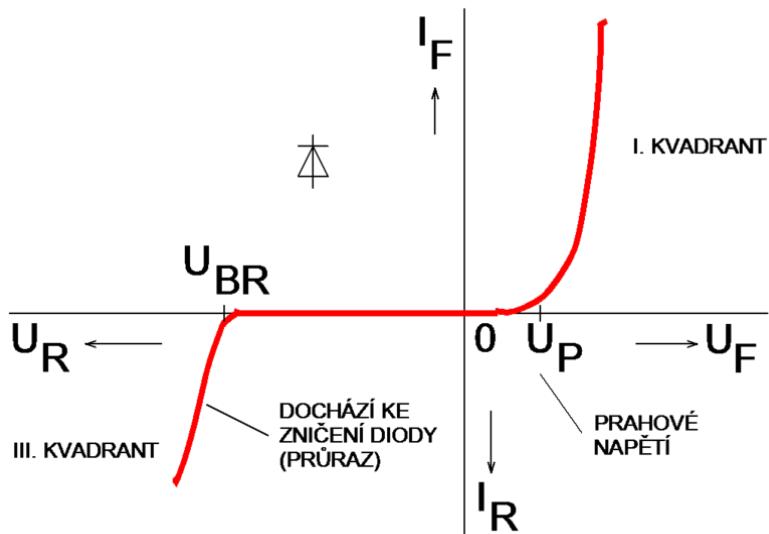


Polovodičová dioda

obsahuje PN přechod, propouští proud pouze v jednom směru. Ideální dioda by propouštěla proud pouze v jednom směru, reálná dioda propouští v jednom lépe než v druhém. V závěrném směru prakticky nepropouští proud až do tzv. průrazného napětí, poté začne procházející proud strmě narůstat, což obvykle vede na zničení diody (přehřátí a spálení). Pokud k tomu není dioda určená (zenerovy diody), ale i ty

Ize spálit příliš velkým proudem, musí být omezen prvky v obvodu.

- **Přechodová charakteristika diody** - (obr.)

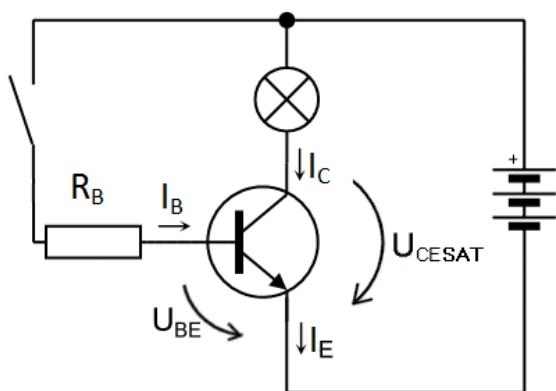


- Existuje více druhů diod (usměrňovací, spínací, fotodiody, luminiscenční...)

Tranzistor

Elektronická součástka, která má schopnost zesilovat proud.

Tranzistor jako spínač (spínací režim) - Dokáže spínat velké proudy a pracuje velmi rychle. Pokud do báze teče proud (stačí malý) tak začne mezi kolektorem a bází a tedy i spotřebičem procházet velký proud. Malým proudem spínáme proud velký.



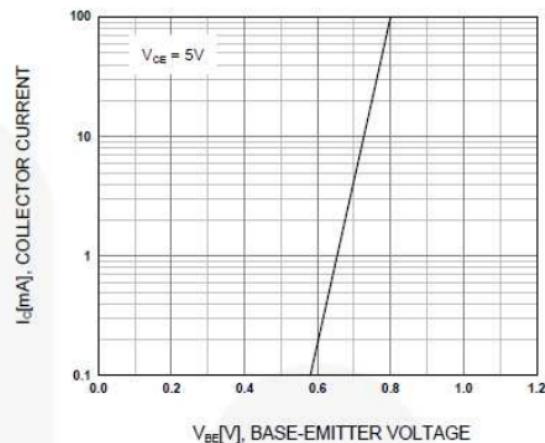
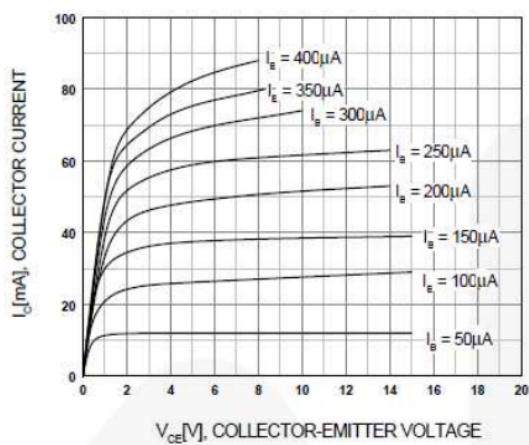
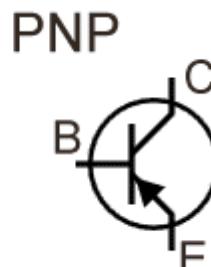
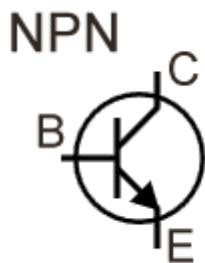
Bipolární tranzistor BJT (Bipolar Junction Transistor)

Přenos proudu uskutečňují oba nosiče jak N tak P. Jedná se o **2 PN** přechody vedle sebe - **PNP** nebo **NPN** tranzistor.

- **E - Emitor** je hodně nadopovaný, **B - Báze** je málo nadopovaná a velmi

tenká, C - Kolektor je středně nadopovaný

- **Princip NPN** - emitor je připojen na katodu (elektrodu uvolňující elektrony) a společně s bází tvoří diodu v propustném směru. Pokud je na bázi přivedeno napětí o $\sim 0.7V$ větší než na emitoru, začne touto diodou procházet proud. Díky tomu, že emitor je daleko více dopován než báze, dostává se do báze mnohonásobně větší množství elektronů, než je zde děr. Protože báze je navíc tenká, jsou tyto elektrony přitahovány kladným elektrickým nábojem vznikajícím na kolektoru, který je připojen na anodu, a tranzistorem tak prochází proud. Pokud není na bázi stejně napětí jako na emitoru, nedojde k překonání **potenciálové bariéry** a celým tranzistorem proud neteče.
<https://youtu.be/DXvAlwMAXiA>
- **Princip PNP** - emitor musí být připojen na anodu, kolektor na katodu a napětí mezi bází a emitorem musí být $\sim -0.7V$ (mezi emitorem a bází $\sim -0.7V$), jinak je princip obdobný s tím, že uvažujeme pohyb děr..



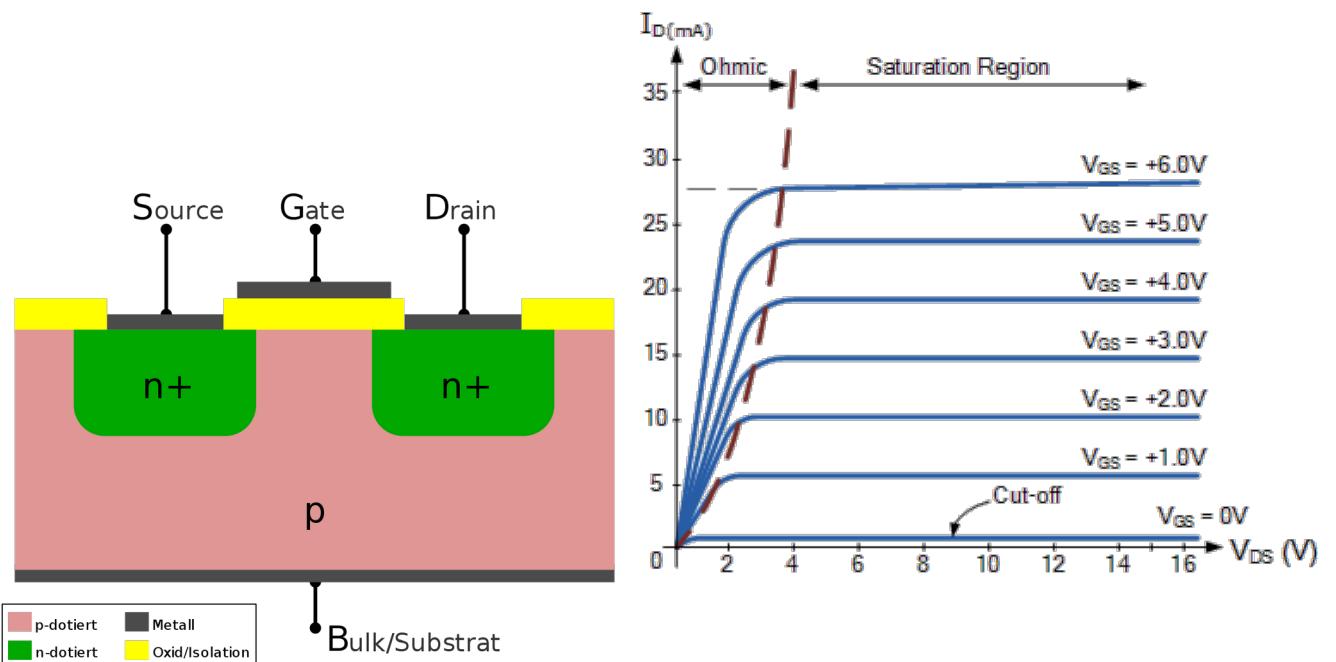
Unipolární tranzistor MOSFET (Metal Oxide Semiconductor Field Effect Transistor)

Přenos proudu uskutečňuje pouze jeden nosič - **N-channel** nebo **P-channel**, dále jsou děleny na **enhancement** (nevedou proud a musí se zapnout) a **depletion** (vedou proud a musí se vypnout). Jsou řízeny napětím.

- **Gate** - řídící elektroda - přes vrstvu dielektrika (oxid - izolace) připojena k substrátu, **Source** - zdrojová elektroda - silně dopovaná, **Drain** - výstupní

elektroda - silně dopovaná, **Body/Substrate** - málo dopován.

- **Princip NPN enhancement** - na Source je přivedeno záporné napětí (stejné jako na Bulk/Body, aby tranzistor nefungoval jako dioda) a na Drain je přivedeno kladné napětí (Drain a Source lze zaměnit). Aby tekla ze Source do Drain proud, musí být na Gate dovedeno kladné napětí (řádově menší, než mezi Source a Drain). Kolem Gate tak vznikne kladný náboj a začne odpuzovat elektrony ze Substrate/Body (díry se přesunují dolů) a zároveň přitahovat elektrony ze Source. Začne se vytvářet postupně od Source k Drain kanál (trojúhelníkový tvar). Při dostatečném napětí na Gate je elektrické pole natolik velké, že kanál propojí Source s Drain a začne téct proud (s vyšším Gate napětím se dále zvyšuje). Zvyšování napětí mezi Source a Drain

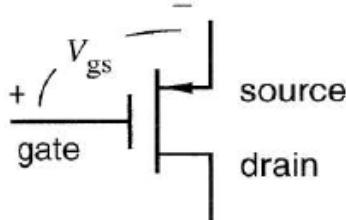


způsobní i zvyšování **potenciálové bariéry** kolem Drain, což způsobí saturační fázi (Saturation Region - nezvyšuje se prakticky protékající proud).

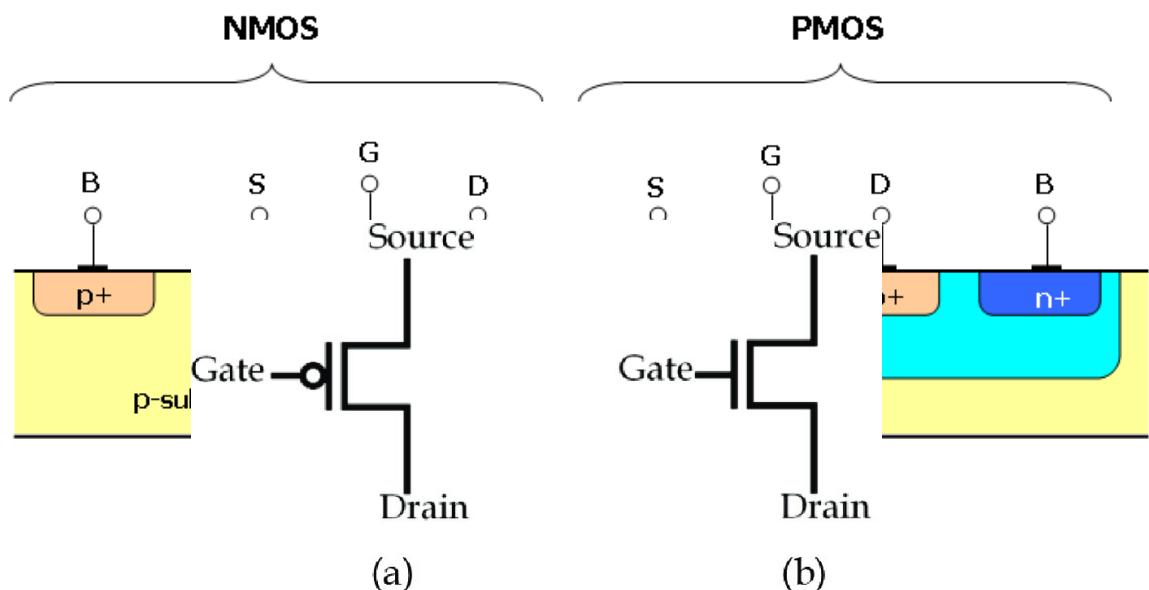
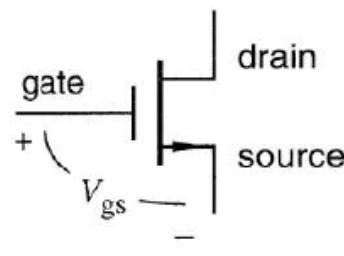
How Does a MOSFET Work?

- **CMOS** - Typ MOSFET, využívá se pro výrobu logických integrovaných obvodů. **Nízká spotřeba a odolnost proti šumu**. Dělí se na (a) **PMOS** a (b) **NMOS**.

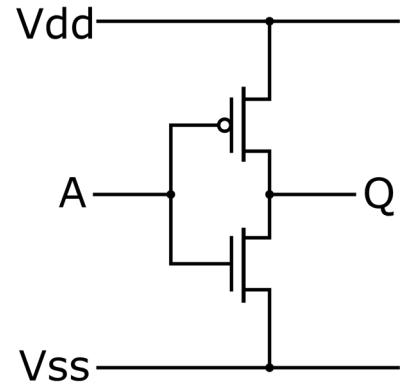
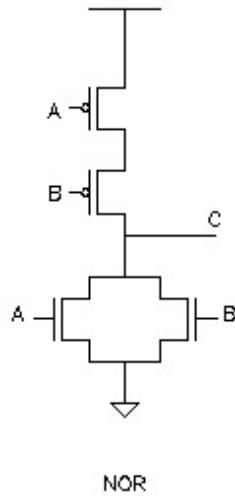
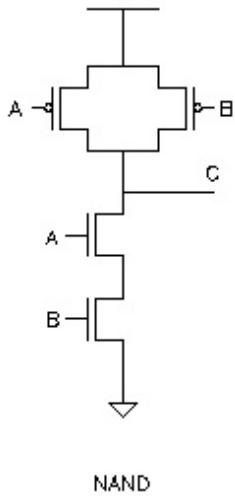
- **PMOS** - záporné (**low - 0**) napětí na gate sepne tranzistor - **děrová vodivost**. Source a Body/Substrate se zapojují na **kladný** pól zdroje, Drain se zapojuje na **záporný** pól zdroje. Source a Drain jsou typu **P**, Substrate je typu **N**. Na Source je menší napětí než na drain.



- **NMOS** - kladné (**high - 1**) napětí na gate sepne tranzistor - **elektronová vodivost**. Source a Body/Substrate se zapojují na **záporný** pól zdroje, Drain se zapojuje na **kladný** pól zdroje. Source a Drain jsou typu **N**, Substrate je typu **P**. Na Source je větší napětí než na Drain.



- **NAND, NOR A INVERTOR (Technologie CMOS)**



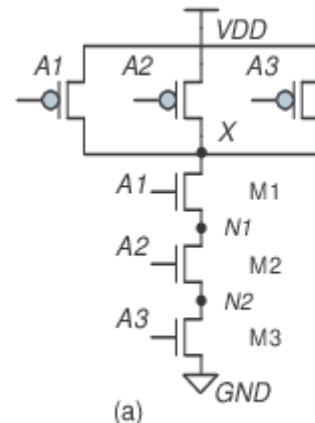
Z hradla **NAND** lze záměnou **PMOS** za **NMOS** a naopak vytvořit **OR**, z hradla **NOR** lze záměnou **PMOS** za **NMOS** a naopak vytvořit **AND**.

NAND (Shefferova funkce) logika

$$\begin{aligned}\neg p &\equiv \neg(p \wedge p) \\ p \wedge q &\equiv \neg(\neg(p \wedge q)) \equiv \neg(\neg(p \wedge q) \wedge \neg(p \wedge q)) \\ p \vee q &\equiv \neg(\neg p \wedge \neg q) \equiv \neg(\neg(p \wedge p) \wedge \neg(q \wedge q)) \\ p \rightarrow q &\equiv \neg p \vee q \equiv \neg(p \wedge \neg q) \equiv \neg(p \wedge \neg(q \wedge q))\end{aligned}$$

NOR (Peirceova funkce) logika

$$\begin{aligned}\neg p &\equiv \neg(p \vee p) \\ p \wedge q &\equiv \neg(\neg p \vee \neg q) \equiv \neg(\neg(p \vee p) \vee \neg(q \vee q)) \\ p \vee q &\equiv \neg(\neg(p \vee q)) \equiv \neg(\neg(p \vee q) \vee \neg(p \vee q)) \\ p \rightarrow q &\equiv \neg p \vee q \equiv \neg(p \vee p) \vee q \equiv \\ &\quad \neg(\neg(\neg(p \vee p) \vee q) \vee \neg(\neg(p \vee p) \vee q))\end{aligned}$$



Odkazy:

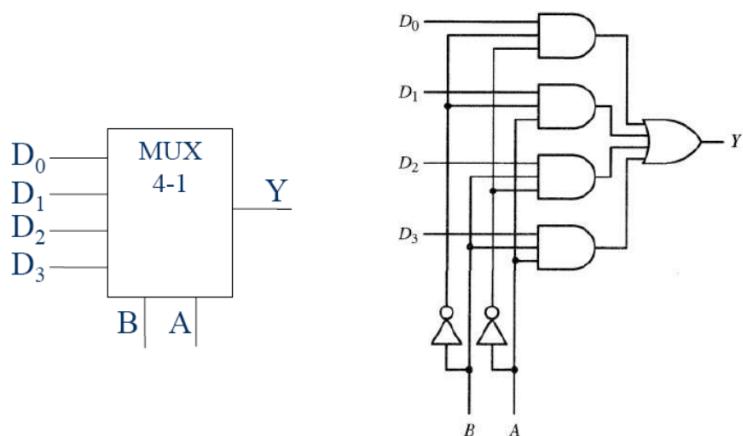
- Jak funguje tranzistor - [Transistors, How do they work ?](#)
- VA charakteristika - https://upload.wikimedia.org/wikipedia/commons/4/43/ThresholdFormation_no_watermark.gif
- [Working of Transistor as a Switch - NPN and PNP Transistors](#)
- <http://old.spsemoh.cz/vyuka/zel/tranzistory-bip.htm>
- [Transistor Logic NOT Gate - Inverter](#)
- Prakticky polovodiče a tranzistory - kapitola 7
https://knihy.nic.cz/files/edice/hradla_volty_jednocipy.pdf

2. Kombinační logické obvody (multiplexor, demultiplexor, kodér, dekodér, binární sčítka).

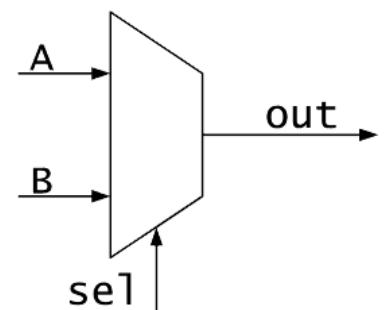
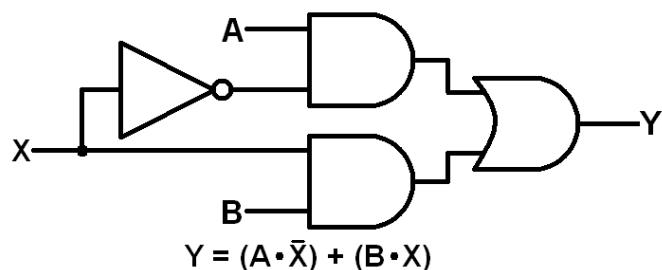
- Kombinační obvod je takový, kde jsou **hodnoty výstupních hodnot závislé pouze na kombinaci hodnot na vstupu**. Tedy nemají paměť.

Multiplexor

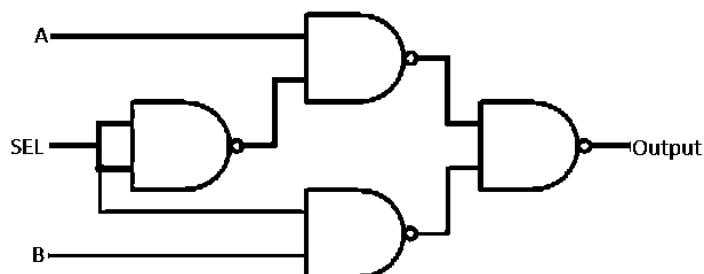
Kombinací **N řídících vstupů** vybírá na **výstup hodnotu jednoho** z až 2^N vstupů.



2-1 multiplexor pomocí **NOT, AND, OR**

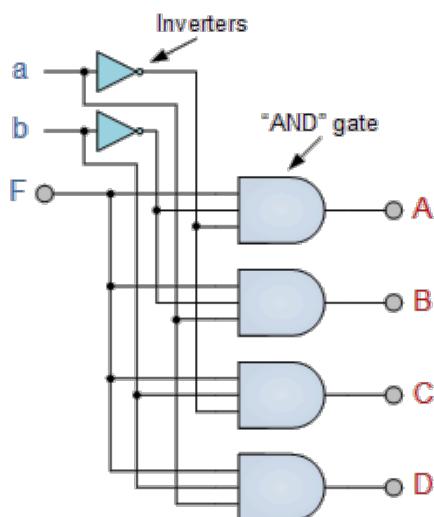


2-1 multiplexor pomocí **NAND hradel**

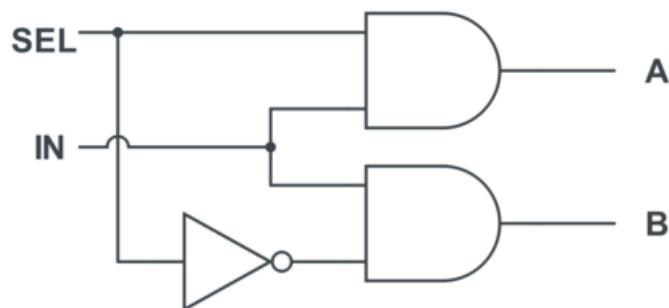
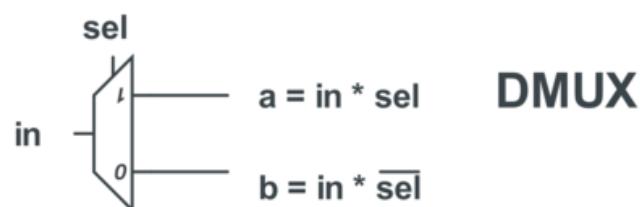


Demultiplexor

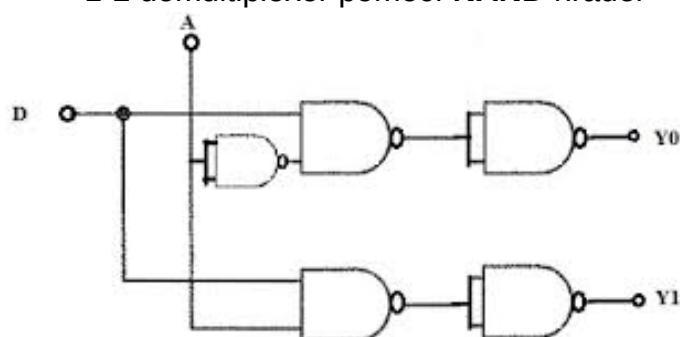
Kombinací **N řídících vstupů** vysílá vstup na jeden z obvykle 2^N výstupů.



1-2 demultiplexor pomocí **NOT** a **AND** hradel



1-2 demultiplexor pomocí **NAND** hradel

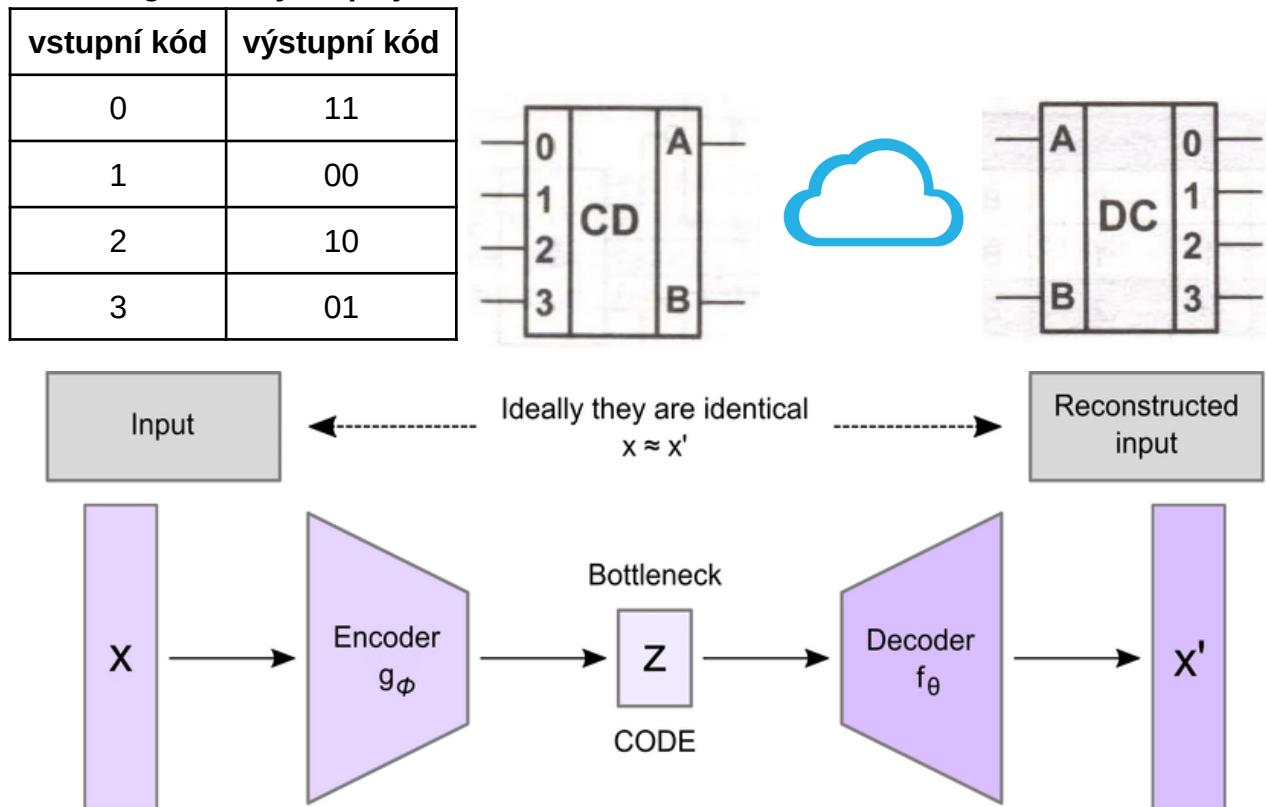


Kodér a dekodér

Kodér kóduje (**kombinuje**, převádí) informaci z **N** vstupů na **K** výstupů na základě **kombinační tabulky**. Obecně platí, že $N > K$ (jinak jde spíš o dekodér) a $2^K \geq N$ (jinak nelze zaručit jednoznačné kódování).

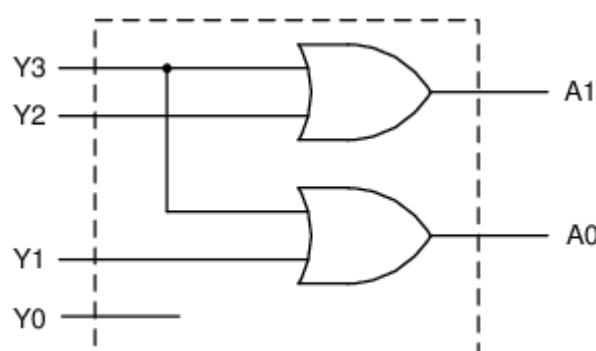
Dekodér je kombinační logický obvod **komplementární** ke kodéru. Dekóduje (**kombinuje**, převádí) informaci z **N** vstupů na **K** výstupů. Obecně platí, že $N < K$.

- To znamená, že zapojení kodéru a za ním dekodéru se **stejnou kombinační** tabulkou, která je **jednoznačná** (pro každý vstup generuje unikátní výstup), se bude chovat, jako by tam nebyl ani jeden. Může jít například o kodér ze **7-segmentového displeje** na **BCD** (nepoužívá se...) a dekodér z **BCD** na **7-segmentový displej**.



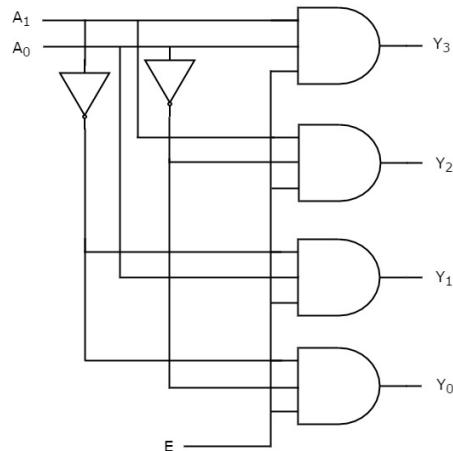
Binární kodér (Encoder)

Umožňuje na **N výstupů** zakódovat až **2^N vstupů**. Musí ale platit, že vždy je **aktivní pouze jeden** ze vstupů. Pomocí kodéru lze například převádět stisknutou klávesu (vstup) na její binární hodnotu. Klávesa 0 je zapojena na první vstup, klávesa 1 na druhý atd. Na výstupu je poté 0b0000 pro klávesu 0, 0b0001 pro klávesu 1, 0b0010 pro klávesu 2, ...



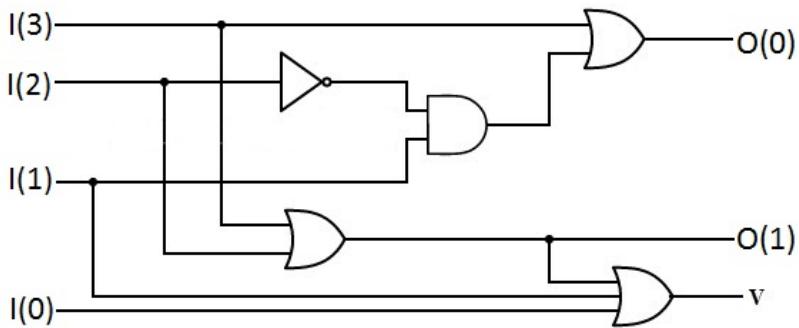
Binární dekodér (Decoder)

Dekóduje **N vstupů** na 2^N **výstupů**, narozdíl od kodéru mohou být vstupy libovolně aktivní. Pro daný vstup je ale aktivní vždy **pouze jeden výstup**. Dekodér lze například použít pro adresaci zařízení na sběrnici (binární adresa umožňuje aktivovat jedno z připojených zařízení). Dekodér lze implementovat pomocí demultiplexoru, vstup se zapojí na log. 1 a na **sel** se zapojí vstupy.



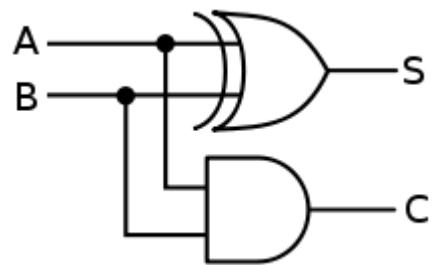
Prioritní kodér

Narozdíl od běžného binárního kodéru umožňuje mít na **vstupu více aktivních vodičů**. V tom případě na výstup kóduje hodnotu odpovídající tomu s **největší prioritou**. Využívá se například k řízení obsluhy přerušení. Na obrázku má **I(3) největší** prioritu a **I(0)** nejmenší.



Binární sčítáčka

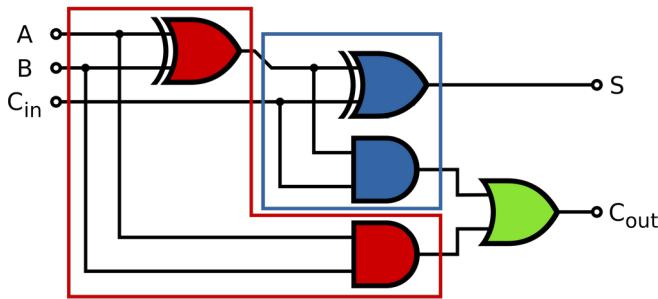
- **Poloviční sčítáčka (half adder)** - realizuje sčítání dvou jednobitových čísel. Výstupem je **jednobitový součet (S)** a jednobitový **příznak přenosu** do vyššího řádu (**C**). Poloviční sčítáčka ale sama **nedokáže zpracovat přenos z nižšího řádu**.



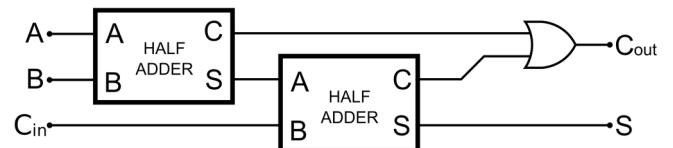
4 až 2 Prioritní Encoder

I ₃	I ₂	I ₁	I ₀	O ₁	O ₀	PROTI
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

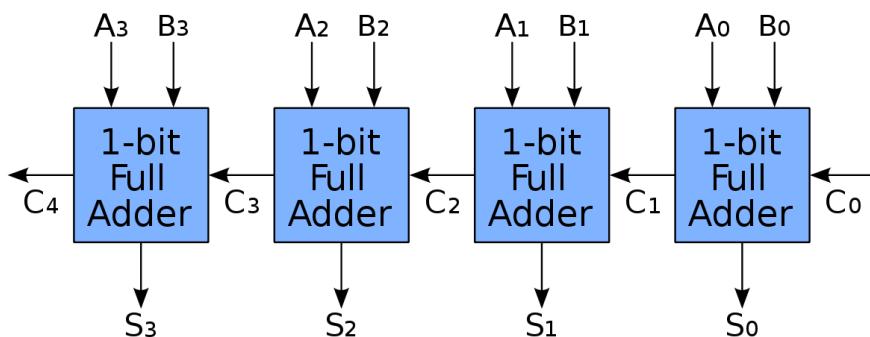
- **Úplná sčítáčka (full adder)** - narozdíl od poloviční sčítáčky umožňuje zpracovat i přenos z nižšího řádu. Vstupem jsou tedy sčítance **A**, **B** a přenos z nižšího řádu **C**. Výstupem je součet **S** a přenos do vyššího řádu **C**.



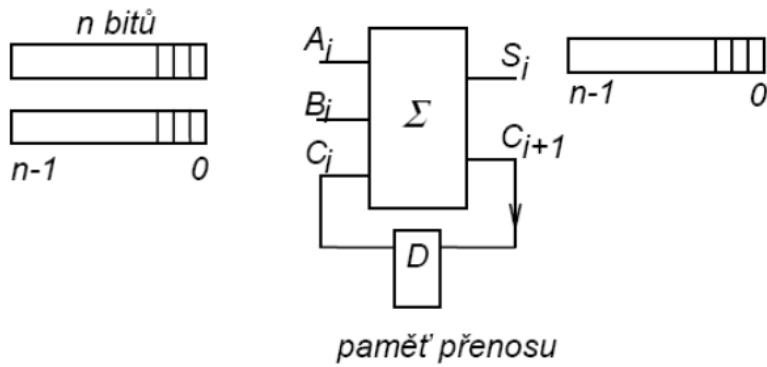
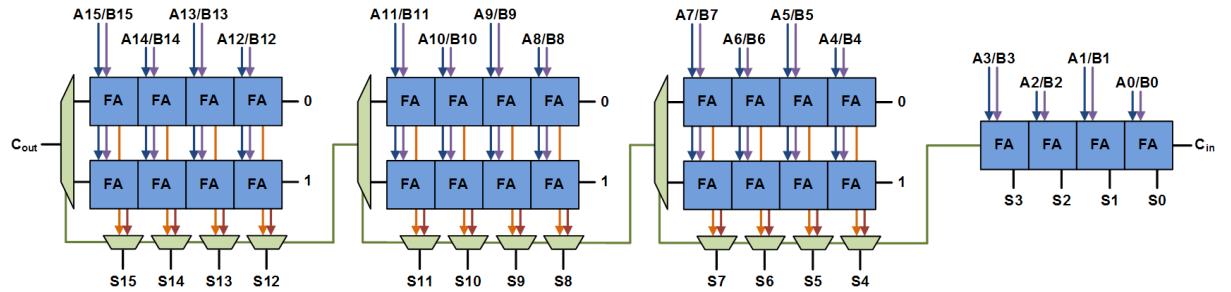
- **Sčítačka s přenosem (Ripple carry adder)** - Vzniká propojením více úplných sčítaček, tak aby se mohl šířit carry bit. **Pseudoparalelní**, přenos carry se **postupně šíří sčítačkou**. Poměrně pomalá kvůli čekání na carry bit. Zpoždění každého carry bitu je rovno zpoždění **3 log. členů**, zpoždění



sečtení je rovno zpoždění **2 log. členů**. Z toho plyne, že zpoždění $S_i = 3(i-1) + 2 = 3*i - 1$ ($i-1$ protože na 1. carry bit se nečeká)



- RCA lze urychlit **sčítačkou s výběrem přenosu (carry-select adder)**
- předpočítáním si výsledků pro carry = 0 i carry = 1; 4 bitovými sčítačkami a zvolení těch správných pomocí multiplexoru na základě výsledků z předchozích sčítaček. Zpoždění této sčítačky je rovno zpoždění 4 bitové RCA sčítačky = $4*3-1 = 11$ log. členů a zpoždění 3 multiplexorů = $3*3 = 9$ log. členů. Celkově tedy **20 log. členů**.
- libovolně dlouhá binární čísla lze také sečíst jednou **úplnou binární sčítačkou** za využití **posuvných registrů** (pro sčítance a součet; délka registrů určuje maximální délku binárního čísla) a **klopného obvodu typu D**, který je použit pro zapamatování si carry bitu (přenosu) **nejedná se kombinační obvod**.



- **Sčítačka s predikcí přenosu (CLA - Carry Lookahead Adder)** - Rychlejší jak **RCA**, výpočet opravdu probíhá paralelně, teoreticky lze sečít dvě jakkoliv dlouhá bitová čísla s konstantním zpožděním $S_i = 4 \log. \text{ členů}$ (a se zpožděním carry bitu $C_i = 3 \log. \text{ členů}$). Reálně je délka sčítaných čísel omezena schopností implementovat více vstupů hradla s ekvivalentním zpožděním jako dvojvstupů, počtem hradel sčítačky a s tím souvisejícím počtem spojů. Pro velká čísla přestává být tudíž praktická (náročná na výrobu, vysoká cena atd.) a je potřeba výpočet rozdělit. [16-bit adder using 4-bit CLA](#)

- z tabulky vstupů do CLA lze odvodit carry bit $C_{i+1} = A_i \text{ and } B_i \text{ or } (A_i \text{ xor } B_i) \text{ and } C_i = G_i \text{ or } P_i \text{ and } C_i$ (G - generate, P - propagate).

For the example provided, the logic for the generate (G) and propagate (P) values are given below.

$$C_1 = G_0 + P_0 \cdot C_0,$$

$$C_2 = G_1 + P_1 \cdot C_1,$$

$$C_3 = G_2 + P_2 \cdot C_2,$$

$$C_4 = G_3 + P_3 \cdot C_3.$$

Substituting C_1 into C_2 , then C_2 into C_3 , then C_3 into C_4 yields the following expanded equations:

$$C_1 = G_0 + P_0 \cdot C_0,$$

$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1,$$

$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2,$$

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3.$$

To determine whether a bit pair will generate a carry, the following logic works:

$$G_i = A_i \cdot B_i.$$

To determine whether a bit pair will propagate a carry, either of the following logic statements work:

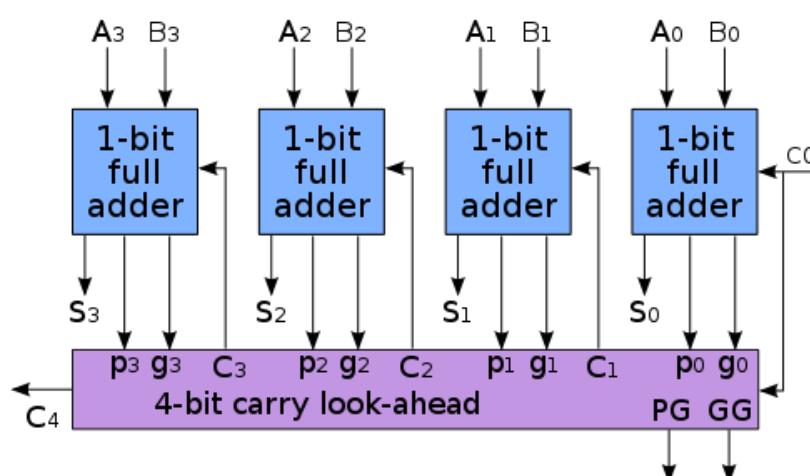
$$P_i = A_i \oplus B_i,$$

$$P_i = A_i + B_i.$$

Z těchto rovnic lze učit, že výpočet C_i bude mít zpoždění **3 log. členů**,

v případě, že máme více vstupní hradla (1. - A xor B, 2. P and C, 3. or všech vstupů).

- výpočet S_i bude probíhat jako $((A_i \text{ xor } B_i) \text{ xor } C_i)$, což znamená zpoždění dalšího log. členu a celkově tedy **4. log členů**.



- Vícebitové sčítáčky s použitím 4 bitových (případně více) CLA sčítáček:**
 - zřetězení CLA sčítáček ve stylu RCA sčítáčky, např. 16 bitová sčítáčka pomocí 4 CLA sčítáček bude mít zpoždění $C_{16} = 4*3 = 12 \text{ log. členů}$

- a zpoždění $S_{15} = 3*3 + 4 = 13$ log. členů (3x carry + poslední výpočet).
- spojením více CLA sčítáček pomocí **LCU (lookahead carry unit)**, které dokáží opět s **konstantní rychlostí** spočítat přenosy mezi jednotlivými CLA sčítáčkami. **C_4, C_8, C_12** a **C_16** jsou dle rovnice níže vypočteny se zpožděním **5 log. členů** a protože **G** a **P** již jsou vypočteny, zabere výpočet **S_i4** až **S_i15** už jen další **3 log. členy** a zpoždění tak bude celkem **8 log. členů**.

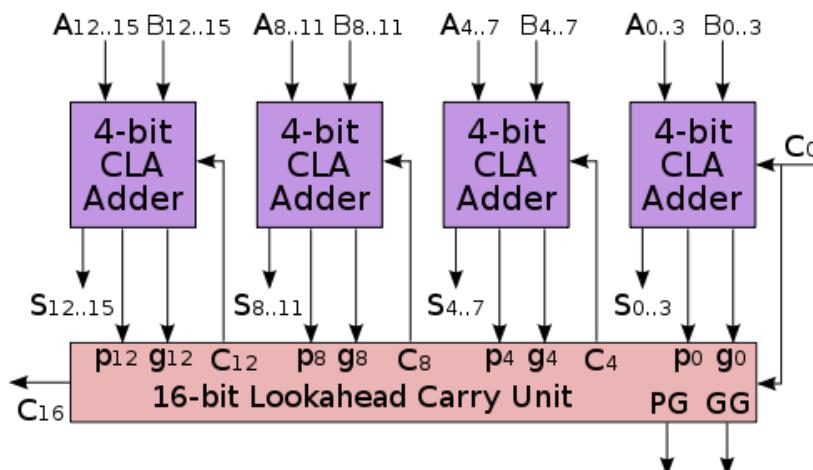
$$PG = P_0 \cdot P_1 \cdot P_2 \cdot P_3,$$

$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1.$$

They can then be used to create a carry-out for that particular 4-bit group:

$$CG = GG + PG \cdot C_{in}.$$

- obdobně jde vytvořit pomocí čtyř 16 bitových **LCU** sčítáček 64 bitovou sčítáčku s dalším zanořením **LCU**.



3. Sekvenční logické obvody (klopné obvody, čítače, registry, stavové automaty – reprezentace a implementace)

Sekvenční logický obvod (SO) - Výstup obvodu závisí **nejen na vstupu**, ale také na **minulých vstupech**. Skládají se z **kombinační části** a **paměťové části**. Paměť v sekvenčním obvodu uchovává **vnitřní (současný) stav**.

- **Asynchronní SO** - Změna vstupní hodnoty hned ovlivní výstup. Reaguje okamžitě.

- **Synchronní SO** - Řízený hodinovým signálem. Až při přechodu hodinového signálu se vstup projeví na výstupu. Reagují v periodických intervalech.
 - **Úrovňové SO** - SO sleduje vstupy po celou dobu hodinového signálu a průběžně na ně reaguje.
 - **Hranové (derivační) SO** - Reaguje na vstupy jen při přechodu hrany (náběžné nebo sestupné).

Klopný obvod

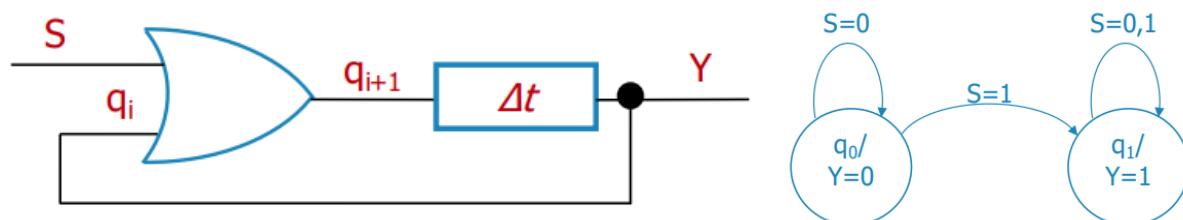
Klopné obvody jsou nejjednodušší sekvenční obvody. Jedná se o obvody, které **skokově** (neuvážujeme zpoždění změny úrovně hradla) přechází mezi **několika diskrétními stavy**.

Klopné obvodu dělíme na:

- **Astabilní** - nemají žádný stabilní stav - neustále kmitají (**osculují**).
 - např. pro generování obdélníkového signálu, hodinového signálu
- **Monostabilní** - mají jeden **stabilní stav**, jedná se o sekvenční obvod, který při spuštění **generuje puls**. Lze si jej představit jako misku a do ní hozený míček.
 - časovače
- **Bistabilní (flip-flop)** - nejpoužívanější typ klopních obvodů, má dva stabilní stavy, mezi kterými lze přepínat (**0 a 1**)
 - je základním stavebním prvkem rychlých **volatileních** pamětí (registry, paměti operandů, čítače, ...)
- **Schmittovy** - slouží k **úpravě tvaru impulzů**. Jeho základní vlastností je **hystereze**. Jeho výstup je závislý nejen na hodnotě vstupu, ale i na jeho **původním stavu**. Hystereze zabraňuje vzniku **zákmitů** výstupního signálu v okolí **střední úrovně** spínání.

Bistabilní klopné obvody

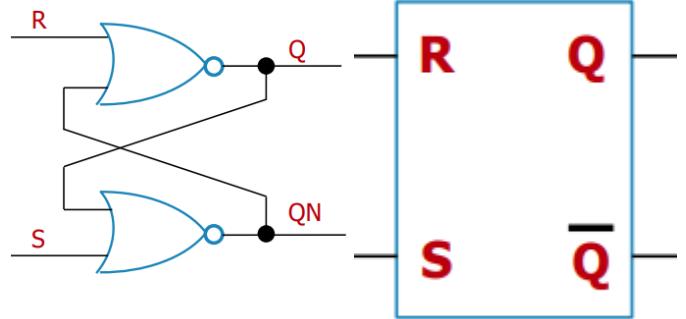
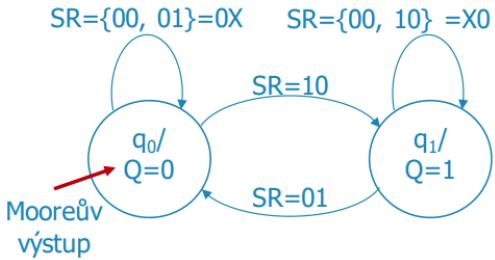
- **SET klopný obvod** - velmi jednoduchý, ale prakticky nepoužitelný. Nelze překlopit zpět ze stavu v **log. 1**. Produkuje **Moorův výstup**.



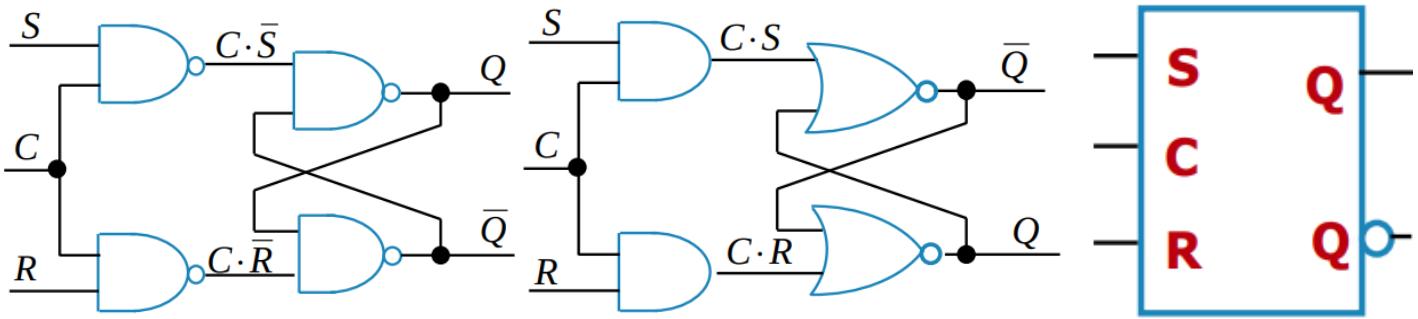
- **R-S klopný obvod (reset-set)**

- Přivedení **log. 1** na vstup **R** (reset) nastaví na výstupu **log. 0**.
- Přivedení **log. 1** na vstup **S** (set) nastaví na výstup **log. 1**.
- Současné přivedení **log. 1** na **S** i **R** je **zakázaná** kombinace, dříve **nešlo určit** v této situaci hodnotu na výstupu, dnes u hradel **NOR** bude na obou **log. 0** (negovaný výstup bude stejný jako přímý). Lze tomu zabránit prioritou jednoho ze vstupů.

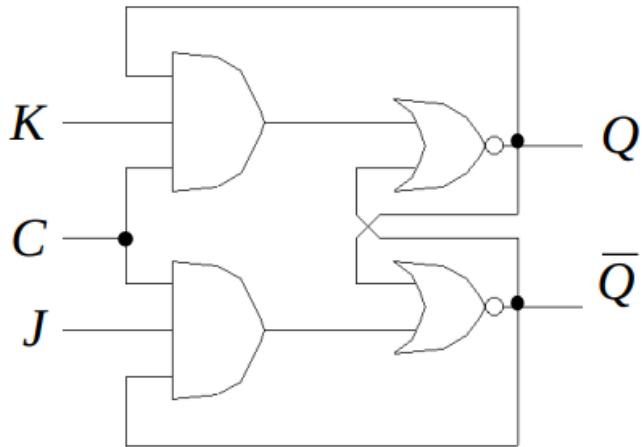
R	S	Q_{n+1}
0	0	Q_n
0	1	1
1	0	0
1	1	X



- **R-S s povolovacím vstupem** - Povolovací vstup **C** (Control, Enable), který povoluje činnost KO – obvod lze **nastavit** či **nulovat**, jen pokud je vstup **C** aktivní (aktivní může být v log. 1 i log. 0, záleží na návrhu).

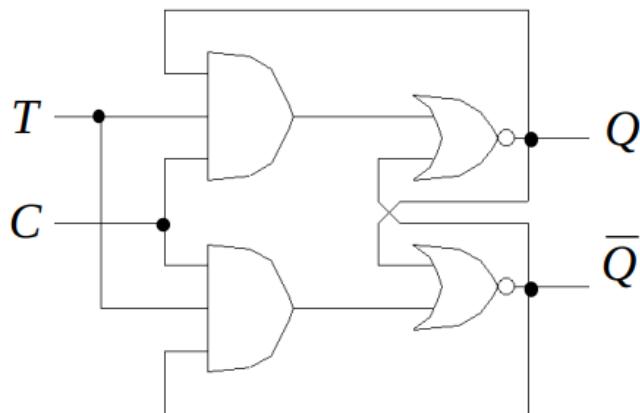


- **J-K klopný obvod (K-J: K odpovídá S, J odpovídá R)** - Zavádí zpětnou vazbu, která eliminuje zakázanou kombinaci. V případě, že jsou oba vstupy v log. 1, dochází ke změně aktuální hodnoty na výstupu (překlop - toggle).

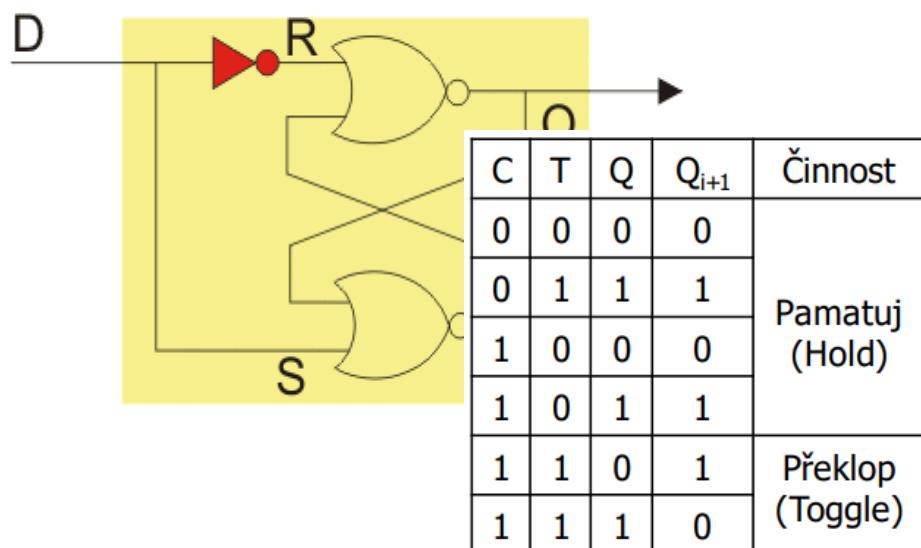


C	J	K	Q	Q_{i+1}	Činnost
0	X	X	0	0	Pamatuj (Hold)
0	X	X	1	1	
1	0	0	0	0	
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	
1	1	0	1	1	Set
1	1	1	0	1	
1	1	1	1	0	Překlop (Toggle)

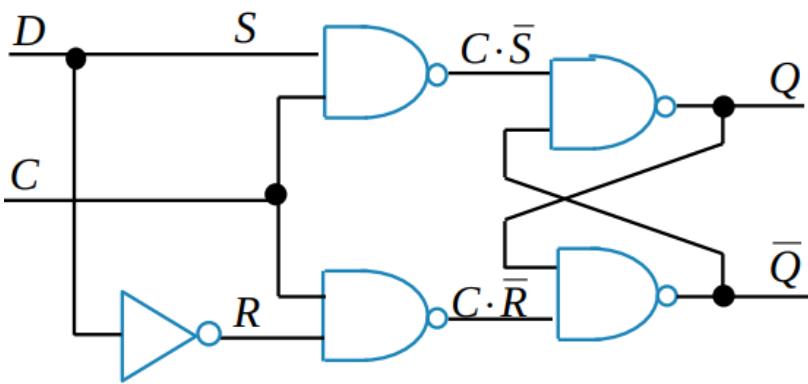
- **T (toggle) klopný obvod** - Jedná se o **J-K** klopný obvod, který na oba vstupy posílá jednu hodnotu **T**. Pokud je **T** v **log. 0**, na výstupu zůstává aktuální hodnota, pokud je na vstupu **log. 1**, dochází k překlopení aktuální hodnoty na výstupu. Držení **log 1.** na vstupu **T** (a případně vstupu **C**) způsobí oscilování, změny log. 1 a log. 0.



- **D (data) klopný obvod** - Jedná se R-S klopný obvod, který posílá na vstup **S** hodnotu **D** a na vstup **R** její negaci **not D**. Nemůže takto vzniknout nepovolený stav (**R** a **S** nikdy nebude současně v log. 1). Na výstup posílá hodnotu na vstupu (vytváří tedy zpoždění).



- **D klopný obvod s povolovacím vstupem** - Narozdíl od jednoduchého D klopného obvodu, zde dochází ke změně hodnoty na výstupu, **jen** pokud je aktivní vstup **C** (pokud je C v log. 0, na výstupu setrvává nastavená hodnota).

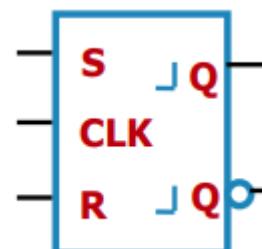
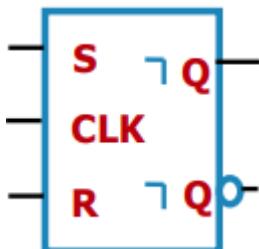


C	D	Q	Q_{i+1}	Činnost
0	X	0	0	Pamatuj (Hold)
0	X	1	1	
1	0	0	0	Ulož 0 (Store 0)
1	0	1	0	
1	1	0	1	Ulož 1 (Store 1)
1	1	1	1	

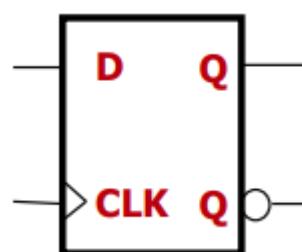
Dvoufázové bistabilní klopné obvody

Využívají se v **synchronních obvodech**, které jsou řízeny hodinami **CLK**. Mohou být typu **Master-Slave** nebo **Derivační**.

- U **Master-Slave** dochází ke změně hodnoty na výstupu **buď** když je **CLK** v **log. 0, nebo v log. 1** _.
- Informace ze vstupů se při **CLK=0** zapisuje do **Master R-S KO** a následně se při **CLK=1** přepisuje do **Slave R-S KO**. Případně opačně.
- různé typy: **RS, JK, T, D**.



- **Derivační** mění hodnotu na výstupu při změně úrovně **CLK**, tj. buď s **nástupnou hranou** nebo se **sestupnou hranou**. (> nástupná, náběžná; < sestupná, doběžná)



Čítače

Jedná se o speciální automaty, které reagují na vstupní impulsy (např. nástupná hrana) přechodem ze stavu do stavu (**přičítají** nebo **odečítají**).

- **Asynchronní čítač** - Neobsahuje synchronizační hodiny, čítání (přechod ze stavu do stavu) je řízeno pouze vstupem.
- **Synchronní čítač** - Je řízen hodinami, k přechodu ze stavu do stavu dochází např při náběžné hraně hodin.

Mohou mít řídící vstupy, které umožňují:

- **UP/DOWN** - specifikovat **směr čítání** (vzestupná/sestupná posloupnost stavů)
- **CE (clock enable)** - pro **CE=0 si pamatuje** aktuální stav, pro **CE=1 čítá**
- **P Enable** - čítač čítá pouze když je v **log. 1**, nemá vliv na **RCO**.
- **T Enable** - čítač čítá pouze když je v **log. 1**

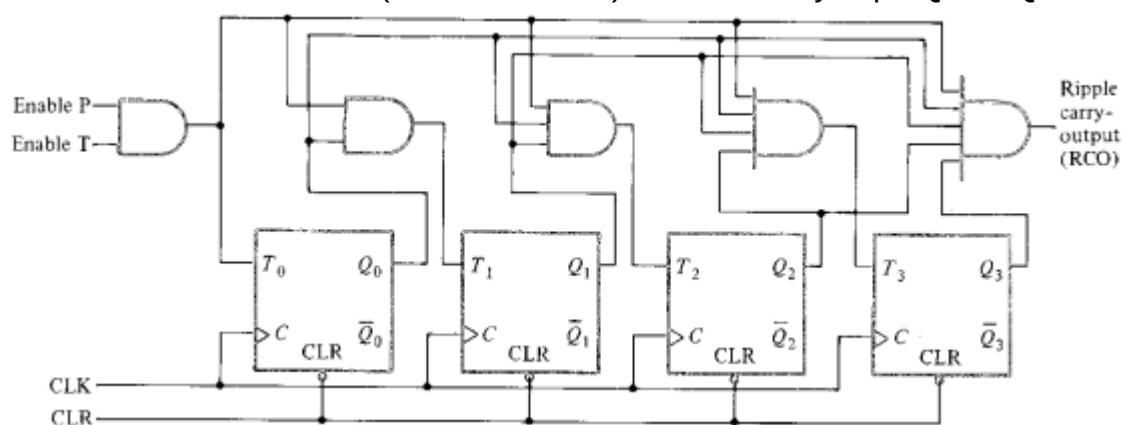
Mají výstup indikující přetečení (podtečení) čítače:

- **RCO** nebo **TC**

Prodloužení doby **čítání/dělícího poměru/vzniku přetečení** se implementuje kaskádovým zapojením čítačů. Čítače jsou zapojeny **vstupem CE** na **výstup RCO (TC)** předchozího čítače. (tři čítače MOD 10 za sebou přetékají jednou za $10 \times 10 \times 10 = 1000$ impulsů)

Čítače řešené T klopným obvodem

- 4 bitový čítač
- při přechodu z **15 do 0** dochází k přetečení (RCO v log. 1)
- aktuální stav čítače (binární hodnotu) lze získat z výstupů **Q0** až **Q3**



Čítač v Grayově kódu

- Grayův kód kóduje po sobě jdoucí čísla tak, že je mezi nimi vždy změna pouze v jediném bitu.
- redukce počtu operací při změně čísla o jedničku
- binární reprezentace (b) na Grayův kód (g):
 - MSB zůstává stejný
 - pro každý bit: $g_i = b_{i+1} \oplus b_i$
 - **1110 bin == 1001 Gray**

Up/Down čítač

Postup: [3-Bit & 4-bit Up/Down Synchronous Counter](#)

- určení tabulky ... Q2 Q1 Q0 (podle počtu bitů) aktuálního stavu - **present state logic**

Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

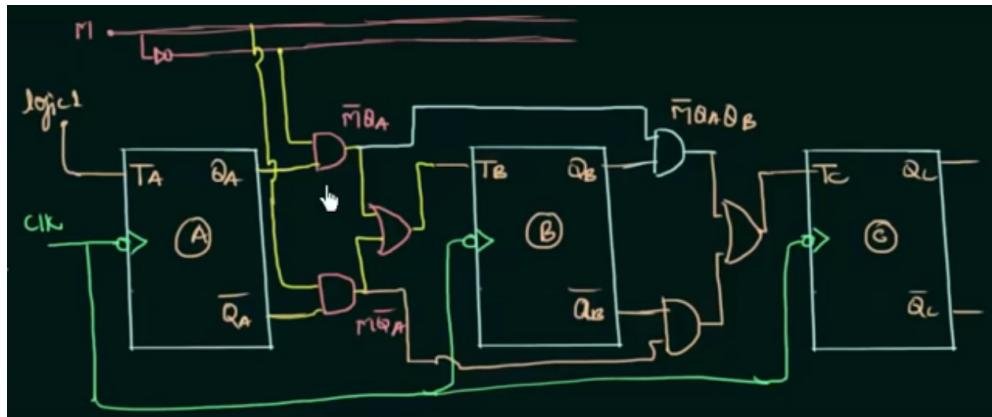
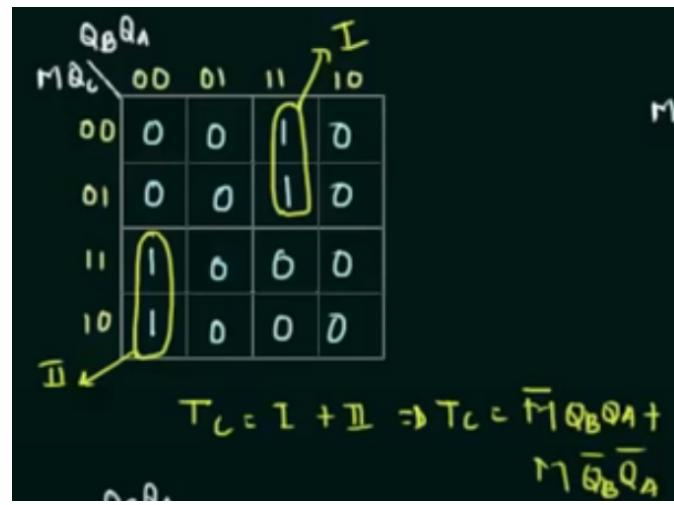
- určení tabulek následujícího stavu (Up - obrázek/Down) - **next state logic**

\bar{Q}_c	\bar{Q}_B	\bar{Q}_A	\bar{Q}_c^+	\bar{Q}_B^+	\bar{Q}_A^+
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

- určení log. úrovní vstupů klopných obvodů (budou se lišit u R-S, J-K, T - na obrázku, D) pro jednotlivé stavy

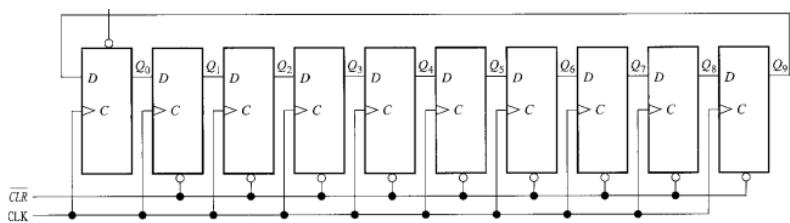
\bar{Q}_c	\bar{Q}_B	\bar{Q}_A	\bar{Q}_c^+	\bar{Q}_B^+	\bar{Q}_A^+	T_c	T_B	T_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

- zjednodušení logiky vstupů klopných obvodů pomocí **Karnaughových** map, pro každý KO jedna mapa
- implementace obvodu na základě určené logiky přechodů s použitím zvoleného klopného obvodu (na obrázku T)



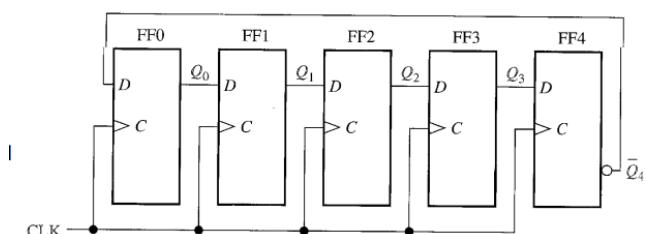
Čítač one-hot (straight counter)

V log. 1 je pouze jeden D klopný obvod, výstup kaskády je přiveden na vstup.



Čítač v Johnsonově kódu

Opět čítá pomocí **D** klopných obvodů, **negovaný** výstup kaskády je přiveden na vstup.



Registry

Umožňují ukládat informaci o určitém počtu bitů. Pro realizaci se obvykle používá klopný obvod typu **D**. KO jsou připojeny ke stejnému zdroji hodinového signálu. Mohou mít asynchronní **CLR** pro nulování, povolení činnosti **CE**. Mají vektor vstupů např. **D31-D0** a vektor výstupů např. **Q31-Q0**, lze je zapisovat jednotlivě, ale obvykle se tak neděje. Reprezentují např. 32-bit číslo, které vstupuje do **ALU** jako **operand**. Dále se používají při konstrukci automatů pro uchování **vnitřního stavu**.

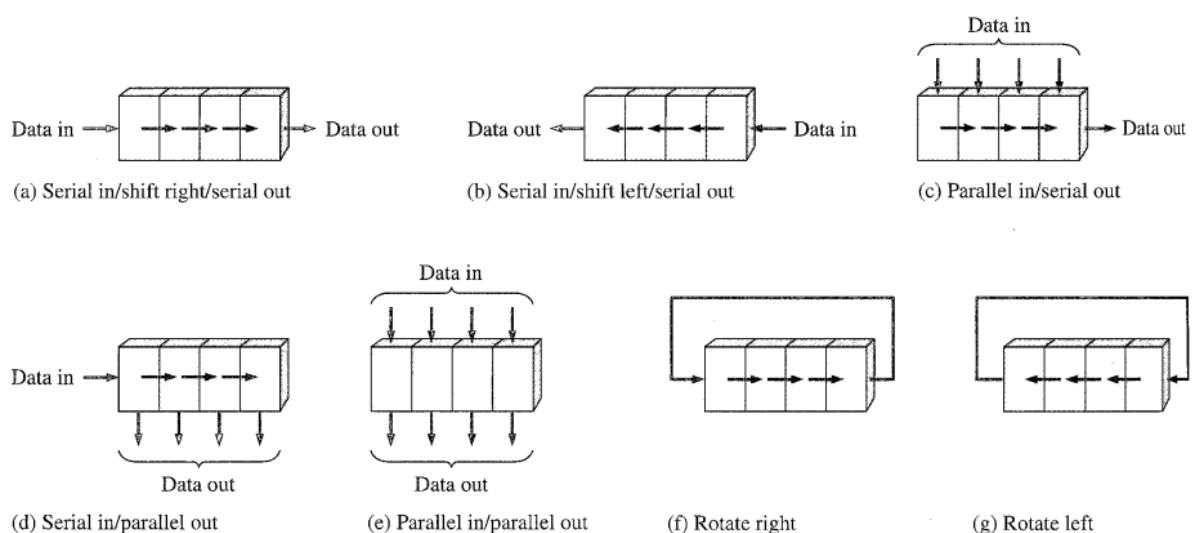
Paměťový registr (latch)

Paměť pro několik bitů.

Posuvný registr (shift)

Synchronní obvod (mají stejný CLK) sestaven z klopných obvodů zapojených do série. Posouvání uložených bitů od jeden bit vlevo nebo vpravo při každém hodinovém impulsu. Existuje mnoho druhů, lze je použít např:

- A. **Master-Slave SPI** - MISO, MOSI,
- B. zápis dat pro odeslání přes **UART**,
- C. čtení obdržených dat z **UART**,
- D. ukládání hodnot operandů,
- E. sekvenční sčítačka**

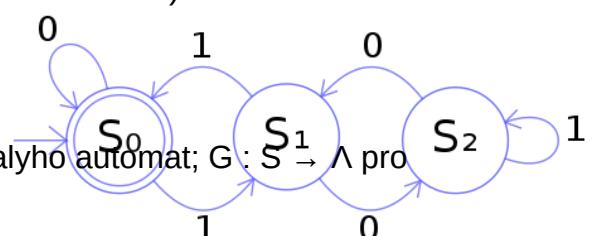


Stavové automaty

Stavový automat - Konečný automat (KA) je šestice:

$M = (S, \Sigma, \Lambda, T, G, s_0)$, kde:

- S je konečná neprázdná množina stavů (vnitřní abeceda)
- Σ je konečná vstupní abeceda
- Λ je konečná výstupní abeceda
- T je přechodová funkce ($T : S \times \Sigma \rightarrow S$)
- G je výstupní funkce ($G : S \times \Sigma \rightarrow \Lambda$ pro Mealyho automat; $G : S \rightarrow \Lambda$ pro Mooreho automat)



Mooreův automat)

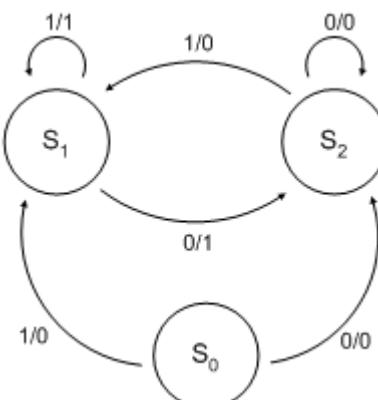
- $s \in S$ je počáteční stav

V elektronice ho tvoří 3 části:

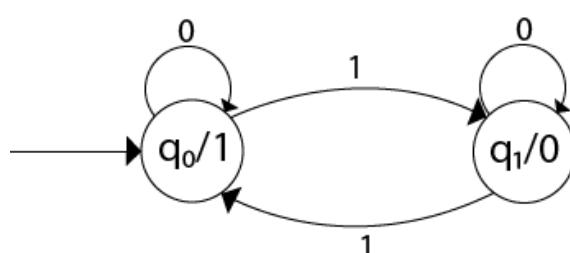
- **Next-state logika** (přechodová funkce) - Kombinační logická síť KLS. Na základě **současného stavu** (paměť) a **vstupu** generuje hodnotu následujícího stavu automatu.
- **Paměť** - Pro **zapamatování si současného stavu**.
- **Výstupy** - Mealy, Moore, kombinace.

Dle výstupní funkce je dělíme na:

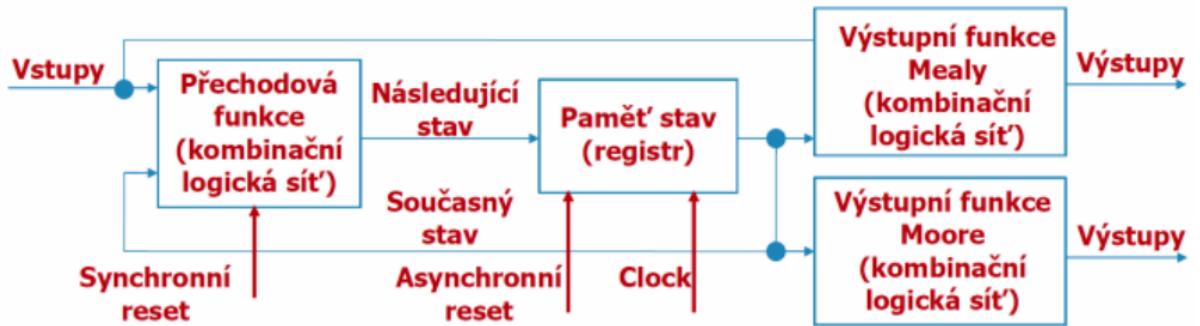
- **Mealyho automat** - Výstup je funkcí jak aktuálního stavu, tak vstupu $Y = f(X, Z)$.
 - značení přechodu: **vstup/výstup**



- **Mooreův automat** - Výstup je funkcí aktuálního stavu automatu - $Y = f(Z)$. Tedy změna na výstupu se projeví až v následujícím stavu.
 - značení stavů: **jméno stavu/výstup**



- **Moore/Mealy automat** - Kombinace mealyho a mooreova automatu.



Odkazy:

- SO: [SEKVENČNÍ LOGICKÉ OBVODY](#)
- Mealyho automat: [Mealyho automat](#)
- Mooreův automat: [Mooreův stroj](#)

4. Hierarchie paměti v počítači (typy a principy pamětí, princip lokality, organizace rychlé vyrovnávací paměti).

Paměť - Uchovává data, které je zde možné zapsat a později zase získat (přečíst).

Dělení pamětí

Hierarchie - Hierarchie existuje z důvodu co nejlepšího **poměru cena/výkon**. Stejně tak je potřeba jak **volatilní**, tak **nevolutilní** paměti. Obecně platí, že čím **blíže** procesoru paměť je, tím je **rychlý**, ale také **dražší** (náročnější na výrobu) a má **menší kapacitu**.

- **Primární/vnitřní**
 - uvnitř procesoru: **registry, cache** (RVP, obvykle několik úrovní),
 - mimo procesor: **RAM**
- **Sekundární/hlavní** (uvnitř PC) - **HDD, SSD**.
- **Terciární/vnější** (mimo PC) - **CD, DVD, Flash disk**.

Fyzikální princip - opět výhody a nevýhody cena/volatilita/rychlost:

- polovodičové - bipolární, unipolární, unipolární s floating gate - **SSD**
- magnetické - **diskety, HDD, pásky**,
- optické - **DVD, CD, Blue Ray**,
- magnetoopické
- molekulární

Stálosti obsahu:

- **Volatilní** - Po odpojení paměťové buňky od el. energie se data ztratí (RAM),
- **Nevolatilní** - Data setrvávají i po odpojení paměti z el. sítě.

Doba uchování informace:

- **Statická** - Drží data libovolně dlouho (dokud jsou připojeny k el. energii).
Velká plocha na čipu (klopné obvody), **rychlá, drahá, malá kapacita** - používá se pro registry a RVP
- **Dynamická** - Data je třeba po určité době (u DRAM v řádu milisekund) obnovovat, jinak se ztratí. **Pomalejší, levnější** (oproti statické), **menší plocha na čipu** (tvořena tranzistorem a kondenzátorem) - používá se pro DRAM paměti.

Přístup k datům:

- **Libovolný** (RAM - Random Access Memory) - přístupová doba **nezávisí** na umístění paměťové položky.
- **Sériová** (SAM - Serial Access Memory) - přístupová doba **závisí** na umístění v paměti, respektive na aktuální pozici čtecí hlavy (pásy)
- **Smíšený** - kombinace výše popsaných přístupů. Např. HDD s více záznamovými povrchy. Výběr povrchu je libovolný, natočení hlavy je sekvenční.

Možnost přístupu/měnitelnost:

- **ROM (Read Only Memory)** - Pouze čtení
- **PROM (Programmable ROM)** - Lze **jednou** zapsat data a poté pouze číst
- **EPROM (Erasable PROM)** - Pro vymazání je potřeba použít externí proces, například pomocí **UV záření**.
- **EEPROM (Electrically EEPROM)** - Lze **vymazat elektronicky**
- **RWM (Read Write Memory)** - Lze **číst i zapsat** do ní (SRAM, DRAM)

Princip lokality

Aplikace pracuje obvykle pouze s **malou částí paměťového prostoru** (data programu nebo samotný kód programu).

- **Časová lokalita** - Pokud procesor používá nějakou položku v paměti, je vysoká pravděpodobnost, že ji bude používat **znovu**. To znamená je vhodné položku uložit, co nejblíže k procesoru.
- **Prostorová lokalita** - Pokud procesor pracuje s nějakou položkou v paměti, potom položky, které jsou umístěny v paměti v **blízkosti této položky**, budu s vysokou pravděpodobností **také použity** (aspoň u dobré napsaného kódu). To znamená, že je vhodné okolní položky uložit co nejblíže procesoru, aby toto očekávané čtení/zápis mohlo proběhnout rychleji.

Parametry pamětí

- **Kapacita** - Udává množství dat, která paměť dokáže uložit. Udává se ve tvaru **respektujícím organizaci paměti**, tedy jako součin počtu paměťových míst a délky paměťového místa, tj. $N \times n$ bitů, např. **16K x 1 bit**
- **Přístupová doba** - doba od **zahájení čtení** (tj. udáním adresy paměťového místa a povelu R) po **získání obsahu** paměťového místa.
- **Přenosová rychlosť** - Počet **bitů/bytů**, které paměť přenese za **jednotku času**, např. 1 Gb/s.
- **Chybovost** - Počet chyb za určitý čas.
- **Poruchovost** - Střední doba mezi poruchami (MTBF - mean time between failure).

Rychlá vyrovnávací paměť (RVP)

Anglicky **cache** - SRAM blízko procesoru, pomáhá rychlejšímu čtení/zápisu dat, než jaké by bylo použitím pouze operační paměti (RAM - pomalejší DRAM). V procesorech často rozdělena do několika vrstev - L1, L2 pro každé jádro a L3 sdílená mezi jádry. Hierarchické členění pamětí se snaží eliminovat rozdíl mezi rychlostí procesoru a rychlostí operačních pamětí.

Čtení/zápis	Počet cyklů
-------------	-------------

Registr	1 (i méně při pipelining)
RVP L1	cca 3
RVP L2	cca 14
Hlavní paměť (RAM)	cca 240

RVP je rozdělena do **bloků** o konstantní velikosti, která ideálně odpovídá velikosti bloků v hlavní paměti (RAM), což umožňuje jednodušší přesun dat (po blocích) z RAM do RVP. V RAM je ale paměťových bloků **řádově více** než v RVP, takže musí existovat mechanismy zajišťující, aby v RVP byly **potřebné bloky**. Tyto mechanismy vybírají bloky, které budou do RVP nově přidány a bloky, které v důsledku budou odstraněny. Musí pracovat s velmi **vysokou pravděpodobností úspěchu**, jinak by paměťová hierarchie zpomalovala přístup k datům.

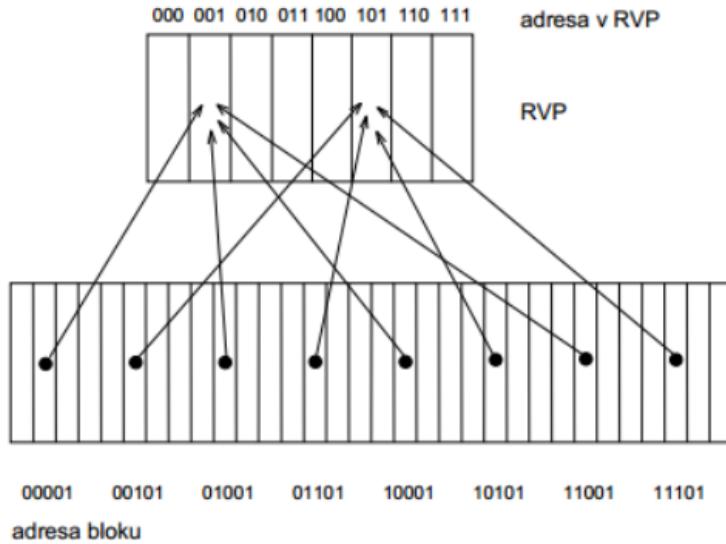
- **Hit Rate** - udává pravděpodobnost nalezení dat v RVP, v praxi **95-99%**.
- **Miss Rate** ($1 - \text{Miss Rate}$) - pravděpodobnost, že data **nejsou** v cache nalezena. V tomto případě je nutné potřebný blok s daty načíst z hlavní paměti, což obnáší uvolnění místa v RVP, vyhledání bloku v RAM a přenos dat. Tato doba se označuje jako ztrátová doba (**miss penalty**).
- **Přístupová doba** - doba potřebná k nalezení bloku v RVP, blok dat musí být v RVP.
- **Ztrátová doba** - doba, za kterou se musí popřípadě uvolnit místo v cache, nalézt data v hlavní paměti a nahrát je do cache.
- Doba čtení:
 - **Hit** = přístupová doba
 - **Miss** = přístupová doba + ztrátová doba (tato doba je delší, než v případě **nepoužití** RVP, musí k ní docházet minimálně)

Organizace RVP

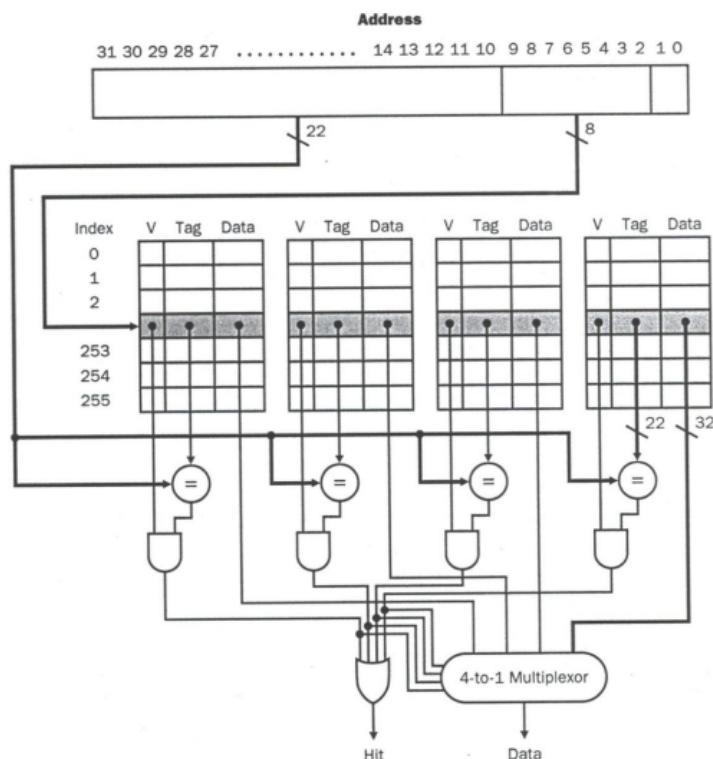
Je dána zvoleným mapováním. Účel organizace RVP je především co nejvíce snížit **Miss Rate**. Organizace určuje jak jsou bloky z hlavní paměti přiřazovány/mapovány do RVP.

- **Přímé mapování** - Do RVP se mapují bloky na základě adres. Protože adresový prostor RVP je několikanásobně **menší**, než adresový prostor hlavní paměti, **maskují** se u adres hlavní paměti **horní bity**, tak aby délka adresy odpovídala délce adresy v RVP. To znamená, že bloky s adresou se stejnými spodními se mapují na jeden blok v RVP a **nemohou** tam tak být umístěny **současně**. Jedná se o jednoduchý koncept, který ale může způsobovat velký počet výpadků (procesor současně pracuje s dvěma bloky, které mají stejnou adresu v RVP).
 - např. pro RVP s 2^{10} bloky a pamětí s 2^{30} bloky bude pravděpodobnost při náhodném výběru bloku $2^{10}/2^{30} = 1/2^{20}$ zaokrouhleně $1/1\ 000\ 000$, tedy **0.0001%**.

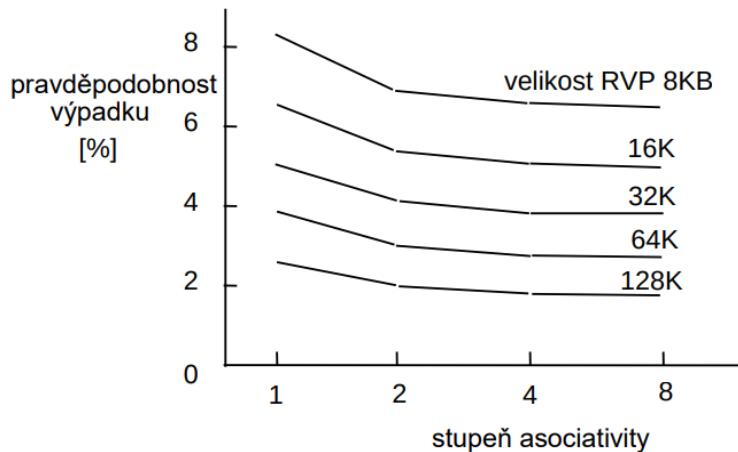
- díky časové a prostorové lokalitě není ale přístup do paměti náhodný, proto je v praxi pravděpodobnost **Hit** tohoto řešení výrazně vyšší, okolo **90%**.



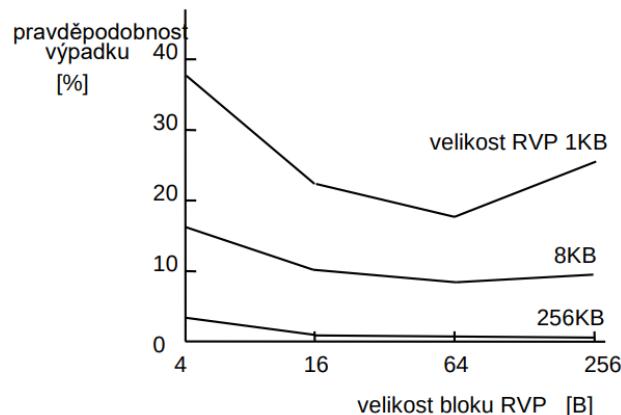
- **Více cestné mapování** - umožňuje do RVP uložit současně více bloků, které mají stejné spodní bity adresy (ukazatel). Adresový prostor RVP je tak rozdělen do více tabulek (2 pro dvoucestné, 4 pro čtyřcestné, ...), u záznamů v těchto tabulkách navíc musí být uložena zbylá část (maskovaná) adresy - **příznak**, aby se mohl vybrat ten správný záznam pro čtení/zápis. To znamená, že více cestná RVP o stejně **kapacitě dat** jako jednocestná, bude vyžadovat větší celkovou kapacitu a samozřejmě logiku (**komparátor**), která bude provádět **výběr správného záznamu**.



- **Plně asociativní mapování** - K této úrovni lze dospět zvyšováním stupně asociativity, až je tento stupeň roven počtu ukládaných záznamů v RVP (tj. nepoužívá se už adresa bloků RVP). Poté může být libovolný záznam uložen kdekoli v RVP a s ním **celá** jeho adresa (do RAM) jako příznak. Tento způsob uložení ale implikuje to, že musí být zajištěno **souběžné porovnání** adres všech uložených bloků v RVP, aby bylo možné **rychle** najít ten požadovaný. To je ale technologicky náročné (pokud vůbec možné pro velké počty bloků), drahé a v praxi **nepoužitelné**.



Pravděpodobnost výpadku lze samozřejmě nejlépe snížit **zvýšením kapacity RVP**, ale také lze snížit určením vhodné **velikosti bloku**.



U vícecestných RVP se musí také řešit problém výběru oběti, pokud je RVP pro daný ukazatel (adresu) plná (všechny bloky jsou označeny jako **validní**). Používají se metody jako náhodný výběr, LRU, MRU, FIFO atd. Obecně tyto metody vyžadují další logiku (obvody).

- **Skupinově asociativní mapování** - Kompromis mezi přímým a plně asociativním mapováním. Paměť je rozdělena do skupin, kde každá skupina obsahuje stejný počet bloků. Adresa skupiny je vyhledávána přímo, blok v ní pak asociativně.
- **Sektorové mapování** - Hlavní paměť je rozdělena do sektorů, které mají několik bloků. Cache je také rozdělena do sektorů o několika blocích. Sektor hlavní paměti může být uložen do libovolného sektoru v cache, ale bloky musí být v sektoru shodné. Při přenosu bloku do cache jsou tedy staré bloky označeny

za neplatné (příznakovým bitem) a je tam blok vložen.

Konzistence dat v RVP

Při zápisu dat do bloku RVP (nejblíž CPU), ztratí bloku nižších RVP a hlavní paměti platnost a **nesmí** se již použít, musí se nějak označit. Problém je zejména zjevný u více jádrových procesorů (např. každé jádro má vlastní L1 a L2 cache, sdílí L3).

Zápis do L1 jednoho jádra musí případně nevalidovat blok v L1 druhého jádra, které si jej musí znova načíst (problém souběžného přístupu řeší programátor).

Metody zaručení konzistence:

- **Write-Through** (přímý zápis) - Při přímém zápisu do RVP se zapisuje okamžitě i do bloku v hlavní paměti. Tento princip je jednoduchý na realizaci, ale při častém zápisu je pomalý (musí se často čekat na pomalou RAM). Lze poté rozdělit:
 - **Write-Through with Write Allocate** - Write-through, pokud ale nejsou zapisovaná data v cache dojde k jejich načtení.
 - **Write-Through with no Write Allocate** - Nedojde k načtení zapisovaných dat do cache.
- **Write-Back** (zpětný zápis) - K úpravě (zapsání) bloku do nižší úrovně paměti (hlavní paměti) dochází až když je blok z RVP **odstraněn** (odsunut). Neefektivní protože k zápisu dochází **vždy**, i když **nedošlo** ke změně (musí se čekat). Proto se bloky označují příznakem změny (**dirty bit**) a zápis pak probíhá následovně:
 - **Flagged Write Back** - Zápis do paměti se uskuteční až při uvolnění bloku z cache, ale **pouze** u těch bloků, které byly modifikovány (mají **nastavený dirty bit**).
 - **Flagged Register Write Back** - Opět jsou ukládány pouze modifikované bloky, ale ne přímo, jsou prvně zapsány do pomocného rychlého bloku - zápis je odložen a je možné tedy číst bez čekání. **Nejrychlejší**, ale **nejnáročnější** (nejdražší) způsob.
- **Write-Buffer** (zápis s mezipamětí) - rozšiřuje metodu **Write-Back** o to, že při vybrání oběti (bloku, který bude odsunut z RVP s nastaveným dirty bitem), se data bloku přesunou do mezipaměti pro zápis, ze které budou zapsány, až nebudou požadavky na čtení z hlavní paměti. Nemusí se tak čekat na zápis dat, která jsou odsouvána z RVP, předtím, než budou nahrazena novým blokem.

5. Vestavěné systémy (mikrokontrolér, periferie, rozhraní, převodníky).

Evropské strategické iniciativy definuje **embedded systems** jako kombinaci hardwaru a softwaru, jejímž smyslem je řídit externí proces, zařízení nebo systém.

Vestavěný (Vestavný/Embedded) systém je jednoúčelový počítač, který je zabudován do zařízení, které ovládá. Na rozdíl od univerzálních počítačů jsou vestavěné systémy **specializované** (nikoliv součástkami) a mají **předem určenou činnost**, kterou řídí. Např. výdej nápojů v automatu, volba pracího režimu a samotná činnost praní v pračce, automobil tesla atd. Snaží se o:

- **Autonomie** - Funkčnost bez lidského zásahu.
- **Reaktivnost** - Odezva na podnět v reálném čase.
- **Kritičnost** - Vliv odchylek na normální chování na bezpečné plnění úlohy.

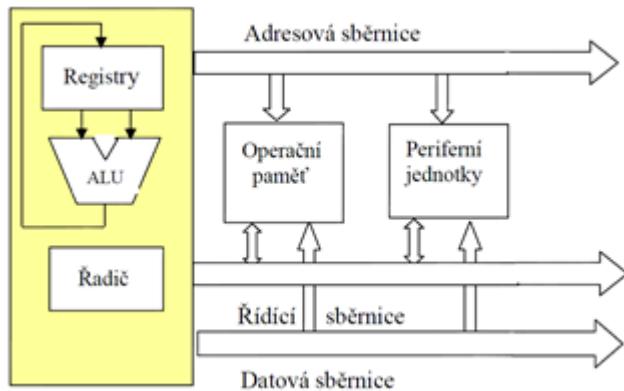
Vlastnosti vestavných systémů:

- Omezená množina aplikačních systémů (vykonává pouze jeden úkol).
 - Často pouze jeden program na celý život.
- Často zpracovává fyzikálních veličin.
- Měly by být spolehlivé, bezpečné, zabezpečené a efektivní.
- Hlavní interakce nemusí být s člověkem.
- Ideálně by člověk neměl ani přemýšlet nad tím, že pracuje s počítačem.

I když je většinou funkce vestavěného systému **specializovaná**, nechceme pro každý systém vyvíjet **nový** integrovaný obvod např. s automatem, který by popisoval jeho činnost. Bylo by to časově a hlavně **finančně** náročné, navíc by poté nešlo provádět případné **změny** (i když v některých extrémních případech ano - optimalizace). Proto je dnes základem téměř každého vestavěného systému **mikrokontrolér** a na něj jsou napojeny dle potřeby periferní zařízení (senzory, display, reproduktor, ...), které jsou opět vyráběny ve velkých počtech. Samozřejmě existuje nespočet mikrokontrolérů, které se liší především cenou, výkonem, spotřebou, pamětí, a je na návrháři vestavěného systému, aby vybral pro účel navrhovaného systému vhodný. Stejně tak to platí o různá periferní zařízení.

Mikrokontrolér (MCU)

Integrovaný obvod implementující kompletní počítač (ALU, paměť, síťové rozhraní, ..), který lze naprogramovat např. v jazyce C (nemusí být moc výkonný). Navíc od běžných procesorů často mikrokontroléry obsahují např. AD převodník, DA



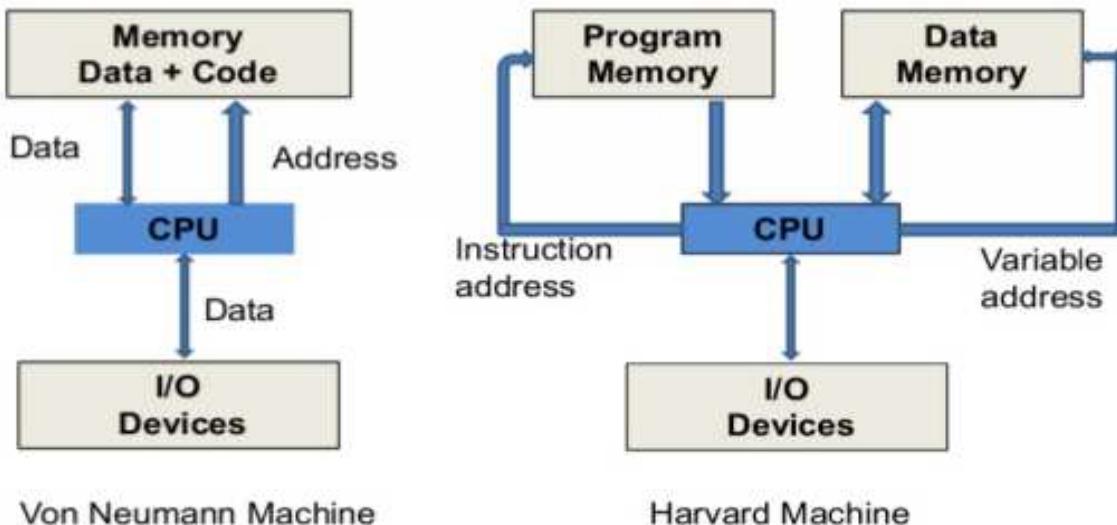
převodník, programovatelné časovače atd. Jsou hlavní komponentou vestavných systémů. Dle architektury je dělíme na:

Von Neumannova architektura MCU

- **Společná paměť pro data i program,**
- **flexibilnější** pro měnící se aplikace,
- umožňuje **samomodifikující** se kód,
- instrukce (program) i operandy (data) se musí z paměti vybírat **postupně**, sběrnice je úzkým hrdlem. Je **pomalejší** než Harvardská.
- Jednodušší implementace, není třeba rozlišovat čtení dat od čtení programu.

Harvardská architektura MCU

- **Paměť programu a dat jsou odděleny** (dvojí adresový prostor a dvě paměti),
- kód může být uložen v paměti jiného typu (FLASH) než data (RAM),
- Umožňuje použít **rozdílnou velikost buňky** paměti – např. 8 bitů (1 byte) pro data a 18 bitů pro instrukci (podle délky instrukce, tak aby jedno čtení načetlo vždy celou instrukci),
- Umožňuje v jednom taktu získat instrukci a **zároveň** její operandy - rychlejší než Von Neumann.



Instrukční sada CISC (Complex Instruction Set Computing)

- Složitá instrukční sada.
- Mnoho typů instrukcí s mnoha variantami a mnoha adresovacími režimy (jedna instrukce může kombinovat více elementárních operací)
- Typicky memory-to-memory instrukce (výběr operandů a uložení výsledků je součástí instrukce).
- instrukce nemusí mít stejnou délku - náročnější na dekódování
- provedení instrukce trvá více taktů
- nepoužívá mnoho registrů, někdy pouze **střadač**.
- Na čipu dominuje logika pro implementaci instrukcí.
- procesory Intel.

Instrukční sada RISC (Reduced Instruction Set Computer)

- Redukovaná instrukční sada.
- Oproti CISC malé množství instrukcí, které provádí pouze **elementární** operace, časté instrukce jsou LOAD a STORE
- všechny instrukce jsou stejně dlouhé (stejný počet bitů v paměti) a typicky trvají **jeden** takt
- využívá velké množství registrů
- na čipu dominují registry (paměť) oproti obvodům pro vykonávání instrukcí
- procesory s touto architekturou mají menší spotřebu, dnes začínají převládat
- Apple Silicon

Struktura MCU

- **Procesor**
- **Operační paměť** - RAM
- **Paměť programu** - EEPROM, PROM, ROM, FLASH, ...
- **Oscilátor** - Zdroj hodinového signálu (RC/Krystal)
- **Vstupně výstupní rozhraní (GPIO - general purpose input output)** - AD/DA převodník, paralelní/sériové porty, Ethernet... Obvykle jeden port lze

využít k více účelům, aby nemělo MCU zbytečně velký počet výstupů.

Periferie MCU

- **Časovač** - Měření času, např. RTC (real time clock), **TODO**
- **Hodiny** - přesnější zdroj hodinového signálu (**krytal**)
- **Čítač** - Viz otázka 3
- **Watchdog** - Stará se o to aby nedošlo k "zaseknutí" MCU při chybě. Pokud mu nepřijde pravidelné upozornění, tak resetuje celý MCU.
- **FPGA** - Programovatelné hradlové pole
- **Řadič přerušení** - klávesnice
- **senzory**

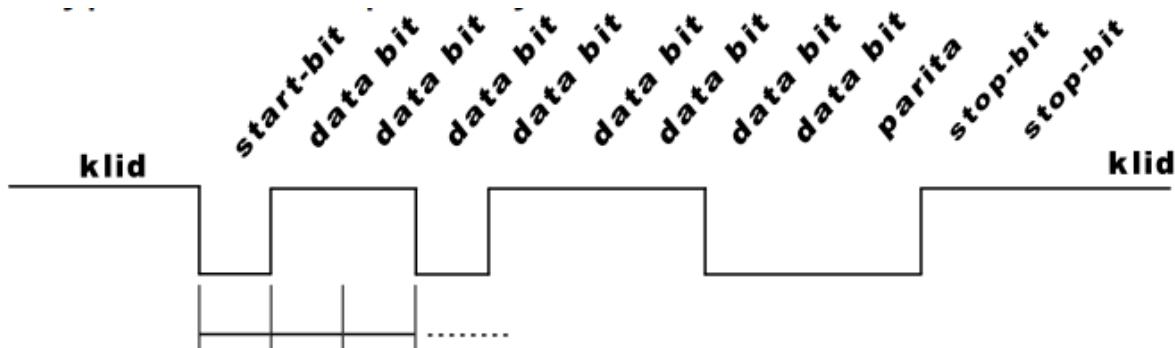
Sériové rozhraní

Slouží pro komunikaci s MCU. Komunikace probíhá po jednotlivých bitech (**sekvenčně**). Bity jsou přenášeny po jediném vodič jeden za druhým. Je třeba jednoznačně určit, v kterém okamžiku je na datovém vodiči hodnota kterého bitu.

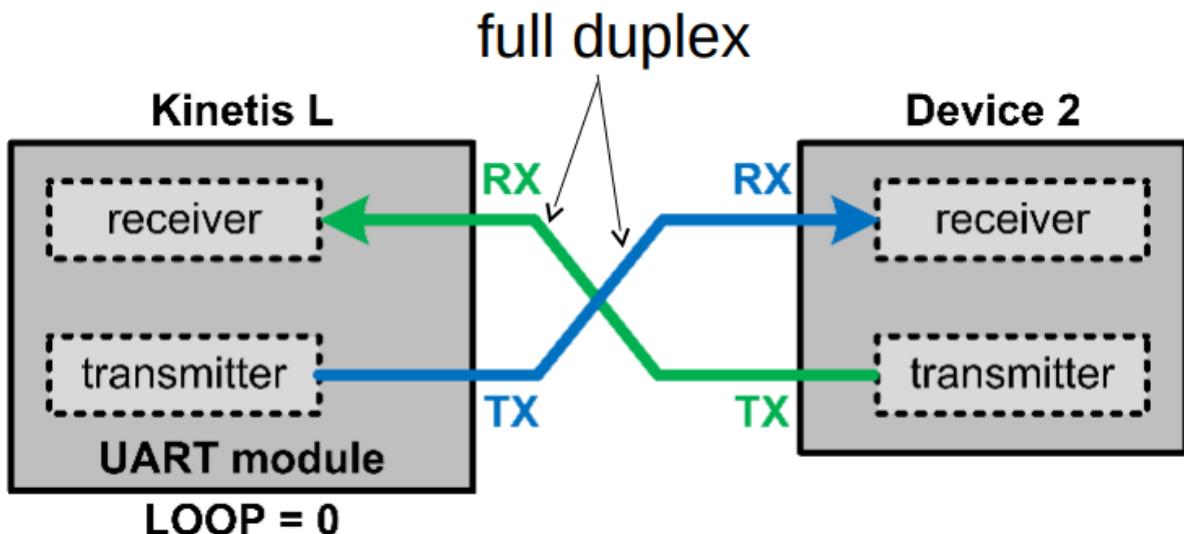
- **Synchronní přenos**: Spolu s daty je přenášen i hodinový signál, který určuje, kdy lze číst další bit. (nutné jsou minimálně 2 vodiče)
- **Asynchronní přenos**: hodinový signál není přenášen, přijímač i vysílač si jej generují sami (musí mít smluvený **baud rate**). Je nutné zajistit dostatečnou přesnost hodinového signálu a jeho synchronizaci.
 - baud rate (Bd)
 - počet symbolů (bitů, bytů, případně jiných přenášených prvků) přenesených za sekundu
 - u sériové komunikace například 1 baud = 1 bit/s

UART (Universal asynchronous receiver-transmitter)

Zařízení umožňující asynchronní komunikaci. Synchronizace je řešena pomocí **start bitu**, který je na začátku každého přenášeného rámce. Jedná se o přechod z klidové úrovni **log. 1** do **log. 0**. Přijímač podle něj **synchronizuje** hodiny, **za** tímto bitem již následují bity datové (8 bitů). Po datových bytech může následovat **paritní bit** a **jeden** nebo **dva stop bity** (vrátí linku opět do klidové úrovni **log. 1**). Přenos je tedy **start bit - LSB - ... - MSB - (paritní bit) - stop bit - (stop bit)**. Příklad na obrázku přenáší hodnotu **0x3B = 0b00111011**.



Komunikace u UART je **většinou plně duplexní** (**full-duplex**), může být ale i **half-duplex** nebo pouze **simplex** (pokud přijímající stanice neobsahuje vysílač).



SPI (Serial Peripheral Interface)

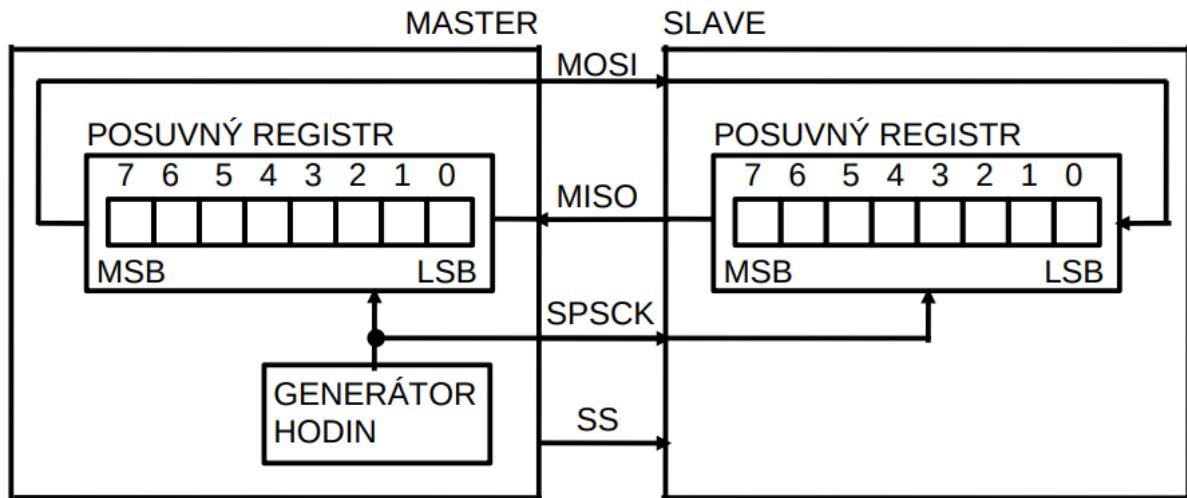
Jedná se **synchronní sériové** rozhraní typu **Master-Slave**, které je **plně duplexní**.

V každém okamžiku **vždy** probíhá přenos **oběma** směry. Levná varianta pro připojování periferních zařízení, lze pomocí SPI realizovat ale i sběrnici.

- **Master**: na SPI sběrnici je vždy **jediný**. Je **zdrojem** hodinového signálu, **iniciuje** a **řídí** komunikaci. Obvykle se jedná o **MCU**.
- **Slave**: ke sběrnici může být připojeno **více** zařízení typu slave, **Master určuje** se kterým v daný okamžik komunikuje.

Vodiče, po kterých probíhá přenos, nebo jej řídí, jsou následující:

- **MISO** (Master In Slave Out) - vodič, po kterém jsou přenášena data od **Slave (zapisuje)** k **Master (čte)**.
- **MOSI** (Master Out Slave In) - vodič, po kterém jsou přenášena data of **Master (zapisuje)** k **Slave (čte)**
- **SPSCK** (SPI Serial Clock) - vodič po kterém je přenášen hodinový signál. Na zařízení Master je to vždy **výstup** a na zařízení Slave **vstup**.
- **SS** (Slave Select) - vodič, který vybírá Slave pro komunikaci. Na zařízení Master je **výstupem**. V **log. 1** je neaktivní (stejně jako UART), v **log. 0** je **aktivní**. V log. 0 musí být po **celou** dobu přenosu. V daném okamžiku může **Master** takto aktivovat **maximálně jeden** modul **Slave**, se kterým chce komunikovat. Zařízení, které mají SS v **log. 1** nezasahují do probíhající komunikace.

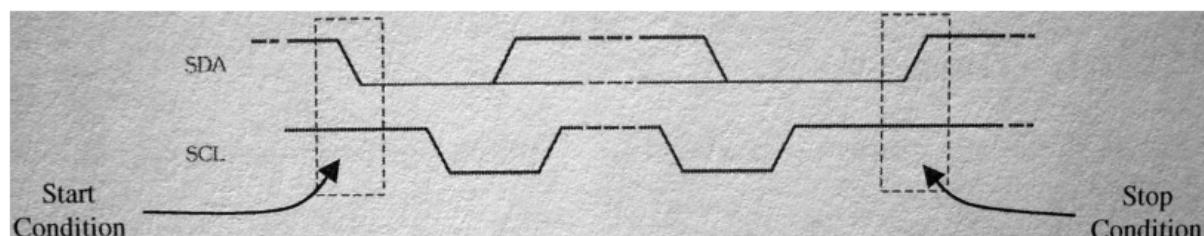


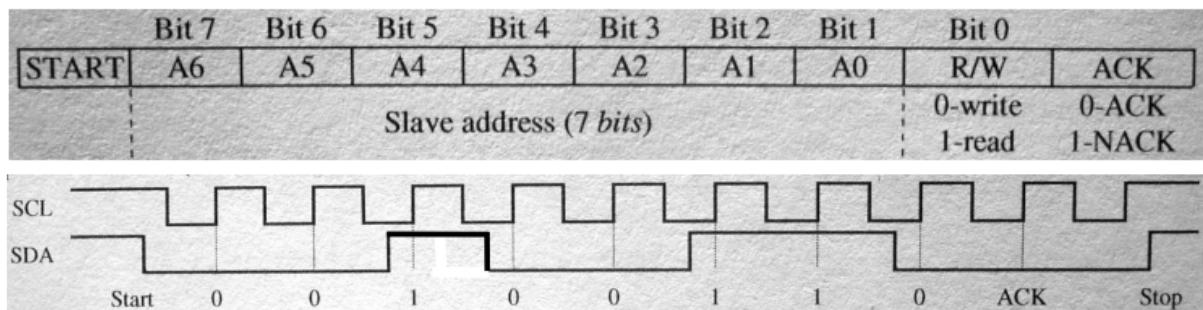
I2C

Jedná se o **synchronní sériové** rozhraní, které funguje na principu **Master-Slave** komunikující způsobem **half-duplex**. Rozhraní tvoří dva vodiče: **SDA** (Serial Data - vodič po kterém jsou přenášena data) a **SCL** (Serial Clock - vodič nesoucí synchronizační signál - hodiny). Oba vodiče jsou zapojeny přes **pull up** rezistory k **Vdd** (log. 1), to znamená v klidu jsou oba vodiče v **log. 1**. Přenos dat probíhá následovně:

- **Start condition** (zahájení komunikace): komunikaci vždy zahajuje Master (obvykle MCU) při **SCL v log. 1** změnou úrovně **SDA** z **log. 1** na **log. 0** ($1 \rightarrow 0$).
- **Komunikace**: Data jsou **vzorkována** z vodiče **SDA** s **náběžnou** hranou na vodiči **SCL**, změna úrovně na vodiči **SDA** musí být tedy prováděna, když je **SCL v log. 0**.
- **Stop condition** (ukončení komunikace): Master na vodiči **SDA** generuje hranu z **log. 0** do **log. 1** ($0 \rightarrow 1$). **SCL** musí být v **log. 1**, jinak by šlo pouze o změnu dat.

Standardně se v datovém rámci přenáší **1 byte** užitečných dat (**po bitech**), každý rámec je zakončen s **ACK** (potvrzení Slave, že data obdržel). Volbu komunikujícího zařízení (Slave) Master realizuje pomocí **adresy** na začátku komunikace v **1. adresovém rámci**. Tato adresa má **7 bitů** (na **I2C sběrnici** tak může být připojeno až **128** zařízení Slave - Master je vždy **jeden**), **osmý** bit určuje, jestli půjde o **čtení** nebo **zápis**.





Paralelní rozhraní

Slouží pro komunikaci komponentů MCU. Je současně posíláno více bitů po více vodičích. Tento způsob může být rychlejší, ale musí se brát ohled na interference paralelních vodičů (magnetická indukce).

<https://electronics.stackexchange.com/questions/21675/what-should-i-know-about-interference-between-wires-in-a-multi-conductor-cable>

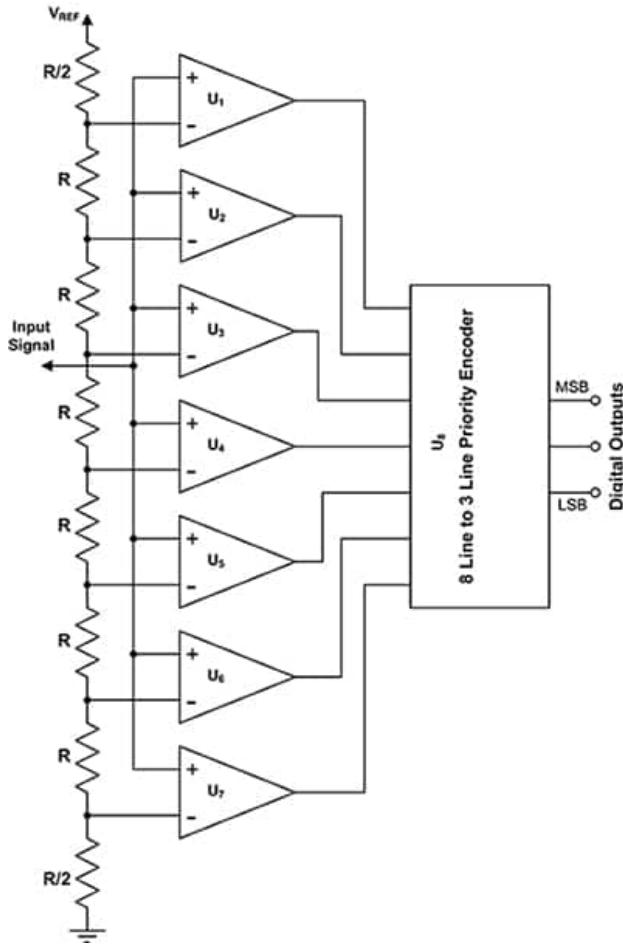
Převodníky

Umožňují komunikaci se světem, který je analogový. Slouží většinou k získávání informací z různých senzorů, které převádí měřené veličiny na napětí a toto napětí se poté převádí pomocí **AD** převodníku na binární hodnotu, kterou dokáže interpretovat MCU. U **DA** převodníku pak MCU pomocí binární hodnoty může vysílat potřebné napětí na výstupu, které např. nastavuje teplotou.

AD převodník (analog-to-digital converter)

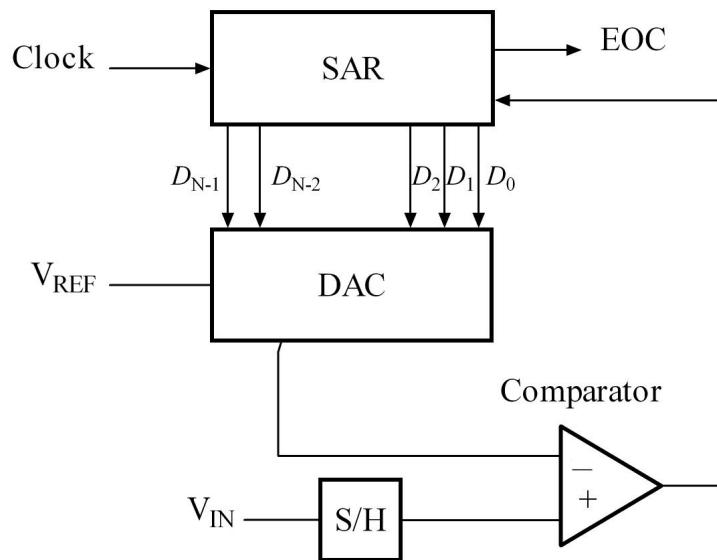
Převádí **analogový** (spojitý) signál na **digitální** (diskrétní - **binární číslo**, které dokáže interpretovat **MCU**). Naměřené hodnoty MCU ale dostává v rozsahu **0 až 2^N-1** a musí je převést na **odpovídající** hodnotu napětí. Existuje více druhů AD převodníků, dva základní jsou:

- **Flash ADC** (direct-conversion ADC): Jedná se pouze o **kombinační** obvod, převod probíhá **paralelně** s **konstantní** rychlostí. N bitový převodník je tvořen **$2^N - 1$ komparátory**, které jsou připojeny k **napěťovému žebříku** na vstupu **minus** (-), který dělí **referenční napětí**, a na vstup **plus** (+) je přivedeno **měřené napětí**. Výsledky komparátorů jsou připojeny k **prioritnímu kodéru**, na jehož výstupu je výsledek převodu. Jeho výhodou je **rychlosť** (převod probíhá v reálném čase kontinuálně pouze s nepatrnným zpožděním), nevýhodou je cena a složitá realizace (velký počet **komparátorů**, nutnost mít **velmi přesné odpory**).
 - Pokud je Vin **poloviční** jako Vref, bude v **log. 1** spodní polovina výstupů z komparátorů.
 - Pokud je Vin **nulové**, nebude v **log. 1** **žádný** komparátor.
 - Pokud je Vin **rovné nebo větší** Vref, budou v **log. 1** všechny výstupy komparátorů.



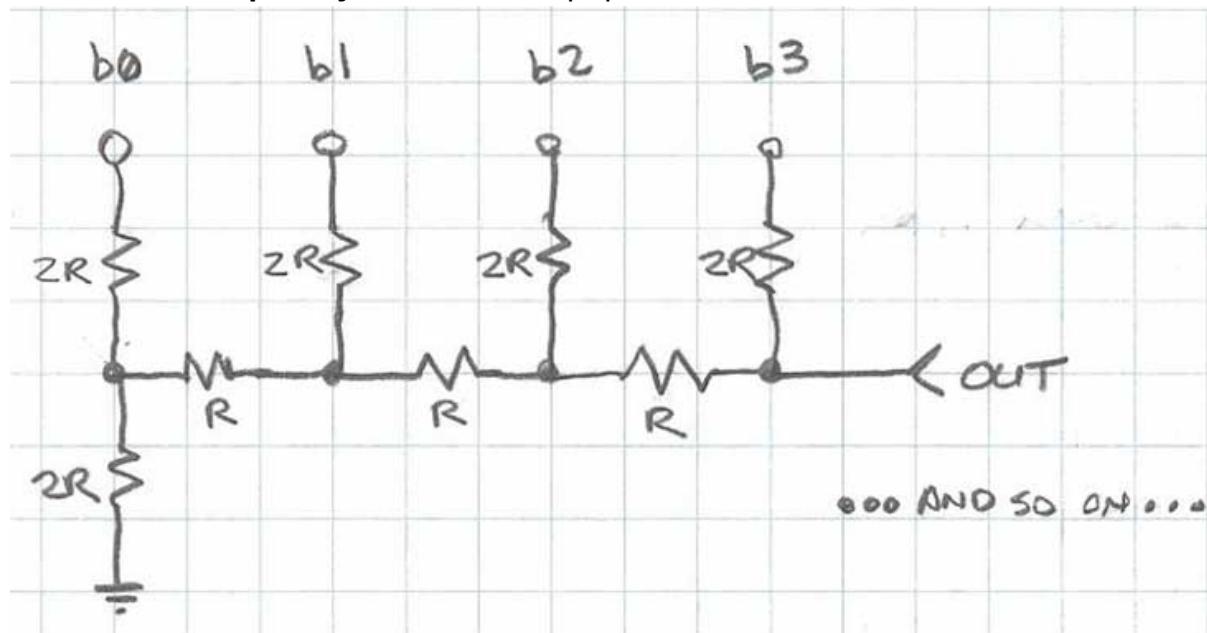
- **Aproximační ADC:** Jedná se o **sekvenční obvod**, měřené napětí musí být **navzorkováno** a poté až probíhá převod. Funguje na principu binárního vyhledávání (**půlení intervalů**) správné hodnoty napětí, pracuje ale s lineární časovou složitostí **$O(N)$** , kde **N** je počet převáděných bitů (musí se zkontořovat každý). Je tvořen:
 - **Sample and hold obvod:** navzorkuje napětí na začátku převodu a poté jej drží beze změny až do jeho konce,
 - **DA převodník:** převádí aktuální **odhad** napětí z binární hodnoty na napětí,
 - **Komparátor napětí:** porovnává **odhad** napětí naměřeného napětí s **měřeným** napětím,
 - **SAR** (successive approximation register): obsahuje aktuální odhad naměřeného napětí v binární podobě. Na začátku je tato hodnota rovna polovině rozsahu (pouze **MSB** je v **log. 1**).

Převod probíhá tak, že **MSB** v **SAR** je nastaven na **jedna** a zbylé bity jsou **vynulovány**. Tato bin hodnota je převedena na napětí pomocí **DAC** a **komparátorem porovnána** s měřeným napětím. Pokud je odhad napětí **menší** než měřené napětí, zůstane **MSB** v **log. 1**, **jinak** je změněn na **log. 0**. V obou případech je nastaven druhý nejvíce signifikantní bit na **log. 1** a proces se opakuje. Teď už se ale po porovnání případně mění tento bit a na **log. 1** se nastavuje další.



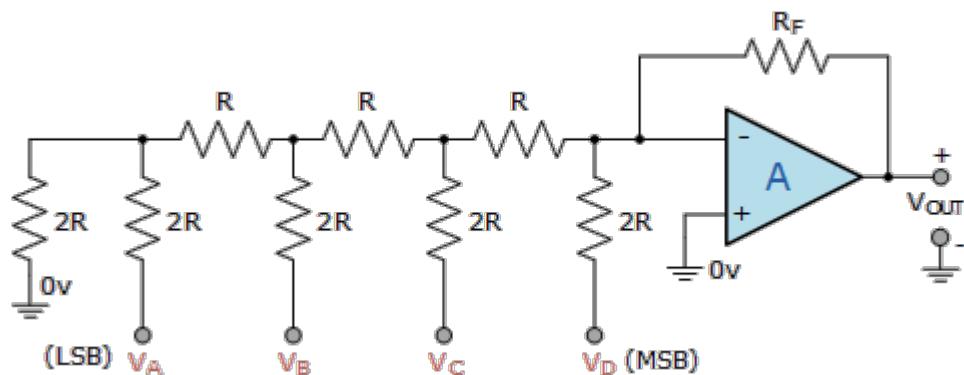
DA převodník (digital-to-analog converter)

Převádí **digitální** signál (binární číslo) na **analogový** (spojitý). Většina DAC je tvořena **R-2R odporovým žebříkem** a případně ...



b0 je **LSB**, **b3** je **MSB**. Celkový odpor obvodu v místě **out** je **R**. Paralelně zapojené dva **2R** rezistory pod **b0**, mají odpor roven součtu jejich **převrácených** hodnot, tedy **R**, což znamená, že pod **b1**, vzniknou také paralelně zapojené dva **2R** ($2R$ a $2 \times 1R$ v sérii) rezistory atd. Napětí z **b0** bude po cestě k **out** rozděleno **čtyřikrát**, tedy doputuje tam $1/2^4 = 1/16$, z **b1** bude rozděleno **třikrát** na $1/8$, z **b2** **dvakrát** na $1/4$ a z **b3** jednou na $1/2$ [DAC Methods R2R Ladder](#). Na **out** takhle může být přivedeno napětí v rozmezí **0** až **Vcc**, které je na vstupech **b0-b3**, po šestnáctinových skocích. Pokud chceme převádět na jiná napětí, musí být **out** zapojeno ještě na **operační**

zesilovač. [How OpAmps Work - The Learning Circuit](#)



Odkazy:

- Seriové rozhraní: [Sériová rozhraní u mikrokontrolerů – informatika](#)

6. Principy řízení a připojování periferních zařízení (přerušení, programová obsluha, přímý přístup do paměti, sběrnice).

Přístup k periferním zařízením řídí řadič periferních zařízení, jeho funkce jsou:

- komunikace s CPU,
- vyvolání přerušení,
- řízení a časování operací periferních zařízení,
- realizace vyrovnávací paměti,
- detekce a reportování poruch.

Řadič periferních zařízení obsahuje registry, kterými procesor se zařízením komunikuje - **datový register**, **řídící register**, **stavový registr**. Logicky lze periferní zařízení připojit dvěma způsoby:

- **mapovaný IO**: registry pro ovládání zařízení jsou **namapovány** na adresy **hlavní paměti**, periferní zařízení (registry) a paměť **sdílí stejný adresový prostor** a CPU tak může operace s periferním zařízením provádět stejně jako operace s pamětí (čtení a zápis). Řešení využívá adresový dekodér.
- **izolovaný IO**: hlavní paměť a periferní zařízení mají **různé adresové prostory**. Operace s periferními zařízeními provádí procesor pomocí speciálních instrukcí, např. IN, OUT nebo READ, WRITE.

Fyzicky lze periferních zařízení obecně připojit dvěma způsoby:

- **Point-to-point linky**: Mezi **každými dvěma** zařízeními (většinou je jeden ze dvojice **CPU**), které chtějí komunikovat je nutné vést k tomu vyhrazené vodiče.
 - **výhody**: kratší vodiče - vyšší kmitočty a **rychlosť** přenosu, komunikace 1 ku 1, současně může komunikovat **více párů** zařízeních - **velká propustnost**.
 - **nevýhody**: velký **počet** vodičů, **cena**, složitost, zařízení musí mít adekvátní počet vstupů/výstupů.
- **Sběrnice**: **velký** počet zařízení sdílí poměrně **malý** počet vodičů (připojují se na jeden centrální). Jedná se např. o Universal Serial Bus (**USB**), Peripheral Component Interconnect (**PCI**), PCI-Express, I2C
 - **výhody**: **nižší** počet vodičů, **levné**, lze dobře **škálovat**, zařízení stačí **jeden vstup/výstup** pro komunikaci s více jinými zařízeními.
 - **nevýhody**: delší vodiče - nižší kmitočty - **pomalejší** přenos, současně může komunikovat pouze **jeden páru** zařízení (respektive pouze jedno zařízení může v daný okamžik **vysílat**, přijímat mohou všechny)- **nižší propustnost**, složitější komunikace - **výběr příjemce**.

Přerušení

přerušení, respektive **obsluha přerušení**, je mechanismus, kterým lze obsloužit **asynchronně** vznikající události mimo procesor - z **periferních zařízení**, např. úder do klávesnice (přerušení mohou vznikat i **uvnitř** procesoru, např. **dělení 0**, umožňuje tak OS řešit tyto události, případně mohou být vyvolána i programově). Průběh obsluhy přerušení:

1. Periferní zařízení vyvolá přerušení (elektrický impuls) na řadič přerušení.
2. Řadič přerušení identifikuje a upozorní procesor.
3. Jakmile CPU je ve stavu, kdy může přerušení obsloužit (dokončí instrukci), požádá řadič o aktuálně **nejprioritnější nemaskované** přerušení a započne jeho obsluhu.
4. CPU uloží **stav** aktuálně běžícího procesu na **zásobník**.
5. Na základě vektoru přerušení vybere **obslužnou rutinu** (její adresu v paměti), vyhledá ji a spustí.
6. Po dokončení obslužné rutiny **obnoví stav** přerušeného programu a pokračuje v jeho vykonávání.

Priorita přerušení udává, v jakém **pořadí** budou přerušení zpracována, pokud jich současně vznikne více. Existují maskovatelná přerušení, která procesor ignoruje.

Programová obsluha (polling)

Jedná se o nenáročný a neefektivní způsob obsluhy periferních zařízení. Běžící program se **periodicky dotazuje** ve smyčce na periferní zařízení, jestli nedošlo ke změně jeho stavu (stisk klávesy). Běžící program tak zbytečně plýtvá výpočetním výkonem a elektrickou energií, protože změny stavu zařízení vznikají v poměru k testování na tyto změny málo často.

Přímý přístup do paměti (DMA - Direct Memory Access)

Koncept přerušení je nevýhodný, pokud je třeba přenášet **větší objemy dat**. Při obsluze přerušení se procesor **podílí na datových přenosech**, což ho zatěžuje. Přímý přístup do paměti umožňuje přenášet data z periferního zařízení do paměti nebo z paměti do paměti bez zatížení procesoru. Ten tak může provádět výpočet běžícího programu. Proces přímého přístupu do paměti probíhá následovně.

1. CPU předá **řadiči DMA** (specializovaný obvod, který obstarává přímý přístup do paměti) **adresu** v paměti, **adresu** periferního zařízení, **druh přenosu** (čtení/zápis) a **počet** přenášených **slov**.
2. Řadič DMA provádí **přesun** dat **bez** zásahu CPU, který může vykonávat další kód. Procesor je **omezen** na využití sběrnice.
3. Po **dokončení** přenosu dat (případně při chybě) **řadič DMA** zasílá procesoru **přerušení**, čímž ho o této skutečnosti informuje.

Rozšířením řadiče DMA je koncept IO procesoru, což je co-procesor, řídící IO periferních zařízení.

Sběrnice

Jedná se o propojovací soustava umožňující komunikaci mezi **více** než jedním **zdrojem** dat a **přijímači** dat. Zajišťuje přenos dat a řídících povelů mezi nimi a definuje:

- **komunikační rozhraní**: soustava vodičů, jejich význam a elektrické charakteristiky.
- **komunikační protokol**: přesně **určuje**, jak bude **komunikace** mezi dvěma zařízeními probíhat, např. definuje **pořadí změn hodnot signálů** (viz start a stop condition u I2C)
- **transakce** (cyklus): **posloupnost** kroků, která **realizuje** danou funkci, např přečtení dat z adresy.

Způsoby komunikace

popis viz minulá otázka.

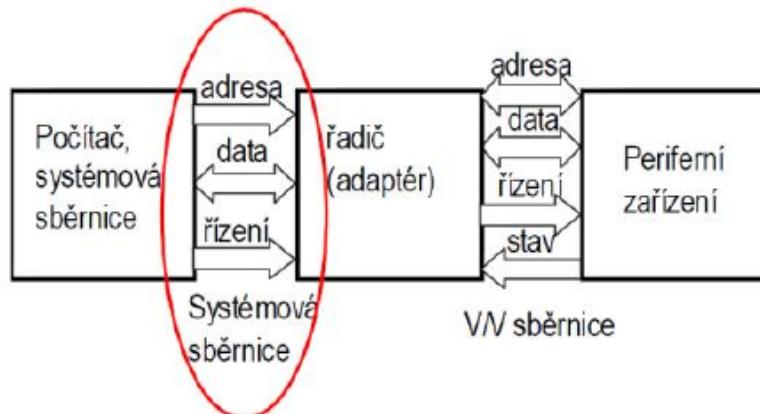
- **synchronní - asynchronní**,
- **sériová - paralelní**,
- **simplex - half duplex, full duplex**.
- **jednostranně** (bez handshake) - **oboustranně** (potvrzení přenosu - handshake)

Druhy sběrnic

- **Sériová/Paralelní sběrnice** - viz okruh 5.
- **Interní/Externí sběrnice** - Pro propojení interních nebo externích periferií.
- **Nesdílená (Dedikovaná) sběrnice** - Pro každý **signál (data, adresa, řízení)** je samostatný vodič. Připojení pouze jednoho zařízení.
- **Sdílená sběrnice** - Všechny signály se posílají po **společné** sadě vodičů. Pro rozlišení o jaký signál se jedná se používají **identifikační signály (adresy)**.

Komunikační kanály

- **Adresa**
- **Data**
- **Řízení**

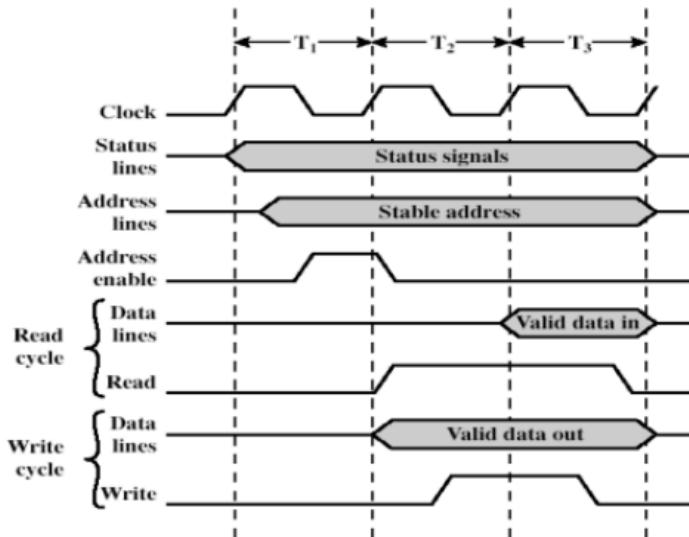


Parametry

- **Rychlosť sběrnice** - určuje ji **šířka sběrnice, technologie** (paralelní/sériová), **kmitočet synchronizačních hodin**.
- **Šířka pásma** - **šířka sběrnice * rychlosť sběrnice**

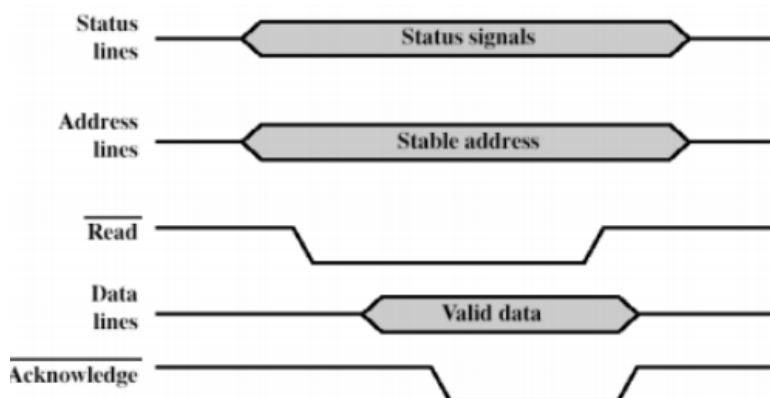
Synchronní sběrnice

Komunikace řízena hodinovým signálem, který je rozveden do všech zařízení.



Asynchronní sběrnice

Generování signálů je **vázáno** na **výskyt** událostí, obvykle zařízení mají signály definující začátek a konec čtení/zápisu (**MSYN** a **SSYN**).

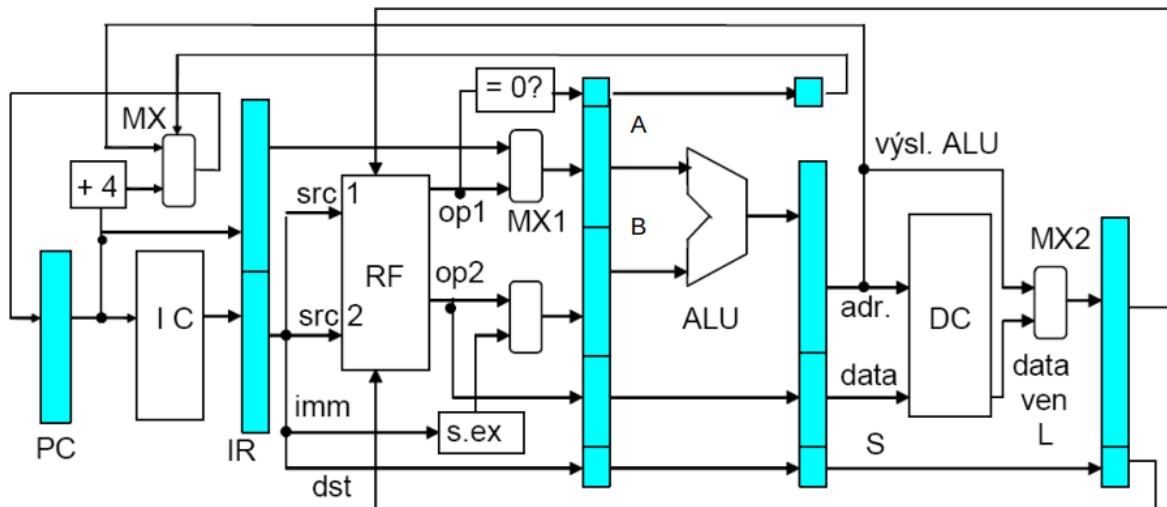


Rozhodování o přidělení sběrnice

Pokud současně žádá o přidělení sběrnice více zařízení, musí být nějak zajištěno, aby na ni vysílalo vždy pouze jedno. Pro zajištění se používají přístupy:

- **centrální řízení - decentralizované řízení,**
- **prioritní přístup - spravedlivé přidělování sběrnice - cyklické přidělování - náhodné.**

7. Princip činnosti počítače (řetězené zpracování instrukcí, RISC, CISC).



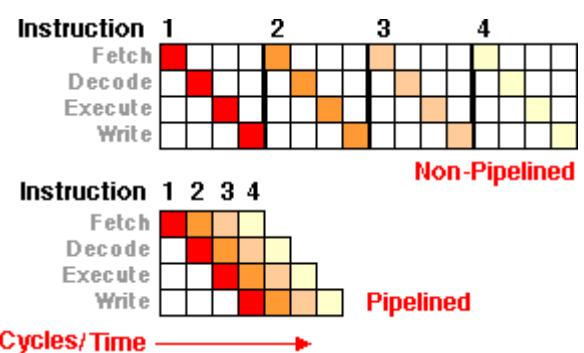
Zřetězené zpracování instrukcí (pipelining)

Procesor je velmi složitý obvod a provedení instrukce se typicky děje v **různých** jeho částech. Proto jsou CPU rozděleny na **stupně**, každý stupeň může **současně** řešit jinou část instrukce. To znamená, že lze současně provádět výpočet více instrukcí, každou v **jiném stavu dokončení**. Mezi stupni CPU přechází instrukce s **taktem** hodin, ideálně lze díky **řetězenému zpracování (pipelining)** neustále využívat **všechny** stupně CPU a po **každém taktu** dokončit jednu instrukci (**CPI = 1 cycles per instruction**) .

		Basic five-stage pipeline						
Instr. No.	Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB			
2		IF	ID	EX	MEM	WB		
3			IF	ID	EX	MEM	WB	
4				IF	ID	EX	MEM	
5					IF	ID	EX	

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.



Stavy provádění, ve kterých se instrukce může nacházet vypadají:

- **Fetch (F)** - načtení instrukce,
- **Decode (D)** - dekódování instrukce,
- **Execute (E)** - vykonání instrukce,
- **Memory Access (M)** - čtení z paměti,
- **Write Back (W)** - zápis do paměti.

Zrychlení zřetězené linky

$$\frac{N * k * t}{(k + N - 1) * (t + d)}$$

kde:

- **N** je počet instrukcí,
- **k** je počet stupňů zřetězené linky,
- **t** je zpoždění stupně,
- **d** je zpoždění registru (mezipaměti - klopný obvod)

Konflikty (hazardy)

Situace, které je nutné při řetězení instrukcí řešit, jinak by mohl být výsledek výpočtu nesprávný. Často konflikty způsobují **zpomalení** linky (**stall**) linka musí čekat na dokončení některé instrukce, někdy stačí, když překladač změní pořadí **nezávislých** instrukcí. Konflikty dělíme na:

- **Strukturální** – obvodová struktura neumožnuje současně provedení určitých akcí, např. současně čtení dvou hodnot z paměti.
- **Datové** – jsou zapotřebí data z **předcházející** instrukce, která ještě není dokončena (např. u dvou následujících instrukcí využívající stejný register). Instrukce ADD R1, R2, R3 a po ní následující SUB R4, R5, R1 pracují s registrem R1, druhá instrukce by pracovala již s **neplatnou** hodnotou, pokud by problém nebyl řešen. Hloupé řešení je pozastavit linku a počkat na dokončení první instrukce. Chytré ale dražší řešení je doplnit CPU o datové cesty, které umožní předávat mezivýsledky (**forwarding** nebo **bypassing**) mezi jednotlivými stupni CPU. Né vždy to je ale možné, např. u dvojice instrukcí LOAD R1, addr a ADD R3, R1, R2. obsah R1 je získán až ve na konci fáze Memory Access první instrukce (konec 4. taktu), instrukce ADD jej potřebuje ale už na začátku fáze Execute ve svém 3. taktu (celkově začátek 4. taktu), linka tak musí **čekat**. Datové hazardy dělíme na:
 - **RAW – Read After Write**: 2. instrukce se pokouší číst zdroj před tím, než do něj zapsala 1. instrukce.
 - **WAW – Write After Write**: 2. instrukce se pokouší zapsat operand dřív, než je proveden zápis 1. instrukcí.
 - **WAR – Write After Read**: 2. instrukce se pokouší zapsat operand dřív, než je 1. instrukcí přečten.
- **Řídící** – skoková instrukce **mění** obsah PC (program counter, ukazatel na následující instrukci), nebo jiné. U nepodmíněných skoků procesor nezná

adresu instrukce následující po skoku, nemůže tak načítat (fáze Fetch) další instrukce a linka musí být pozastavena. Pozastavení lze ale předejít v případech:

- V programu se vyskytují jiné nezávislé instrukce, kterými lze vyplnit prostor mezi **získáním** adresy a skokem, jde o **zpožděný skok**. Změnu pořadí instrukcí provádí překladač, musí zajistit, aby nedošlo ke změně výsledku.
- Zavést specializované obvody (**prediktor skoku a paměť cílové adresy skoku - BTB**), které odhadnou, zda **provést/neprovést** skok a adresu skoku. Dle odhadu se začnou načítat instrukce dle adresy z **BTB (branch target buffer** - cache, kterou CPU postupně naplňuje a aktualizuje) a linka se v případě správného odhadu **nepozastaví**. Pokud byl odhad **špatný**, je provádění těchto instrukcí předčasně ukončeno a začíná se s prováděním těch **správných**, to ale způsobí **zpomalení** linky. U moderních procesorů, které mají **delší linky** (10-20 stupňů), je velký požadavek na kvalitní prediktory. Bylo zjištěno, že se provede většinou kolem **80%** skoků. Predikce může být **statická** - určí ji překladač nebo **dynamická**, určuje ji procesor (speciální HW obvod). Nejjednodušší dynamická predikce odhaduje skok na základě toho, jak byl vykonán v předcházejícím běhu.

Ortogonalní instrukční sada

Instrukce fungují se všemi adresovacími mody (instrukce není nucená například mít jako první operand akumulátor).

RISC (Reduced Instruction Set Computer)

- **Redukovaná instrukční sada.**
- Oproti CISC má menší množství instrukcí s **jedním** adresovacím režimem - **register**, načítat data do/z registrů umožňují **pouze** instrukce LOAD a STORE.
- všechny instrukce jsou stejně dlouhé (**stejný počet bitů** v paměti).
- Na zřetězené lince lze ideálně zajistit dokončení instrukce **každý** takt **CPI = 1**. To znamená, že každá instrukce musí trvat **stejný počet taktů**.
- Využívá velké množství **registrů**.
- Na čipu **dominují registry** (paměť) oproti obvodům pro dekódování a řízení instrukcí.
- obvykle mají **menší** spotřebu energie,
- Apple Silicon.

CISC (Complex Instruction Set Computing)

- **Složitá instrukční sada** - Hlavní myšlenkou procesorů CISC je to, že k provádění operací **načítání, vyhodnocení a ukládání** lze použít jedinou instrukci (**memory-to-memory, mem-to-reg, reg-to-mem, reg-to-reg**).
- Mnoho **typů instrukcí** s mnoha variantami a mnoha **adresovacími režimy**.

- Instrukce nemusí mít stejnou délku v bitech - náročnější na dekódování.
- Dle složitosti instrukce navíc mohou trvat různou dobu (počet taktů), hůře se řeší zřetězené zpracování a **CPI > 1**.
- Komplexní instrukce mohou být na rozdíl od RISC efektivnější, ale složitější na použití.
- Nepoužívá mnoho registrů, někdy pouze **střadač**.
- Na čipu dominuje logika pro **dekódování** a **řízení** instrukcí před registry.
- Intel x86.

8. Minimalizace logických výrazů (algebraické metody, Karnaughova mapa, Quine McCluskey)

Minimalizaci logických výrazů provádíme z důvodů:

- menší, jednodušší logický výraz,
- menší počet logických hradel,
- menší počet spojů hradel,
- rychlejší výpočet,
- levnější výroba,
- méně integrovaných obvodů,
- delší doba bez poruch,
- jednodušší servis,
- větší spolehlivost při praktickém využívání.

Metody minimalizace

- **Algebraické** - Aplikace **axiomů Booleovy algebry**. Náročné pro velké příklady.
- **Grafické** - Vennovy diagramy, **Karnaughova mapa**, jednotková krychle...
- **Algoritmické** - **Quine-McCluskey**.

Algebraická minimalizace

Algebraickou minimalizaci provádíme aplikací **axiomů Booleovy algebry**, které jsou:

- **uzavřenost**: $(a+b) \in B$, $(a \cdot b) \in B$ ($a, b \in B$)
- **identita**: $a+0=a$, $a \cdot 1=a$ - z hlediska implementace není není tuto opraci provádět,
- **komutativita**: $a+b=b+a$, $a \cdot b=b \cdot a$ - z hlediska implementace umožňuje volbu zapojení do hradel,
- **distributivita**: $a+b \cdot c = (a+b) \cdot (a+c)$, $a \cdot (b+c)=a \cdot b+a \cdot c$ - kratší výrazy jsou z hlediska implementace výhodnější,
- **komplementárnost**: $a \cdot a'=0$, $a+a'=1$ - z hlediska implementace lze předem určit výsledek
- v množině B existují alespoň dva různé prvky

Teorémy

Jsou odvozeny na základě **axiomů Booleovy algebry** a definují další užitečné vlastnosti využívané při **minimalizaci** logických výrazů, jsou to:

- **jedinečnost 0 a 1**,
- **idempotence**: $a+a=a$, $a \cdot a=a$,
- **agresivita 1 a 0**: $a+1=1$, $a \cdot 1=1$,

- absorpce: $a+a \cdot b=a$, $a \cdot (a+b)=a$,
- De Morganovy zákony: $(a+b)'=a' \cdot b'$, $(a \cdot b)'=a'+b'$,
- asociativita: $(a+b)+c=a+(b+c)$, $(a \cdot b) \cdot c=a \cdot (b \cdot c)$,
- dvojitá negace: $a''=a$,
- absorpce negace: $a+a' \cdot b=a+b$, $a \cdot (a'+b)=a \cdot b$.

Normální formy

- **úplná normální disjunktní forma ÚNDF** (SoP: Sum of Products): Výraz je sepsán jako **suma součinů**. Z pravdivostní tabulky ji získáme tak, že vytvoříme **součiny** (AND) **vstupních** proměnných v řádcích, kde má výstupní funkce hodnotu **log. 1** tzv. **mintermy**. Všechny tyto mintermy pak **sečteme** (OR). Každá proměnná v součinu je zapsána tak, že pokud nabývá hodnoty **log. 0**, pak ji píšeme s **negací**, pokud **log. 1**, pak píšeme **bez negace**. U zkráceného zápisu jsou uvedeny **stavové indexy**, pro které nabývá funkce hodnoty **log. 1**.

Nezkrácená ÚNDF

$$F(x, y, z) = \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot \bar{z}$$

Varianty zkráceného zápisu ÚNDF

$$F(x, y, z) = \vee(0,2,4,6) = 1(0,2,4,6) = \Sigma m(0,2,4,6)$$

- úplná normální konjunktní forma **ÚNKF** (PoS: Product of Sums): Výraz je sepsán jako **součin sum**. Z pravdivostní tabulky ji získáme tak, že vytvoříme ze **součtu** vstupních proměnných v řádcích, kde má výstupní funkce hodnotu **log. 0** tzv. **maxtermů**, a všechny tyto **maxtermy** pak **vynásobíme**. Každá proměnná v součtu je zapsána tak, že pokud nabývá hodnoty **log. 0**, pak ji píšeme **bez negace**, pokud **log. 1**, pak píšeme s **negací**. U zkráceného

Nezkrácená ÚNKF

$$F(x, y, z) = (x + y + \bar{z}) \cdot (x + \bar{y} + \bar{z}) \cdot (\bar{x} + y + \bar{z}) \cdot (\bar{x} + \bar{y} + \bar{z})$$

Varianty zkráceného zápisu ÚNKF

$$F(x, y, z) = \wedge(1,3,5,7) = \&(1,3,5,7) = \Pi M(1,3,5,7)$$

<i>s</i>	<i>c</i>	<i>b</i>	<i>a</i>	<i>f</i>	<i>minterm</i>	<i>maxterm</i>
0	0	0	0	0	-	$a + b + c$
1	0	0	1	0	-	$\bar{a} + b + c$
2	0	1	0	0	-	$a + \bar{b} + c$
3	0	1	1	1	$ab\bar{c}$	
4	1	0	0	0	-	$a + b + \bar{c}$
5	1	0	1	1	$a\bar{b}c$	
6	1	1	0	1	$\bar{a}bc$	
7	1	1	1	1	abc	

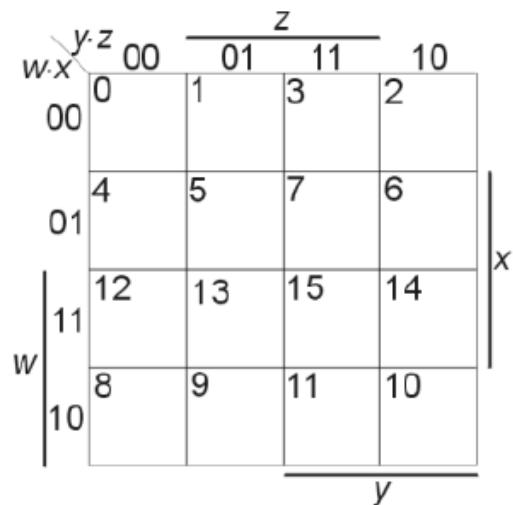
zápisu jsou uvedeny **stavové indexy**, pro které nabývá funkce hodnoty **log. 0**.

- **ÚNDF:** $a \cdot b \cdot c' + a \cdot b' \cdot c + a' \cdot b \cdot c + a \cdot b \cdot c'$
- **ÚNKF:** $(a+b+c) \cdot (a'+b+c) \cdot (a+b'+c) \cdot (a+b+c')$
- **minimální normální disjunktní forma MNDF:** Minimální možné řešení **termu** (uspořádaná skupina proměnných a operátorů) v normální disjunktní formě. Již nelze eliminovat žádnou proměnnou z termu.
- **minimální normální konjunktní forma MNKF:** Minimální možné řešení **termu** v normální konjunktní formě. Již nelze eliminovat žádnou proměnnou z termu.

Karnaughova mapy

Karnaughovy mapy vytváříme s pomocí **Grayova** (sousední hodnoty se liší o jediný bit) kódu tak, aby se každé políčko se všemi sousedními lišilo pouze o **jediný** bit

(přehození 3. sloupce a 3. řádku se 4.). Hodnota proměnné má být **1** v polích, kde je její odpovídající **bit 1**, v ostatních polích má být v **0**. V polích pod **podtržením** má proměnná **nabývat 1, v ostatních 0**.



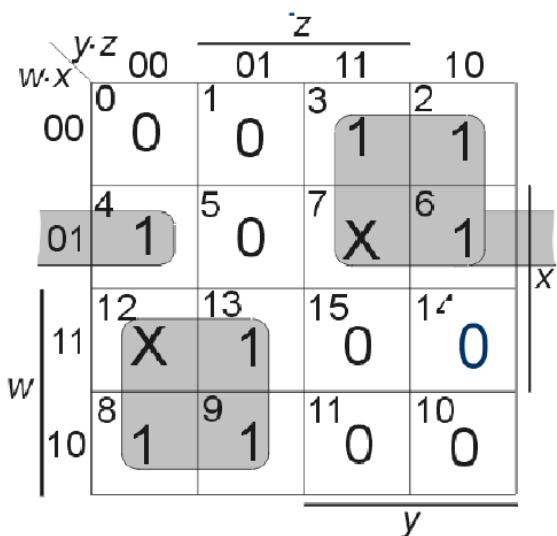
Minimalizace Karnaughovy mapy v disjunktivní formě

Jedničky (popř. x) se sdružují do **skupin**, které jsou **mocniny 2** - lze i přes okraje a rohy. Pokud daná skupina zasahuje do **bitů 0 i 1** dané proměnné, proměnná **nebude** ve výsledném termu. Pokud daná skupina zasahuje pouze do **bitů 1** dané proměnné, **bude** proměnná ve výsledném termu **přímo**. Pokud daná skupina zasahuje pouze do **bitů 0**, **bude** proměnná ve výsledném termu **negovaně**.

		z			
		00	01	11	10
		0	1	3	2
x 	0	0	1	1	0
	1	4	0	7	6
		y			

$$F(x, y, z) = \bar{x} \cdot z + x \cdot y \cdot \bar{z}$$

$$F(w, x, y, z) = w \cdot \bar{y} + \bar{w} \cdot y + \bar{w} \cdot x \cdot \bar{z}$$



$$F(w,x,y,z) = \bar{w} \cdot y + w \cdot \bar{y} + \bar{w} \cdot x \cdot \bar{z}$$

Minimalizace Karnaughovy mapy v konjunktivní formě

Nuly (popř. x) se sdružují do skupin, které jsou **mocniny 2** - lze i přes okraje a rohy.
 Pokud daná skupina zasahuje do **bitů 0 i 1** dané proměnné, proměnná **nebude** ve výsledném termu. Pokud daná skupina zasahuje pouze do **bitů 1** dané proměnné,

bude proměnná ve výsledném termu **negovaně**. Pokud daná skupina zasahuje pouze do **bitů 0**, **bude** proměnná ve výsledném termu **přímo**.

Quine McCluskey

		z				
		00	01	11	10	
		0	1	3	2	
x 0	1	0	0		1	
	1	0	0		1	
		4	1	5	6	
x 1	1	0	0		1	
	1	0	0		1	

$$F(x, y, z) = z'$$

		z				
		00	01	11	10	
		0	1	3	2	
w·x 0	1	0	0		1	
	1	0	0	X	1	
		4	5	7	6	
w 1	1	1	0	X	1	
	1	1	0	X	1	
		12	13	15	14	
w 0	X	1	1	0	X	
	1	1	1	0	0	
		8	9	11	10	
w 1	1	1	0	0	0	
	1	1	0	0	0	

$$F(w, x, y, z) = (w + x + y) \cdot (w + x' + z') \cdot (w' + y')$$

Tabulární minimalizační metoda. Vhodná pro funkce s více proměnnými. Postup pro **ÚNDF (SOP)** [Quine-McCluskey Minimization Technique \(Tabular Method\)](#):

1. Seřaď do **skupin** jednotlivé implikanty v pořadí dle **počtu jedniček** v jejich binárních reprezentaci.

$$\sum_m (0, 1, 3, 7, 8, 9, 11, 15)$$

Step: -↓

Group	Minterm	Bin. Rep.			
		A	B	C	D
0	m_0	0	0	0	0
1	m_1	0	0	0	1
	m_2	1	0	0	0
2	m_3	0	1	1	1
	m_9	1	0	0	1
3	m_7	0	1	1	1
	m_{11}	1	0	1	1
4	m_{15}	1	1	1	1

2. Eliminuj proměnné, jejichž implikanty se liší v sousedních skupinách **jediným** bitem, místo tohoto bitu piš **pomlčku**. Pokud implikant nemá sousedné termíny a nelze u něj eliminovat žádný bit, jedná se o zkrácený implikant.

Step :- 2

Group	Matched pairs	Bin. Rep.			
		A	B	C	D
0	$m_0 - m_1$	0	0	0	-
	$m_0 - m_8$	-	0	0	0
1	$m_1 - m_3$	0	0	-	1
	$m_1 - m_9$	-	0	0	1
	$m_8 - m_9$	1	0	0	-
2	$m_5 - m_7$	0	-	1	1
	$m_3 - m_n$	-	0	1	1
	$m_9 - m_{11}$	1	0	-	1
3	$m_7 - m_{15}$	-	1	1	1
	$m_{11} - m_{15}$	1	-	1	1

3. Pokud byly eliminovány některé implikanty, pokračuj bodem 1 s takto eliminovanými implikanty.

Step :- 3

Group	M = P.	B.R.			
		A	B	C	D
0	$m_0 - m_1 - m_8 - m_9$	-	0	0	-
	$m_0 - m_8 - m_1 - m_9$	-	0	0	-
1	$m_1 - m_3 - m_9 - m_{11}$	-	0	-	1
	$m_1 - m_9 - m_5 - m_{11}$	-	0	-	1
2	$m_3 - m_7 - m_n - m_{15}$	-	-	1	1
	$m_3 - m_{11} - m_7 - m_{15}$	-	-	1	1

4. Hledej minimální řešení pokrytí dané funkce pomocí mřížky implikantů.

a. zapsat zkrácené implikanty do mřížky,

P.I.	Mínimální řešení	0	1	3	7	8	9	11	15
$\bar{B}\bar{C}$	0, 1, 8, 9	X	X			X	X		
$\bar{B}D$	1, 3, 9, 11		X	X			X	X	
CD	3, 7, 11, 15			X	X			X	X

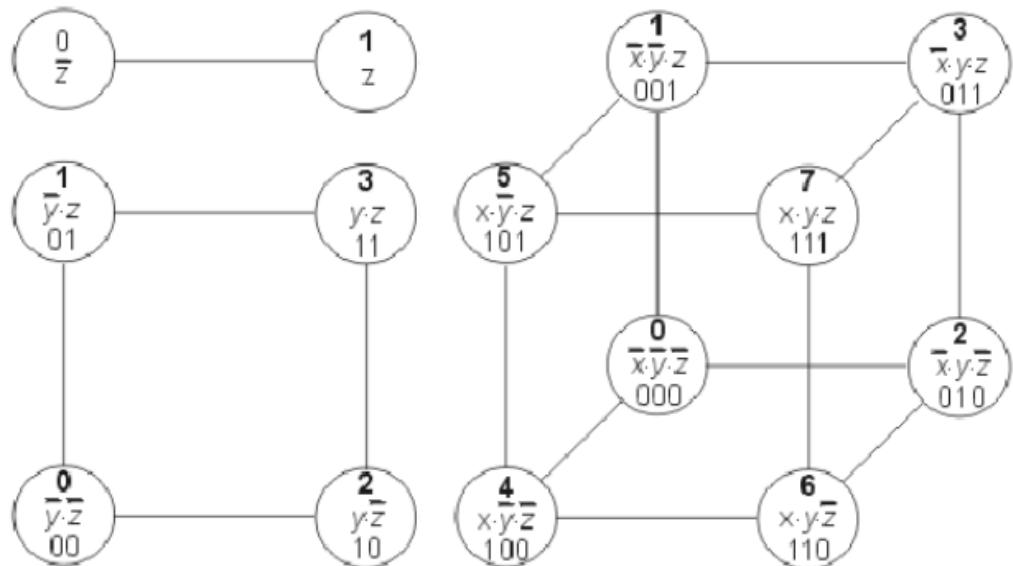
b. vybrat sloupce, kde je pouze jeden křížek,

P.I.	Mínimální řešení	0	1	3	7	8	9	11	15
$\bar{B}\bar{C}$	0, 1, 8, 9	(X)	X			(X)	X		
$\bar{B}D$	1, 3, 9, 11		X	X			X	X	
CD	3, 7, 11, 15			X	(X)			X	(X)

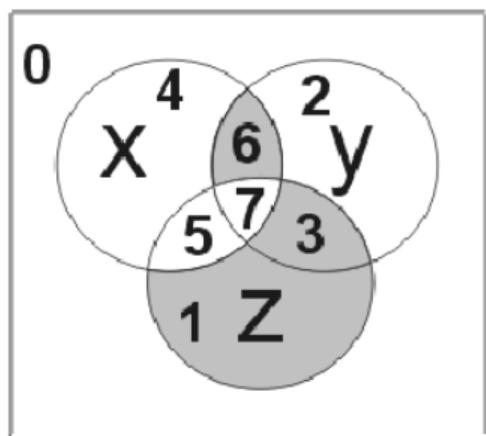
c. zapsat logický výraz na základě řádků, ve kterých se zakroužkované křížky.

$$Y = \bar{B}\bar{C} + CD$$

N-rozměrná jednotková krychle - Model pro 3-4 proměnné, velmi názorný.



Vennovy diagramy - graficky zobrazují boolovskou sousednost.



Obrázek 4 - Příklad funkce

Odkazy:

- [Disjunktivní a konjunktivní normální forma](#)

9. Reprezentace čísel a základní dvojkové aritmetické operace v počítači (doplňkové kódy, sčítání, odčítání, násobení, pevná a plovoucí řádová čárka, standard IEEE 754).

Čísla jsou v počítačích reprezentována hodnotami jednotlivých bitů - **binárně**, tedy ve **dvojkové soustavě**. Narozdíl od reálného světa je rozsah čísel omezen **počtem bitů**, které jsou pro číslo vyhrazeny. Dnes jsou obvykle používány tři číselné rozsahy, které mohou být **znaménkové i bez znaménka**, jsou to:

- **short int** - 16 bitů,
- **integer** - 32 bitů,
- **long int** - 64 bitů.

Reprezentace celých čísel se znaménkem

Reprezentace znaménka vždy sníží rozsah **absolutní** hodnoty čísla na **polovinu**, nicméně umožňuje reprezentovat **záporná čísla a celkový počet** čísel se tak **nezmění**. Reprezentace znaménka je realizována následovně:

- **Přímý kód - První bit** je rezervován pro znaménko (**1 pro záporná, 0 pro kladná**). Existují zde však **2 nuly (+0 a -0)**. Použití například v IEEE754. (10000001 = -1). Kód **komplikuje** algoritmy pro **sčítání a odčítání**, nutnost testovat na znaménko. Dalším problémem je existence **dvojí nuly**.
- **Aditivní kód (kód s posunutou nulou)** - Číslo je posunuté o nějakou danou kontantu (Např. -127 = 00000000; 128 = 11111111; 0 = 01111111; -1 = 01111110...). **Nevýhodou** je, že reprezentace kladných znaménkových čísel se **liší** od neznaménkových, a při **násobení** se musí odečítat určená konstanta. Sčítání lze provádět normálně.
- Jedničkový doplněk (inverzní kód) - Záporná hodnota se získá **znegováním** všech bitů kladného čísla (MSB musí být u **kladných** vždy **0** a u **záporných** vždu **1**). Např. 0101 = 5, 1010 = -5, 0000 = 0, 1111 = -0. Kód **komplikuje** algoritmy pro **sčítání a odčítání** (jiné než u čísel bez znaménka). Dalším problémem je existence **dvojí nuly**.
- **Dvojkový doplněk** (doplňkový kód) - Kladné číslo je reprezentováno normálně, záporné však je **inverzí** kladného čísla a **přičtením 1**. **Nejpoužívanější** formát. Pouze jedna reprezentace 0. (11111111 = -1). Sčítání a odčítání lze realizovat **stejně** jako s neznaménkovými čísly, u násobení je nutné **replikovat MSB na dvojnásobný** rozsah čísla.

Reprezentace desetinných čísel

Umožňují reprezentovat reálná čísla s omezenou přesností.

Pevná řádová čárka

Pro zobrazení kladných reálných čísel. **Přesný počet** bitů je rezervováno pro **celou** část a **zbytek** pro **desetinnou** část čísla. Dnes se skoro nevyužívá. Např. 5 číslic pro celou část a 3 číslice pro desetinnou, **00101001 = 5.125**.

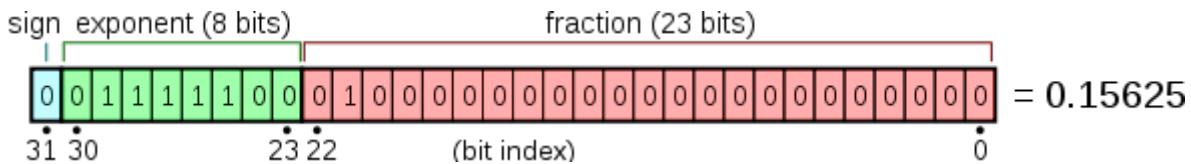
Plovoucí řádová čárka (Floating point) - IEEE 754

Číslo je reprezentováno:

- **znaménkovým bitem (S)** - MSB - 1 znamená záporný, 0 kladný,
- **mantisou (M)** - v přímém kódu, obsahuje **implicitní 1**,
- základem (*B*) - je implicitní, rovna číslu **2**,
- **exponentem (E)** - v kódu posunuté nuly, což umožňuje vyjádřit záporný exponent.

IEEE 754 definuje uložení čísla v plovoucí řadové čárce jako MSB je znaménkový bit, následují bity exponentu a poté až bity mantisy. Definovány jsou dvě reprezentace lišící se přesností a rozsahem čísla.

- **Single precision** (float - 32 bitů): Exponent má rozsah **8** bitů a je posunut o **-127** (polovinu rozsahu), mantisa je vyjádřena na **23** bitech, MSB je znaménkový. **Přesnost** je **7** dekadických čísel, rozsah zobrazení je $(-2^{127}, 2^{127})$, rozlišitelnost **normalizovaných** čísel je 2^{-127} a **nenormalizovaných** 2^{-150} .



podmínka	hodnota	poznámka
$E = 1$ až 254	$X = (-1)^s \times 2^{E-127} \times (1 + Q)$	základní formát
$E = 0, Q \neq 0$	$X = (-1)^s \times 2^{-126} \times Q$	denormalizovaná čísla
$E = 0, Q = 0, s = 0$	$X = 0$	kladná nula
$E = 0, Q = 0, s = 1$	$X = 0$	záporná nula
$E = 255, Q = 0, s = 0$	$X = +\infty$	kladné nekonečno (výsledek byl příliš vysoký)
$E = 255, Q = 0, s = 1$	$X = -\infty$	záporné nekonečno (výsledek byl příliš nízký)
$E = 255, Q > 0$	$X = \text{NaN}$	není číslo

- **Double precision** (double - 64 bitů): Exponent má rozsah **11** bitů a je posunut o **-1023** (polovinu rozsahu), mantisa je vyjádřena na **52** bitech, MSB je znaménkový. **Přesnost** je **16** dekadických čísel, rozsah zobrazení je $(-2^{1023}, 2^{1023})$, rozlišitelnost **normalizovaných** čísel je 2^{-1023} a **nenormalizovaných** 2^{-1075} .



Explicitní jednička - V mantise čísla je nutné explicitně mít jedničku.

Implicitní jednička - Umožňuje zvýšit rozsah reprezentovaného čísla o jeden bit, v mantise se jednička neuvádí. Až na **denormalizované** čísla se s ní pracuje implicitně - **1.011 -> 011**.

Chyba zobrazení

Jelikož se číslo skládá ze **součtu zlomků** mocnin 2 (např. $2^{-2} = \frac{1}{4}$) **není** možné na konečném počtu bitů (registru) **reprezentovat** všechna čísla a tak dochází k **nepřesnostem**. Např. číslo **0.1** nelze přesně vyjádřit, což je obvyklá hodnota ve finančnictví a je nutné pro tyto aplikace používat jiné kódování čísel. Používá se **BCD (Binary Coded Decimal) kód**

- Každá dekadická číslice je zobrazena v jednom niblu (4 bitech bytu).

Znaménko může být v prvním bytu. $01000010 = 42$.

Další nepřesnosti vznikají při sčítání/odčítání čísel, která jsou od sebe **řádově** vzdálená, mají **výrazně** rozdílný exponent (menší číslo může být úplně ignorováno).

Matematické operace

Sčítání

- **Celočíselné** sčítání se provádí stejně jako v 10 soustavě. V případě sčítání v procesoru může dojít k **přetečení** (součet se nevleze do podporovaného rozsahu).

Zobrazitelný součet:

$$\begin{array}{r} 00101100 \\ + 01011010 \\ \hline 10000110 \end{array} \quad \begin{array}{l} \text{desítkově: } 44 \\ + 90 \\ \hline 134 \end{array}$$

Nezobrazitelný součet:

$$\begin{array}{r} 10110100 \\ + 10110100 \\ \hline 1|01101000 = 104_{10} \# \end{array} \quad \begin{array}{l} \text{desítkově: } 180 \\ + 180 \\ \hline 360 \end{array}$$

- **Floating point** sčítání není **asociativní** vlivem **zaokrouhlovacích chyb**. V **IEEE 754** se u sčítání musí nejdříve převést obě čísla na **stejný exponent**. Vždy se převádí číslo **s menším** exponentem **na větší**. (Nezapomenout na implicitní 1)[HOW TO: Adding IEEE-754 Floating Point Numbers](#)

Diagram illustrating floating-point addition:

$X = 1.000111 \times 2^5$ $Y = 1.01001 \times 2^{-1}$

$$\begin{array}{r} 1.000111 \times 2^5 \\ + 0.101001 \times 2^{-1} \\ \hline 1.110001 \times 2^5 \end{array}$$

The diagram shows the addition of two floating-point numbers. Number X has a binary exponent of 5 and a significand of 1.000111. Number Y has a binary exponent of -1 and a significand of 1.01001. The result is 1.110001 times 2^5.

Odečítání

V CPU lze odčítání realizovat **bitovou negací** jednoho ze sčítanců a použít sčítání s nastavením **C0** na **1**. Stejně tak je výhodné provádět ručně, pro člověka je to přirozenější.

Zobrazitelný rozdíl:

$$\begin{array}{r} 01011010 \\ - 00101100 \\ \hline 00101110 \end{array} \quad \text{desítkově: } 90$$

$$- 44$$

$$46$$

Nezobrazitelný rozdíl:

$$\begin{array}{r} 1 | 00101100 \\ - 01011010 \\ \hline 11010010 = 210_{10} \end{array} \quad \text{desítkově: } 44$$

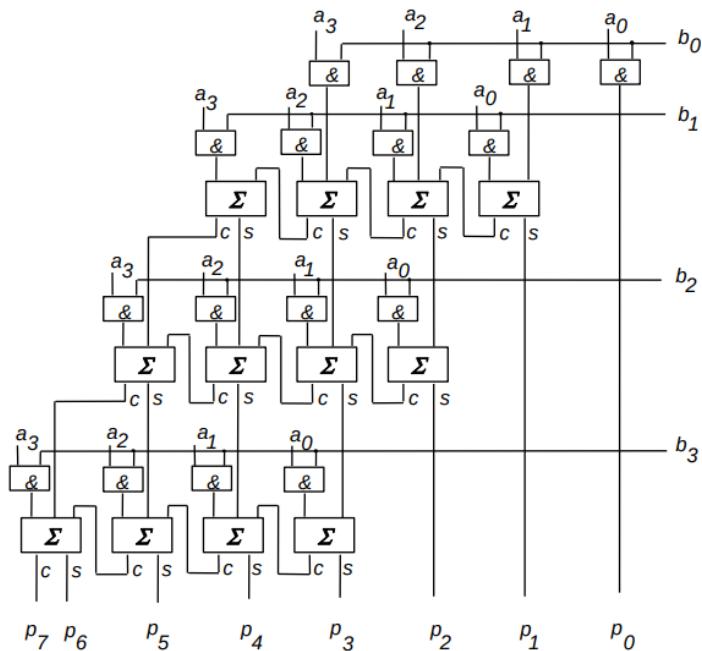
$$- 90$$

$$\# \quad - 46$$

Násobení

- **Celočíselné:** Výsledek může zabírat až **dvojnásobek** bitů. Při násobení čísel v doplňkovém kódru musí být nejdřív u obou čísel **rozšířen MSB** značící

$$\begin{array}{r} 0100 \\ * 0111 \\ \hline 0100 \\ 0100 \\ 0100 \\ \hline 11100 \end{array}$$



znaménko na **dvojnásobnou** přesnost původního čísla ve dvojkovém doplňkovém kódru.

- **IEEE 754:**

- **znaménko** výsledku se určí jako **XOR** znamének **činitelů**,
- **exponent** se určí součtem exponentů činitelů,
- **mantisa** se určí celočíselným násobením činitelů doplněných o **implicitní 1**.

$$X = -0.3_{10} = 0.01\overline{0011}_2 = 1.\overline{0011}_2 \times 2^{-2}_{10}$$

$\boxed{1|0111101|00110011001100110011010}$

$$Y = 500.25_{10} = 11110100.01_2 = 1.1111010001_2 \times 2^8_{10}$$

$\boxed{0|10000111|1111010001000000000000}$

$$X \cdot Y = (X_s \cdot Y_s) \cdot 2^{X_E + Y_E} = (X_s \cdot Y_s) \cdot 2^6$$

$$X_E = -2 \quad X_s = 1.00110011001100110011010$$

$$Y_E = 8 \quad Y_s = 1.111010001$$

$= 1101$

Celočíselné dělení

V počítači velmi náročná operace prováděná mnoha různými metodami. Znaménkové dělení se může provádět s absolutní hodnotou a až poté dojde k doplnění znaménka. Postupy dělení:

- triviálně na principu **dělení pod sebou**,

Dekadicky	$55 : 5 = 11$	
Binárně	$1\textcolor{red}{1}0\textcolor{red}{1}1\textcolor{red}{1}:101=1011$	
	$1\textcolor{green}{1}0$ ↑	Za základ vezmeme 110, 110 > 101, výsledek = 1
	$1\textcolor{brown}{1}$ ↑	$110 - 101 = 1$, přidáme 1, $11 < 101$, výsledek = 0
	$1\textcolor{brown}{1}\textcolor{green}{1}$ ↑	Opišeme 11 a přidáme 1, $111 > 101$, výsledek = 1
	$1\textcolor{brown}{0}1$ ↑	$111 - 101 = 10$, přidáme 1, $101 = 101$, výsledek = 1
	0	$101 - 101 = 0$, zbytek = 0

- **bez restaurace nezáporného zbytku**

- začne se odečtením dělítel od dělence,
- na základě MSB se určí výsledek **i-tého** bitu: pro **MSB=1** je výsledek **0** - záporné číslo, pro **MSB=0** je výsledek **1** - kladné číslo),
- dělitel se posune o 1 bit doleva (násobení 2) a **MSB** se **zahodí**,
- pokud je výsledek i-tého bitu **0**, **přičte** se dělitel, v **opačném** případě se dělitel **odečte**.
- pokud nebylo dosaženo maximálního posuvu, pokračuje se bodem **b**),
- na konci může být nutné provést korekci zbytku

30=00011110, 7=0111, -7=1001
 00011110

$$\begin{array}{r} +1001 \\ \hline 10101110 \end{array}$$
 <0 => c4 = 0
 0101110x posuv <-

$$\begin{array}{r} +0111 \\ \hline 1100110x \end{array}$$
 <0 => c3 = 0
 100110xx posuv

$$\begin{array}{r} +0111 \\ \hline 000010xx \end{array}$$
 >0 => c2 = 1
 00010xxx posuv

$$\begin{array}{r} +1001 \\ \hline 10100xxx \end{array}$$
 <0 => c1 = 0
 0100xxxx posuv

$$\begin{array}{r} +0111 \\ \hline 1011xxxx \end{array}$$
 <0 => c0 = 0

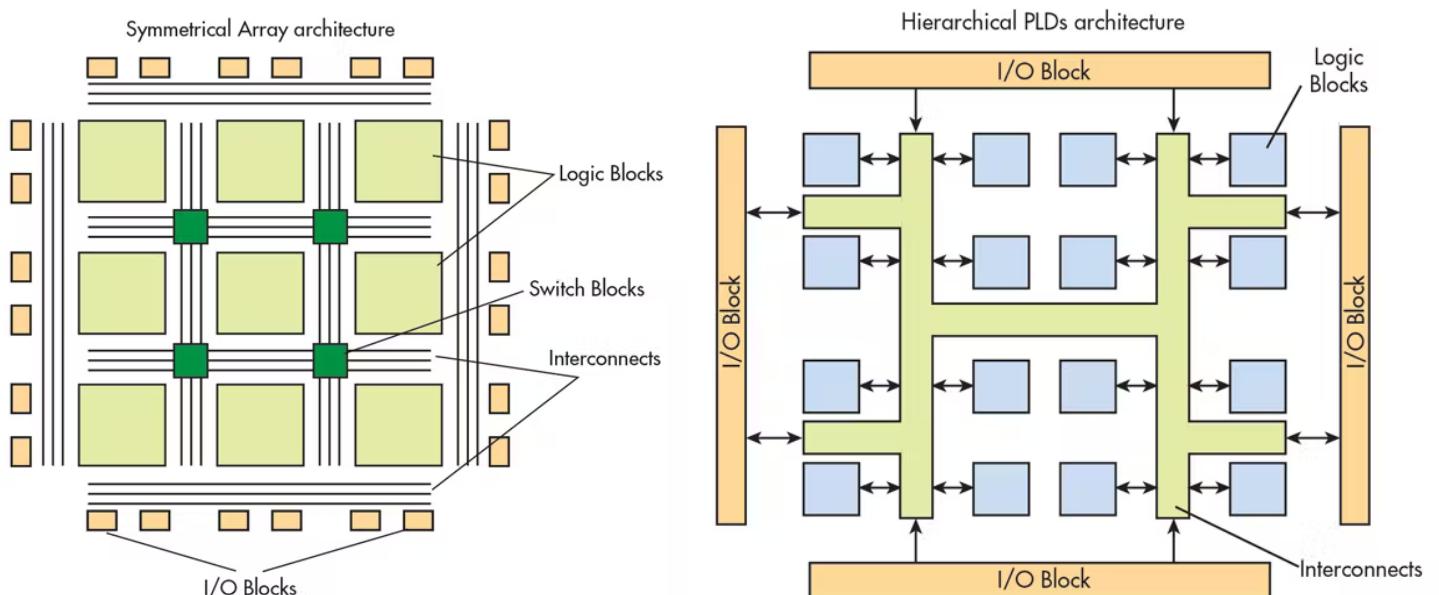
$$\begin{array}{r} +0111 \\ \hline 0010xxxx \end{array}$$
 +d (korekce na zbytek 2)

- algoritmem SRT

Odkazy:

- [How to Convert a Number from Decimal to IEEE 754 Floating Point Representation](#)
 - [Video](#) - HOW TO: Convert Decimal to IEEE-754 Single-Precision Binary
- [HOW TO: Adding IEEE-754 Floating Point Numbers](#)

10. Technologie FPGA (vnitřní struktura, LUT), kroky návrhu aplikací využívajících FPGA a základy syntetizovatelného popisu hardware (strukturní a behaviorální popis obvodů).

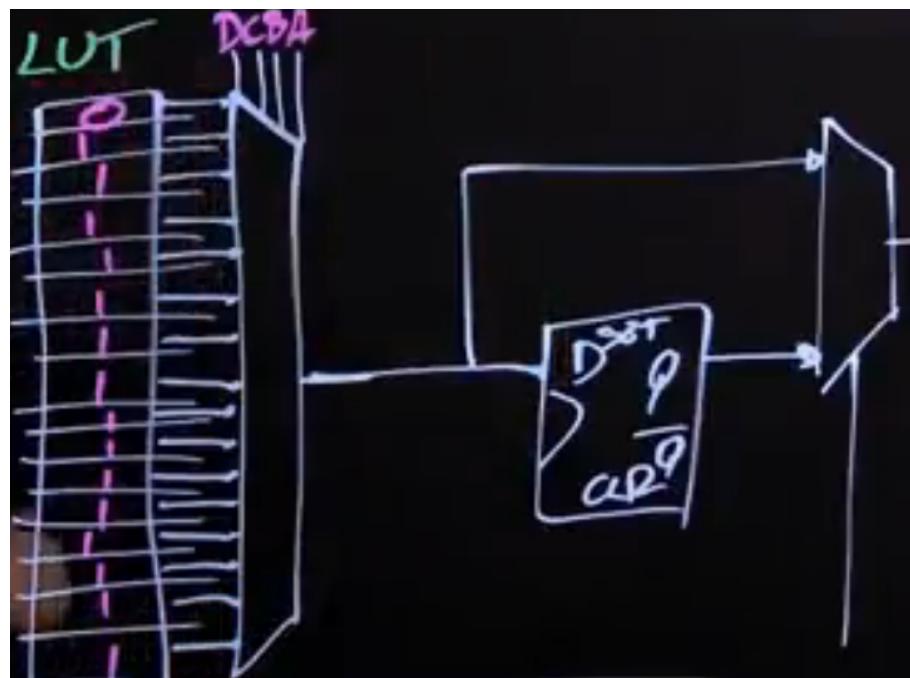


FPGA (Field Programmable Gate Array)

FPGA je programovatelné hradlové pole, které je kompromisem mezi pouze HW řešením - ASIC (Application specific integrated circuit - nejrychlejší, nejúspornější) a

pouze SW řešení (největší flexibilita). Je tvořeno z:

- **Configurable Logic Blocks (CLBs)** — umožňují naprogramovat logické funkce realizované hardwarově. Jsou tvořeny součástkami, které dohromady v nejjednodušším případě mohou tvořit jeden **CLB** nebo **slice**. **CLB** může být tvořeno více **slice**:
 - Look Up Table (**LUT**): 2^N bitový register obsahující výsledky kombinační logiky **N** proměnných. Jedná se volatilní paměť, tzn. konfigurace jednotlivých LUT musí být uložena v nevolatilní paměti (FLASH), ze které jsou LUT při spuštění nakonfigurovány.
 - **2^N-1 multiplexor**: **N** určuje počet vstupních proměnných. Pomocí kterého se **vybírá** hodnota z **LUT** na výstup na základě hodnot **vstupních proměnných**.
 - klopný obvod (typ **D**): vytváří paměť a umožňuje tvorbu sekvenčních obvodů.
 - **2-1 multiplexor**: vybírá mezi výstupem z LUT (získaného pomocí **2^N-1 multiplexoru**) nebo výstupem z klopného obvodu. Tento multiplexor tak určuje, jestli půjde o sekvenční, nebo kombinační logiku.

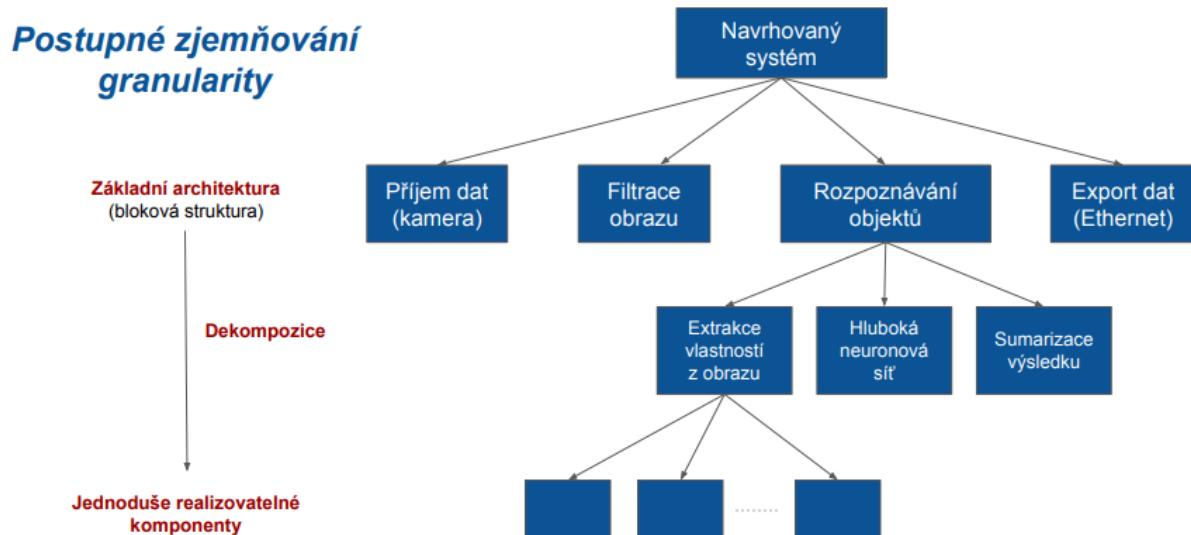


- případně další komponenty...

- **Programmable Interconnects** — programovatelné **propojení** mezi **CLBs**, jedná se o spoje vedené horizontálně a vertikálně. V místech křížení lze naprogramovat logiku propojování (**switch**).
- **Programmable I/O Blocks (IOBs)** — slouží k propojování **konfigurovatelných logických bloků** s externími součástkami pomocí propojovacích **pinů** FPGA čipu.
- **Vestavěné bloky** — paměti, sčítáčky, násobičky, ethernet, ...

Kroky návrhu aplikací využívajících FPGA

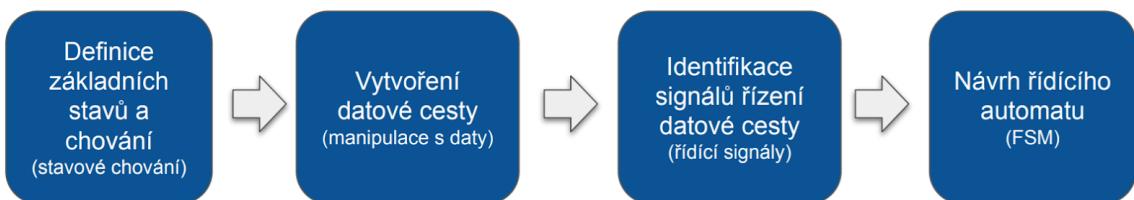
Pro návrh aplikací využívající FPGA se používá systém návrhu **shora dolů (Top-Down)**. Začíná z celkového popisu problému a končí návrhem s úplnou mírou detailů jeho částí - **dekompozice** a začlenění do celku. Na každé úrovni abstrakce je potřeba **verifikovat správnou** funkci navrhovaného systému. Dekompozicí vzniknou **jednoduše realizovatelné komponenty** (sčítáčka, posuvný registr, násobička, čítač, ...), které lze popsát ve VHDL nebo Verilog. Např. zpracování obrazu:



Metodologie návrhu na úrovni meziregistrových přesunů (RT)

Návrh je vhodné rozdělit do několika kroků:

- **Popis obvodu na úrovni základního stavového automatu** - automat popisuje základní stavy obvodu, ale neřeší konkrétní nastavení řídících, vstupních nebo výstupních signálů.
- **Vytvoření datové cesty (datapath)** - datová cesta provádí manipulaci s daty, a to na základě řízení z obecného automatu.
- **Identifikace signálů pro řízení datové cesty** - definice signálů, které jsou potřeba pro napojení datové cesty na řídící blok.
- **Návrh řídícího automatu** - převedení obecného automatu na Mealyho nebo Moorův automat, který řídí datovou cestu



Strukturální, behaviorální a dataflow popis ve VHDL

V praxi se typy popisů často kombinují, jedna logika lze současně popsat strukturálně, behaviorálně i pomocí dataflow. Pomocí syntézy (něco jako komplikace) jsou tyto popisy převáděny na konkrétní obvodová řešení pro danou technologii (FPGA, ASIC). VHDL umožňuje také pouze simulovat dané obvody.

Strukturální popis

Při strukturálním popisu **definujeme entitu** (např. poloviční sčítáčku), **propojením** vstupů a výstupů a **komponent** (AND a XOR hradla), které realizují logiku komponenty. Samotné komponenty mohou být předtím popsány jako entity na nižší úrovni abstrakce. Strukturální popis **umožňuje rozdělit komplexní** systém **na méně komplexní** části, ty mohou být vyvíjeny a ověřovány samotně. Návrhář má u tohoto typu popisu kontrolu (pokud komponenty postupně popisuje od úrovně hradel nebo i níže) nad tím, jak bude výsledný obvod vypadat (z jakých bude prvků - hradel).

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3.
4. entity half_adder is
5.     port (a, b: in std_logic;
6.            sum, carry_out: out std_logic);
7. end half_adder;
8.
9. architecture structure of half_adder is      -- Architecture body for half adder
10.
11.    component xor_gate
12.        port (i1, i2: in std_logic;
13.                o1: out std_logic);
14.    end component;
15.
16.    component and_gate
17.        port (i1, i2: in std_logic;
18.                o1: out std_logic);
19.    end component;
20.
21. begin
22.     u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
23.     u2: and_gate port map (i1 => a, i2 => b, o1 => carry_out);
24.     -- We can also use Positional Association
25.     --     => u1: xor_gate port map (a, b, sum);
26.     --     => u2: and_gate port map (a, b, carry_out);
27. end structure;
```

Behaviorální popis

Popisuje chování systému/funkce/bloku algoritmicky. Jedná se o nejvíce abstraktní popis. Z behaviorálního popisu není přímo jasné, jaká bude implementace na úrovni hradel (HW realizace). Behaviorální popis prvku obsahuje jeden nebo více procesů, které se dějí paralelně. Popis uvnitř procesu se odehrává sekvenčně. V procesu se nastavují výstupy na základě hodnot vstupů.

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3.
4. entity half_adder is
5.   port (a, b: in std_logic;
6.         sum, carry_out: out std_logic);
7. end half_adder;
8.
9. architecture behavior of half_adder is
10. begin
11.   ha: process (a, b)
12.   begin
13.     if a = '1' then
14.       sum <= not b;
15.       carry_out <= b;
16.     else
17.       sum <= b;
18.       carry_out <= '0';
19.     end if;
20.   end process ha;
21.
22. end behavior;
```

Dataflow popis

Pomocí DataFlow popisu se modelují systémy na základě toho, jak jimi putují data. Tento způsob popisu využívá odpovídající implementace logických bran. DataFlow popis popisuje systém/prvek jedním nebo více přiřazeními paralelních signálů.

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3.
4. entity half_adder is
5.   port (a, b: in std_logic;
6.         sum, carry_out: out std_logic);
7. end half_adder;
8.
9. architecture dataflow of half_adder is
10. begin
11.   sum <= a xor b;
12.   carry_out <= a and b;
13. end dataflow;
```

Logická syntéza

Automatická transformace mezi různými úrovněmi popisu. Transformace na jemnější popis s cílem vylepšit parametry zadané uživatelem: rychlosť, spotřeba, rozměry.

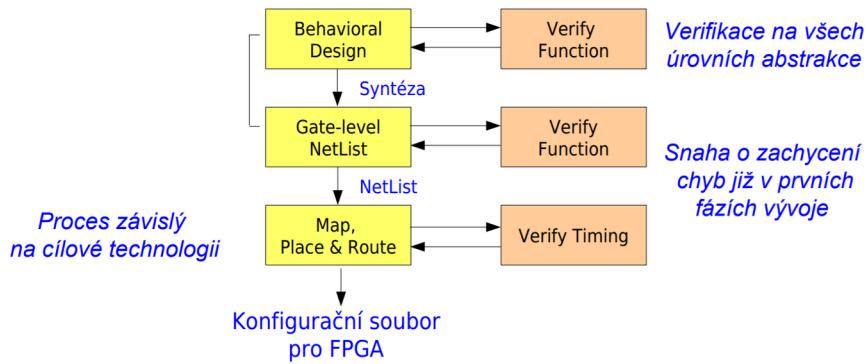
Behaviorální syntéza

Z behaviorálního popisu algoritmu je vytvořena reprezentace na úrovni struktury (sčítáčka, posuvný registr, paměť, řídicí logika)

Logická syntéza

- Z HDL popisu na úrovni RT (meziregistrových přenosů) je vytvořen NetList prvků cílové technologie

| Design Flow pro technologii FPGA



Rozpoznání prvků cílové technologie a jejich mapování do FPGA. Výsledkem procesu je konfigurační soubor pro FPGA.

Automatická syntéza:

- vstup: popis obvodu v HDL, knihovna prvků cílové technologie, constraints (např. spotřeba, velikost na čipu, čas...) výstup: optimalizovaný NetList na úrovni prvků cílové technologie

11. 2D vektorová grafika: metody rasterizace úseček a polygonů, reprezentace objektů pomocí Bézierovy křivky

Vektorová grafika

Zpracovávané a zobrazované informace popisujeme a **ukladáme analyticky** (spojitě) ve formě skupiny **vektorových entit** (**úsečky, kružnice, křivky, polygony, atd.**)

- Přesnost popisu je teoreticky neomezená.
- Vlastnosti uložených objektů (obrázků) lze kdykoliv jednoduše měnit.
- SVG

Rastrová grafika

Zpracovávané a zobrazované informace popisujeme a **ukladáme diskrétně** ve formě **rastrové matice** (2D/3D) po **pixelech**.

- Nelze jednoduše měnit vlastnosti uložených objektů.
- Daná neměnná přesnost (rozlišení), nelze jednoduše měnit.
- PNG, JPG

Rasterizace

Proces převodu vektorové reprezentace dat na rastrovou formu.

Úsečka

Základní vektorová entita definovaná 2 body (konce). Všechny zmíněné algoritmy pracují **správně** pouze v **1. kvadrantu** a jen pokud úsečka **roste** rychleji ve směru osy **X** (má max sklon **45 stupňů**), ve směru **osy Y** také musí **růst**.

- **Obecný tvar** - $Ax + By + C = 0$, kde $A = (y_2 - y_1)$, $B = (x_2 - x_1)$, **POZOR** vektor $[A, B]$ je **normálovým vektorem** přímky.
- **Parametrický tvar** - $x = x_1 + tA$, $y = y_1 + tB$, kde $t \in [0, 1]$ a $A = (x_2 - x_1)$, $B = (y_2 - y_1)$, vektor $[A, B]$ je **směrovým vektorem** přímky.
- **Směrnicový tvar** - $y = k*x + q$, kde $k = (y_2 - y_1) / (x_2 - x_1)$

Algoritmus DDA (Digital differential analyzer)

Výpočet je ve **floating** point aritmetice. Postup je následující:

- Vykresluje úsečku po pixelech od bodu **P1** k bodu **P2**.
- V **ose X** postupujeme s přírůstkem **dx = 1**.
- V **ose Y** je přírůstek dán **velikostí směrnice** ($k = (y_2 - y_1) / (x_2 - x_1)$) úsečky.

- Souřadnice **Y** se **zaokrouhuje** na **nejbližší celé** číslo - (přičítst **0.5** před konverzí na **int**).

```
LineDDA(int x1, int y1, int x2, int y2)
{
    double k = (y2-y1) / (x2-x1);
    double y = y1;

    for (int x = x1; x <= x2; x++)
    {
        draw_pixel( x, round(y));
        y += k;
    }
}
```

DDA s fixed-point aritmetikou

Používá bitový posun a eliminuje tak nutnost používat floating point, jinak pracuje na stejném principu jako floating point DDA.

```
#define FRAC_BITS 8

LineDDAFixed(int x1, int y1, int x2, int y2)
{
    int y = y1 << FRAC_BITS;
    int k = ((y2-y1) << FRAC_BITS) / (x2-x1);

    for (int x = x1; x <= x2; x++)
    {
        draw_pixel( x, y >> FRAC_BITS);
        y += k;
    }
}
```

Bresenhamův algoritmus (midpoint algoritmus)

Algoritmus používá celočíselnou aritmetiku. Je jednodušší pro HW implementaci.

Dnes je tudíž **nejpoužívanější**. Postup:

- Vykresluje úsečku po pixelech od bodu **P1** k bodu **P2**.
- V ose **X** postupujeme s přírůstkem **dx = 1**.
- O posunu v ose **Y** rozhodujeme podle znamenka tzv. **prediktoru**.

Rozhodování a výpočet chyby vykreslování E

$$E_i + \frac{\Delta y}{\Delta x} \begin{cases} < 0.5 & (x_i + 1, y_i) \\ \geq 0.5 & (x_i + 1, y_i + 1) \end{cases} \quad E_{i+1} = E_i + \frac{\Delta y}{\Delta x}$$
$$E_{i+1} = E_i + \frac{\Delta y}{\Delta x} - 1$$

Převod porovnání s 0.5 na test znaménka

- Nerovnice násobíme $2\Delta x$

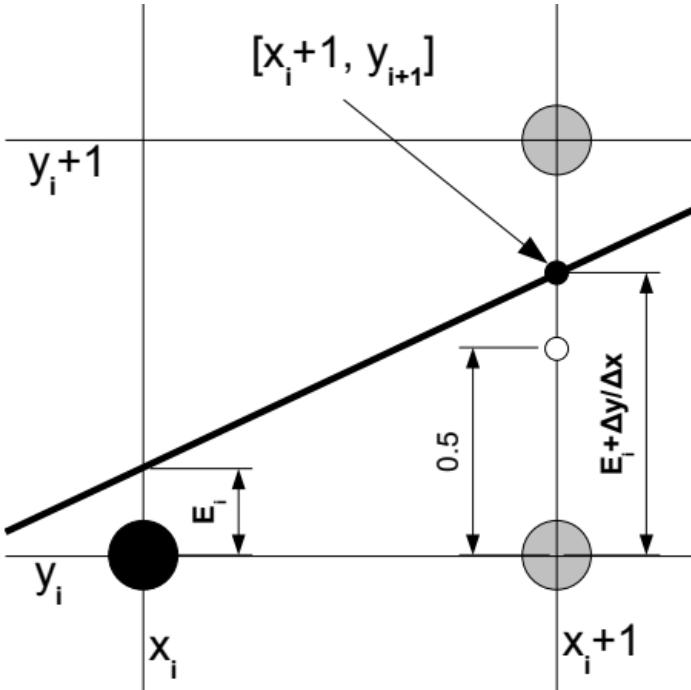
$$2\Delta x E_i + 2\Delta y - \Delta x \begin{cases} < 0 & E_{i+1} = E_i + 2\Delta y \\ \geq 0 & E_{i+1} = E_i + 2\Delta y - 2\Delta x \end{cases}$$

- Rozhodovací člen nazveme **prediktorem** P_i

$$P_i = 2\Delta x E_i + 2\Delta y - \Delta x \begin{cases} < 0 & P_{i+1} = P_i + 2\Delta y \\ \geq 0 & P_{i+1} = P_i + 2\Delta y - 2\Delta x \end{cases}$$

Počáteční hodnota predikce ($E_0 = 0$)

$$P_0 = 2\Delta y - \Delta x$$



LineBres(int x1, int y1, int x2, int y2)

{

```

int dx = x2-x1, dy = y2-y1;
int P = 2*dy - dx;
int P1 = 2*dy, P2 = P1 - 2*dx;
int y = y1;

for (int x = x1; x <= x2; x++)
{
    draw_pixel( x, y);
    if (P >= 0)
        { P += P2; y++; }
    else
        P += P1;
}
}
```

Kružnice

Definována souřadnicí **středu** a **poloměrem**:

$$(x - s_1)^2 + (y - s_2)^2 = r^2$$

Výpočty se provádí pro **% kružnice**, zbylá část se dopočítá díky **symetrii**. Algoritmy jsou odvozeny pro kružnici se **středem v počátku** [0, 0].

Vykreslení po bodech (plyne z rovnice kružnice)

Nejjednodušší pro pochopení a implementaci, ale náročné pro HW zpracování. Pracuje se s **desetinnými** čísly (floating point). Vykresluje se oktant [x,y] na obrázky. **cx** je x-ová souřadnice **středu** a **cy** je y-ová souřadnice **středu**. Algoritmus vykresluje ve směru hodinových ručiček následovně:

- Jdeme po pixelu od bodu [0, R], dokud není $x = y$.
- V **ose X** postupujeme s přírůstkem **dx = 1**.
- Pozici v **ose Y** vypočteme po každé změně X podle vztahu $y = \sqrt{R^2 - x^2}$

- Souřadnice Y se zaokrouhuje na nejbližší celé číslo - (přičíst 0.5 před konverzí na int).

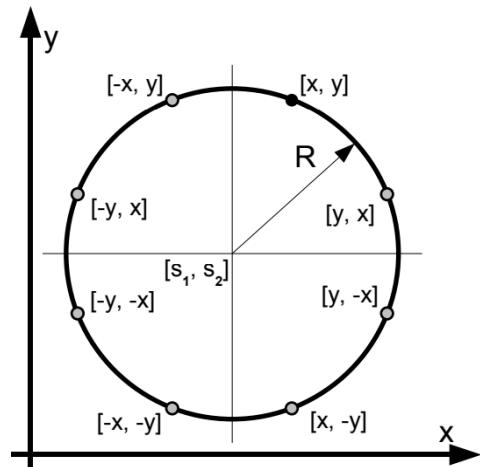
```
CircleByPoints(int cx, int cy, int R)
```

```
{  
    int x = 0, y = R;
```

```
    while (x <= y)
```

```
{  
    draw_pixel_circle(x + cx, y + cy);  
    x++;  
    y = round(sqrt(R*R - x*x));
```

```
}
```

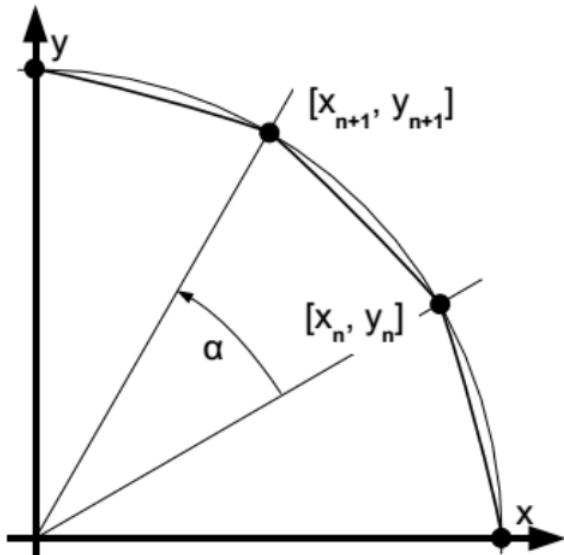


Vykreslení jako n-úhelník

Použitím DDA algoritmu se kružnice vykreslí jako by to byl **n-úhelník** (s větším poloměrem je potřeba větší n) pomocí **úseček**. Využívá k tomu matici otočení

$$x' = x \cos \alpha - y \sin \alpha$$

$$y' = x \sin \alpha + y \cos \alpha.$$

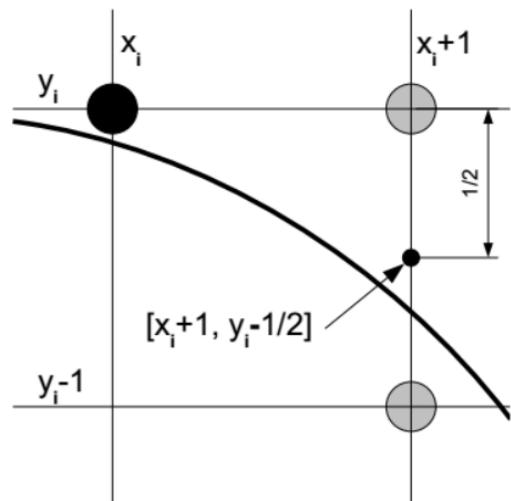


```
CircleDDA(int R, int N)
{
    double cosa = cos(2*PI/N);
    double sina = sin(2*PI/N);
    int x1 = R, y1 = 0, x2, y2;

    for (int i = 0; i < N; i++)
    {
        x2 = x1*cosa - y1*sina;
        y2 = x1*sina + y1*cosa;
        draw_line(x1, y1, x2, y2);
        x1 = x2;
        y1 = y2;
    }
}
```

Midpoint algoritmus

Bresenham ale pro kružnici. Pracuje se s celými čísly, snadná implementace.



```
CircleMid(int s1, int s2, int R)
{
    int x = 0, y = R;
    int P = 1-R, X2 = 3, Y2 = 2*R-2;

    while (x < y)
    {
        draw_pixel_circle(x, y);

        if (P >= 0)
            { P += -Y2; Y2 -= 2; y--; }

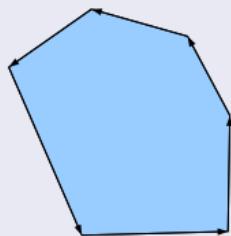
        P += X2;
        X2 += 2;
        x++;
    }
}
```

Polygon (vyplňování uzavřených oblastí)

Proces nalezení a označení (obarvení) všech vnitřních bodů dané oblasti. Vstupem je ohrazení oblasti, výstupem je rastrový popis vyplněné oblasti.

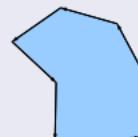
Konvexní (vypouklé, vyduté)

Pro libovolné dva body oblasti platí, že jejich spojnice je součástí oblasti, neprotíná její hranice.



Konkávní (nekonvexní, prohnuté, duté)

Oblast, která není konvexní.



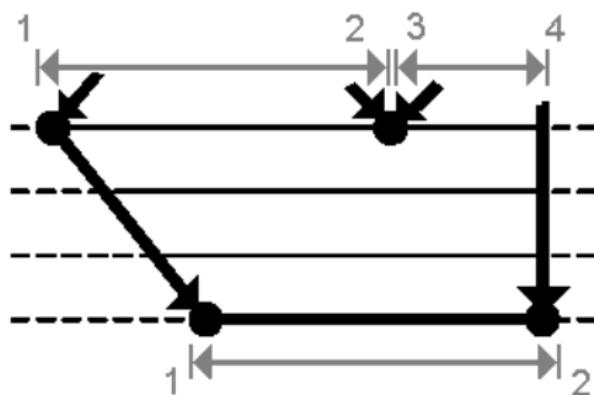
Druhy výplní: barvou (solid), šrafou (hatch), texturou, gradientem

Řádkové vyplňování (Scanline Fill)

Základní algoritmus pro vyplňování obecných mnohoúhelníků. Vstupem je **orientovaný seznam vrcholů**, výstupem vyrasterizovaný polygon (**mnohoúhelník**). Princip rasterizace je následující:

- Vyhledávají se **maximální** a **minimální** hodnoty souřadnic X a Y těchto bodů, čímž se **ohraničí oblast**, ve které se polygon nachází.
- Oblast se prochází **po řádcích** (na konci řádku se **resetuje** čítač hran).
- Počítají se průsečíky s hranami. Pokud je aktuální **počet** průsečíků **lichý**, řádek se **obarvuje**.

Problémy jsou s **lokálními extrémy** a **vodorovnými hranami**. Vodorovné hrany se **přeskakují**, problém lokálních extrémů lze řešit tak, že jsou všechny hrany **zkrácený** ve směru **osy Y** z obou stran **o 1 pixel**, nebo že je **za počítán** v jednom bodě **průsečík obou hran**.



Inverzní řádkové vyplňování

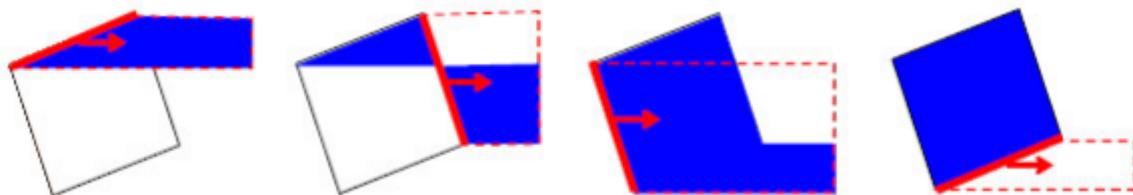
Vstupem je seznam hran nebo orientovaný seznam vrcholů. Není nutné na každém řádku testovat hrany polygonu. Postup je následující.

- Vyhledávají se **maximální** a **minimální** hodnoty souřadnic X a Y těchto bodů,

čímž se **ohraničí oblast**, ve které se polygon nachází.

- Pro každou hranu se získá souřadnice **Ymin** a **Ymax**.
- Pro každou hranu se hledají **průsečíky s jednotlivými řádky** (X-ové souřadnice), pixely **napravo** od průsečíku se invertují (0 černá, 1 bílá).
- Překreslení obrysu.

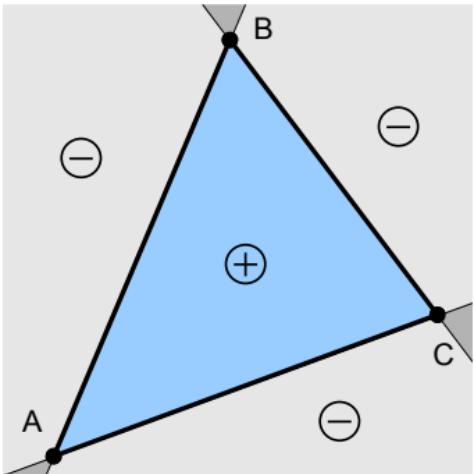
Po provedení postupu pro všechny hrany bude vybarven **pouze vnitřek** polygonu. Problém může být v zasažení oblasti mimo polygon. Umožňuje pouze binární obraz.



Pinedův algoritmus (J. Pineda, 1988)

Pracuje pouze s **konvexními** mnohoúhelníky - trojúhelníky (ty jsou **vždy** konvexní). Umožňuje snadnou realizaci v HW. Vyplňovaná oblast je popsána seznamem hran a vyplňování probíhá:

- **Rozdelení** oblasti každou hranou na **poloroviny (kladnou a zápornou)**.
- **Vybarveny** jsou body (pixely) oblasti, které leží v **kladné polorovině** všech hran.



Hranová funkce $E_i(x, y)$

Vekt. součin vektoru \vec{h}_i hrany a vektoru \vec{b}_{Pi} z počátku hrany k testovanému bodu P .

$$\begin{aligned}\vec{h}_i &= (x_{i1} - x_{i0}, y_{i1} - y_{i0}) = (\Delta x_i, \Delta y_i) \\ \vec{b}_{Pi} &= (x - x_{i0}, y - y_{i0}) \\ E_i(x, y) &= \vec{h}_i \times \vec{b}_{Pi} \\ E_i(x, y) &= (x - x_{i0})\Delta y_i - (y - y_{i0})\Delta x_i\end{aligned}$$

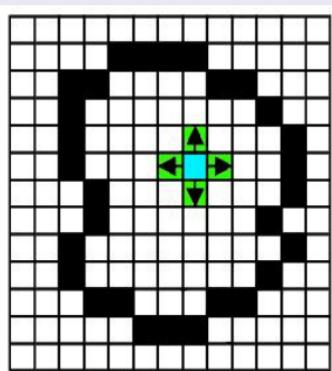
Semínkové vyplňování

Vyplňovaná oblast musí být definovaná **spojitou hranicí** z pixelů **požadované barvy**. Obravují se pouze pixely s barvou pozadí následovně:

- Semínko uvnitř oblasti **šíříme** na sousedy v okolí (**obarvování** sousedních pixelů).
- Obarvené pixely se **rekurzivně** stávají semínky (problém rekurse - může selhat, lze využít frontu a rekurzi odstranit).

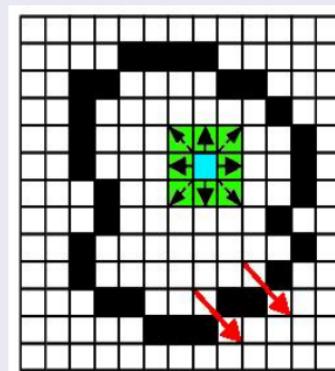
používají se dva druhy okolí **4-okolí** a **8-okolí**, 8-okolí vyžaduje lépe definovanou hranici.

4-okolí



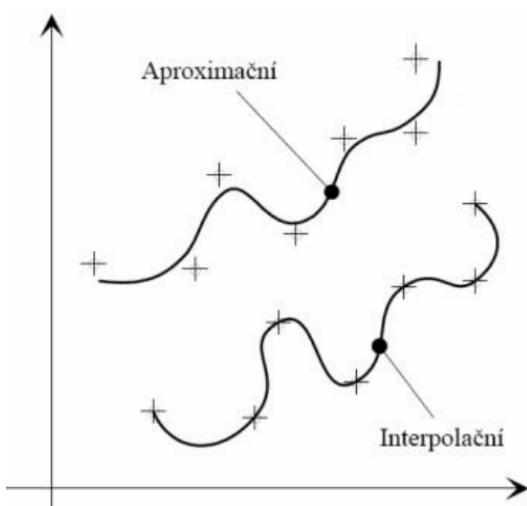
8-okolí

Vyžaduje "silnější" hranice.



Křivky

- **Interpolační** - Křivka prochází stanovenými (řídícími) body.
- **Aproximační** - Křivka nemusí procházet řídícími body - Beziérový křivky.



Beziérový křivky

- approximační křivka (2D grafika, fonty)
- polynomiální křivka využívající Bernsteinových polynomů,
- křivka stupně n určena $n+1$ body,
- prochází koncovými body.

Rekurentní definice s použitím **Bernsteinových** polynomů. Používá se pro fonty.

$$Q(t) = \sum_{i=0}^n P_i \cdot B_i^n(t); \quad i = 0, 1, \dots, n$$
$$B_0(t) = (1-t)^3$$
$$B_1(t) = 3t(1-t)^2$$
$$B_2(t) = 3t^2(1-t)$$
$$B_3(t) = t^3$$

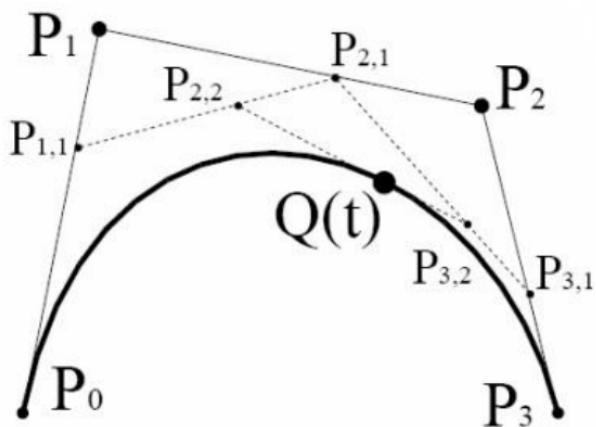
$$P(t) = P_0 B_0(t) + P_1 B_1(t) + P_2 B_2(t) + P_3 B_3(t) = \sum_{i=0}^3 P_i B_i(t)$$

Algoritmus de Casteljau

Rekurzivní algoritmus pro vykreslování **Beziérových křivek** (plyne z rekurentní definice Bernsteinových polynomů). Postup:

- Zvolí se dostatečně jemný krok, se kterým se mění hodnota t (t náleží $<0,1>$).
- Úseky polynomů se dělí v poměru t a $t-1$, v místě dělení vznikne **nový bod** pro další dělení. **Snižuje** se tak s každým krokem **stupeň polynomu**, až na konci zbyde pouze jeden **bod**.

- Vzniklé body se **spojí úsečkami** (tvar křivky závisí na použitém kroku).



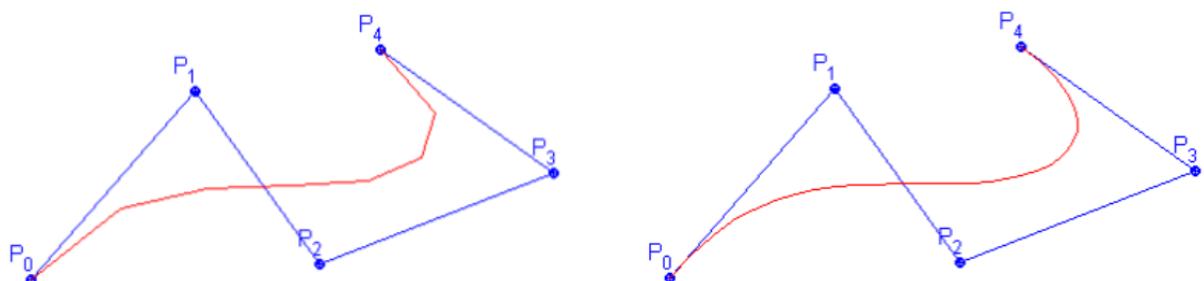
$$P_{i,j}(t) = (1-t) \cdot P_{j-1,i-1} + t \cdot P_{j,i-1}$$

$$P_{0,0} \downarrow$$

$$P_{0,1} \rightarrow P_{1,1} \downarrow$$

$$P_{0,2} \rightarrow P_{2,1} \rightarrow P_{2,2} \downarrow$$

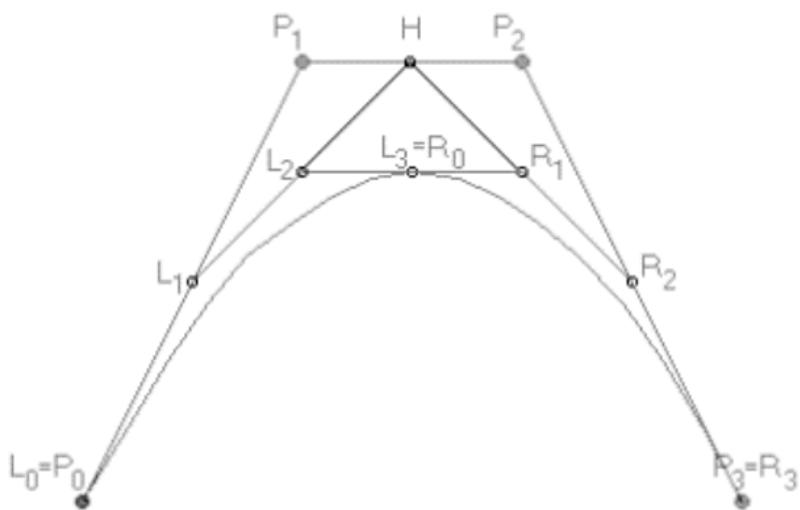
$$P_{0,3} \rightarrow P_{3,1} \rightarrow P_{3,2} \rightarrow P_{3,3}$$



Bézierovy kubiky

Beziérova křivka **3. stupně** popsaná Bernsteinovými polynomem 3. stupně.
Segment je popsaný **4 řídícími body**.

- Divide and Conquer - De Casteljau** a Bézierovy kubiky. Rekurzivní dělení na **2 podkřivky**, dostatečně rovná křivka se dále nedělí a vykreslí se.



Navazování segmentů Beziérových kubik

Vyžaduje **totožný koncový** body a **stejné tečné vektory** v navazujícím bodě.

Koncový bod je středem úsečky mezi předposledním bodem první křivky a druhým bodem druhé křivky.

Tečné vektory v koncových bodech

$$P'(\vec{0}) = 3(P_1 - P_0)$$

$$P'(\vec{1}) = 3(P_3 - P_2)$$

Podmínky spojitosti C^1

- Totožnost koncových bodů.
- Shodné tečné vektory – koncový bod úseku Q_i je středem úsečky předposledního bodu Q_i a druhého bodu Q_{i+1} .

Odkazy:

- [Modelování křivek](#)
- [Animace Bézierova křivka](#)
- [De Casteljau animace \(video\)](#)
- [De Casteljau's Algorithm and Bézier Curves](#)

12. Transformace a zobrazení 3D polygonálních modelů, principy programovatelného vykreslovacího řetězce.

Geometrická transformace

změna pozice **vrcholů** objektů v aktuálním souřadnicovém systému nebo změna souřadnicového systému

Lineární transformace

Lineární transformace **zachovává lineární kombinaci** (vektory je možné aplikovat **zároveň nebo po jednom** a výsledek bude vždy stejný). Pro **libovolné** dva vektory a **skalár** platí:

$$\begin{aligned} f(\vec{x_1} + \vec{x_2}) &= f(\vec{x_1}) + f(\vec{x_2}), \\ f(\alpha \vec{x_1}) &= \alpha f(\vec{x_1}). \end{aligned}$$

Mezi lineární transformace patří **měřítko** (zvětšení, zmenšení), **rotace** a **zkosení**.

Afnní transformace

Afnní transformace zachovává **kolinearitu a dělící poměr** (body ležící na přímce budou ležet na přímce - v jednom bodě; i po zobrazení). Všechny základní geometrické operace (**měřítka Sc**, **rotace R**, **zkosení Sh** i **posunutí T**) jsou afnní. Lze ji vyjádřit jako lineární transformaci následovanou posunem.

Homogenní souřadnice (váha)

Ke standardním **kartézským** souřadnicím (x, y ve 2D a x, y, z ve 3D) je přidána jedna navíc - souřadnice **w** (váha bodu, u **afnních je w = 1**). Umožňují pracovat se **všemi** druhy základních **transformací** jednotně pomocí **maticového zápisu**.

Maticový zápis umožňuje provádět jednoduché skládání transformací.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

- **Posunutí** (translace): Ve **3D** stačí rozšířit o řádek a sloupec s **dz**, resp. **-dz**.

Maticový zápis transformace

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{bmatrix}$$

$$P' = P \cdot T$$

Maticový zápis inverzní transformace

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -d_x & -d_y & 1 \end{bmatrix}$$

$$P' = P \cdot T^{-1}$$

- **Změna měřítka** ve **3D** (scale): $P' = P \cdot S$, ve **3D** stačí rozšířit o řádek s **Sz**, resp. **1/Sz**.

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad S^{-1} = \begin{bmatrix} 1/S_x & 0 & 0 \\ 0 & 1/S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Zkosení**: **1** matice pro zkosení ve **2D**, **3** matice pro zkosení ve směrech **YZ**, **XZ** a **XY** ve **3D**.

$$S_H = \begin{bmatrix} 1 & S_{hy} & 0 \\ S_{hx} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad S_H^{-1} = \begin{bmatrix} 1 & -S_{hy} & 0 \\ -S_{hx} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Rotace kolem počátku souřadného systému:** Ve 3D 3 matice

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 & 0 \\ -\sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[x', y', 1] = [x, y, 1] \cdot \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R^{-1} = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P' = P \cdot R$$

$$S_{HYZ} = \begin{bmatrix} 1 & S_{hy} & S_{hz} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad S_{HXZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ S_{hx} & 1 & S_{hz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_{HXY} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ S_{hx} & S_{hy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Rotace ve 3D kolem obecné osy:** dána směrovým vektorem \mathbf{v} a bodem umístění \mathbf{P} , je třeba rozdělit na posloupnost transformací:
 - 1. Posunutí osy rotace do počátku souřadného systému
 - 2. Sklopení posunuté osy do jedné ze souřadných rovin
 - 3. otočení sklopené osy do jedné ze souřadných os (např. do X)
 - 4. Provedení požadované rotace o úhel ω kolem příslušné osy (zde X)
 - 5. Vrácení osy rotace do původní polohy

$$M = T \cdot R_X \cdot R_Z \cdot R_{X(\omega)} \cdot R_Z^{-1} \cdot R_X^{-1} \cdot T^{-1}$$

Zkráceně musíme nejdřív obecnou osu dostat do pozice jedné z os souřadného systému, provést otočení a vrátit osu do původního místa.

Skládání transformací

U skládání transformací závisí na jejich pořadí, např. **posunutí, rotace, zvětšení**.

- **zápis bodu v řádku:** $\mathbf{P}' = \mathbf{P} \cdot \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$, matici M souhrnné transformace získáme jako $\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$ a $\mathbf{P}' = \mathbf{P} \cdot \mathbf{M}$.
- **zápis bodu ve sloupci:** $\mathbf{P}' = \mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T} \cdot \mathbf{P}$, matici M souhrnné transformace získáme jako $\mathbf{M} = \mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T}$ a $\mathbf{P}' = \mathbf{M} \cdot \mathbf{P}$.

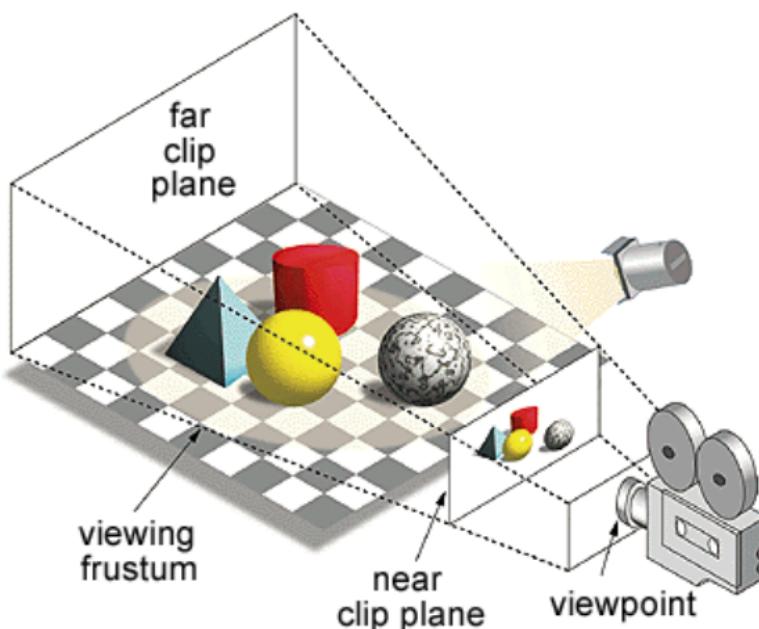
V obou případech musí být první transformace v pořadí **nejblíž transformovanému bodu**.

zobrazení 3D polygonálních modelů

Zobrazení **3D** objektů provádíme většinou na **2D** obrazovku **projekcí** (transformací z 3D do 2D, to znamená ztrátu dat), obrazovka je tedy **průmětna**. Promítání provádíme pomocí paprsků, tzv. **projekčních paprsků**. V počítačové grafice se většinou rasterizují všechny objekty **pomocí trojúhelníku** (konvexnost, lze dobře akcelerovat v HW).

Prvky 3D scény

- **near clip plane:** určuje minimální hloubku zobrazovaných objektů,

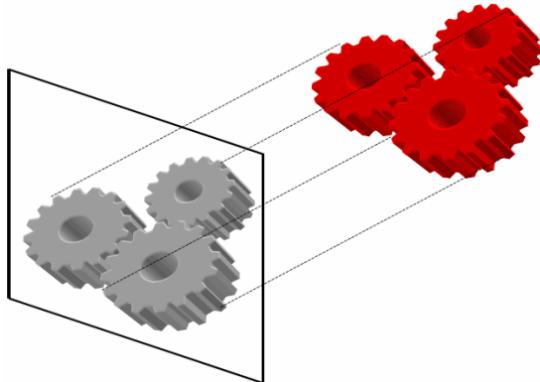


- **far clip plane:** udává maximální hloubku zobrazovaných objektů,
- **viewing frustum:** prostor v modelovaném prostředí, který se objeví na obrazovce,
- **viewpoint:** místo, ze kterého scénu pozorujeme (kamera)
- **světlo:** bodové nebo plošné, umístění rozhoduje o viditelnosti objektů (barvy, stíny, ...),
- **modelované objekty.**

Paralelní projekce (**rovnoběžná**, ortografická)

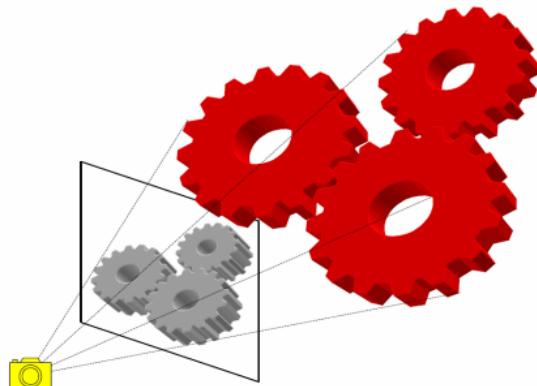
- **zachovává rovnoběžnost hran**

- **vzdálenost** průmětny **neovlivňuje velikost** průmětu (při změně vzdálenosti objektu se nemění jeho velikost)
- **kolmé promítání** - paprsky jsou **kolmé na průmětnu**
- použití: technické **CAD** aplikace, výkresová dokumentace



Perspektivní projekce (středová)

- **nezachovává rovnoběžnost** hran
- použití ve **hrách, VR**, většinou
- **vzdálenost** průmětny **od objektu ovlivňuje velikost** průmětu (se vzdáleností objektu se objekt zmenšuje)
- nelineární středová projekce: paprsky **vycházejí z 1 bodu** - **středu projekce**
- pro jednodušší manipulaci se **kamera** obvykle **zafixuje** do počátku souřadného systému a **hýbe** se (pomocí transformačních matic) se **scénou**.
 - **Geometrický princip projekce**: z vrcholů trojúhelníku se do středu projekce (tj. střed souřadného systému, kam je umístěna i kamera) vrhnou paprsky (u perspektivní projekce se všechny paprsky sbíhají do středu projekce). V tom místě, kde paprsek protnul projekční rovinu se promítne bod, ze kterého paprsek původně vyšel.



RayTracing

Metoda pro realistické zobrazování. Funguje na principu zpětného sledování cest paprsku od kamery ke zdroji světla. Hledá se množství světla, které paprsek přináší.

Paprsky dělíme na:

- **Primární:** Vychází z kamery (pixely obrazovky), mají společný počátek nebo jsou rovnoběžné
- **Stínové:** Z místa dopadu paprsku do každého světla, zajímá nás, jestli jej vidíme nebo nevidíme, podle toho následně spočítáme stín.
- **Sekundární:** Vznikají **odrazem a lomem** z míst **dopadů primárních** a sekundárních paprsků. Jsou chaotičtější než primární paprsky.

Výpočetně je RayTracing velmi náročný, lze optimalizovat například ohledem na to, že **sekundární** paprsky mají **menší vliv** na výsledný obraz, **intenzita světla** se snižuje se **čtvercem** vzdálenosti, **snížit rozlišení**.

Radiozita

Metoda **globálního osvětlení scény**. Řeší **šíření energie**, objekty mohou být sekundárními zdroji světla (odraz).

- Vychází ze zákona zachování energie, vyžaduje energeticky uzavřenou scénu.
- Scéna musí být reprezentovaná **polygonálním modelem** (jiné modely - CSG, B-rep, Drátové modely).
- Vychází z dvousměrové distribuční **funkce BRDF**.
- Před vlastním výpočtem je třeba **polygony** ve scéně **rozdělit** na **malé plošky** a spočítat **konfigurační faktory** (**vliv** každé **plošky** na každou **jinou plošku** ve scéně).
- Nedokáže pracovat s průhlednými objekty.

Principy programovatelného vykreslovacího řetězce (zobrazovací pipeline)

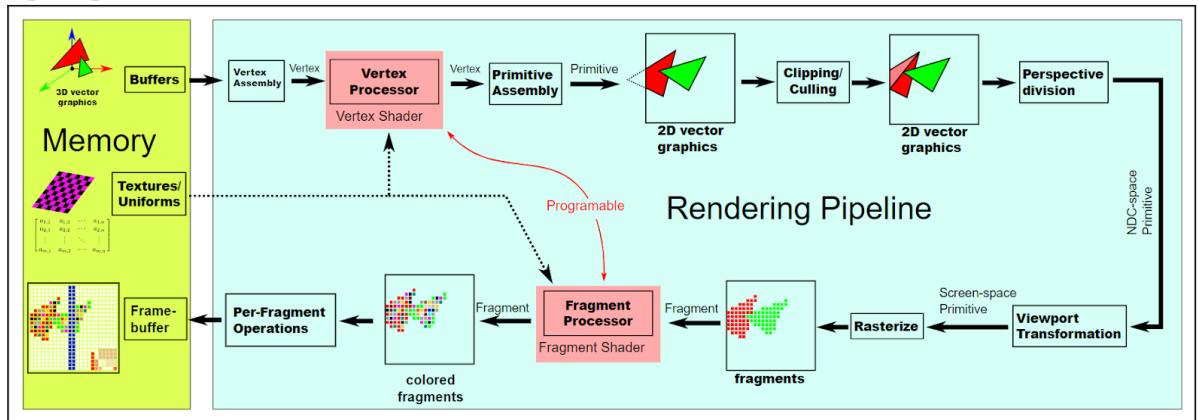
zobrazovací pipeline je tvořena těmito částmi:

1. **Vertex Assembly (Vertex Puller):** je zařízení na grafické kartě, které se stará o **sestavení vrcholů**. Vertex puller je tvořen **čtecími hlavami**, každá konstruuje jeden atribut vrcholu (pozice, normála, souřadnice textury, barva, ...). Čtecí hlavy se pohybují s krokem (**stride**) a mají nějaký posun (**offset**). Data čtou z pole bytů.
2. **Vertex Shader** (programovatelný): Provádí zpracování vrcholů z Vertex Assembly. Jedná se o násobení **modelovou maticí**, **view maticí** a **projekční maticí**. Vstupem jsou vrcholy v **model space**, výstupem vrcholy v **clip space**.
3. **Primitive Assembly:** je jednotka, která sestavuje trojúhelníky. **Čeká na 3** po sobě jdoucí **vrcholy** z vertex shaderu a **sestaví trojúhelník**. Lze na to také nahlížet tak, že Primitive Assembly dostane příkaz vykreslit třeba 4 trojúhelníky. Jednotka tak spustí Vertex Shader 12x, který takto spustí 12x Vertex Assembly.
4. **Clipping/Culling:** provádí **ořez prostoru** na view frustum. Trojúhelníky na

hranicích musí ořezat (může vést na rozdelení na 2) a zahazuje trojúhelníky, které nejsou vůbec vidět (odvrácené, překryté).

5. **Perspective Division:** provádí převod z homogenních souřadnic na kartézských na základě hloubky trojúhelníků (dělením - co je daleko se zmenší více, co je blízko se zmenší méně).
6. **Viewport Transformation:** Převádí souřadnice z NDC (normalized device coordinates, **-1, +1**) na rozlišení okna, aby se mohla provést rasterizace.
7. **Rasterization:** rasterizuje připravené trojúhelníky a produkuje **fragmenty** (čtvercové úlomky trojúhelníku - **pixely**, které se nakonec **zapíší** do **framebufferu**).
8. **Fragment Shader (programovatelný):** **obravuje fragmenty** uvnitř trojúhelníka (střed pixelu musí ležet uvnitř). Pro konkrétní fragment (pixel trojúhelníku) se spočítají **barycentrické souřadnice** a použijí se pro **interpolaci** všech atributů z **vrcholů**.
9. **Per-Fragment Operations:** jedná se o dvě operace **hloubkový test** a **blending**. Hloubkový test se stará o **zahazování fragmentů**, které jsou **hlouběji** než to, co už se **vyrasterizovalo**, naopak pokud je **hloubka** nového fragmentu **menší**, je jeho **barva a hloubka zapsána do framebufferu**. **Blending** místo přepsání barvy ve framebufferu je míchá. Existuje mnoho způsobů realizace, např. pomocí **průhlednosti (alpha blending)**. Obsah framebufferu je poté zobrazen.

GPU



13. Principy grafických uživatelských rozhraní (komunikační kanály, módy komunikace, systémy řízené událostmi, standardní prvky rozhraní).

Komunikaci člověka se strojem lze definovat jako **obousměrnou výměnu informací** mezi člověkem a strojem. Tok informací od počítače k člověku lze uskutečnit pomocí **periferních** výstupních zařízení generujících výstupy, na které mohou reagovat **smysly člověka**. Tok údaj od člověka k počítači lze naopak uskutečnit pomocí vstupních periferních zařízení, která mohou **reagovat na podněty** generované člověkem.

Komunikační kanály

Komunikační kanály jsou způsoby **komunikace člověka a stroje** pomocí periferních zařízení - **monitor, klávesnice, tiskárna, myš, kamera, ...** Jsou **založené na lidských smyslech**.

Komunikační kanály od stroje k člověku

- **Obraz (zrak)** - Nejvýhodnější pro přenos informace. **Nejvyšší informační propustnost**.
- **Zvuk (sluch)** - Vhodný pro přenos menšího počtu informací. Nižší propustnost a "sériový" přístup (je obtížné poslouchat 2 věci zároveň). Zvuk na sebe však může **lépe upozorňovat** - varovné signály.
- **Hmat** - Využívá se pro komunikaci **nevidomých** a stroje (braillovo písmo). Běžní uživatelé se nejčastěji setkávají s tímto druhem komunikace na mobilních zařízeních, formou **vibrací** (náhrada za pocit stisknutí tlačítka) a také na ovladačích her (např. signalizace překážky, **silový odpór** u závodního volantu).
- **Čich, chut'** - V současnosti **nejsou** pro komunikaci **použitelné**. Problémy s umělým syntezováním chuti a vůně v reálném čase.

Komunikační kanály od člověka ke stroji

- **Pohyb (Hmat)** - Nejobvyklejší prostředek - **klávesnice**, tlačítka, přepínače, ... a **mechanické polohovací zařízení** - **myš**, páčky. (Dnes nejpoužívanější, ale neznamená to, že je nejlepší, pouze je obvykle nejjednodušší na implementaci - např. diktování textu může být pro většinu uživatelů příjemnější, než psaní na klávesnici).
- **Zvuk (Řeč)** - Rychle se vyvíjí, dnes už prakticky **použitelný** způsob ovládání, zejména v anglickém jazyce (asistenti na mobilu, speech to text apod.). **Problém** s rozpoznáváním řeči a náročnost zpracování, **soukromí**. Není

vhodné pro sdělování citlivých informací (hesla, číslo OP., ...)

- **Obraz (gesta)** - Sdělování informací **gesty, pohyby a mimikou v obličeji** (filtry na Instagramu). Lze využít ve **virtuální realitě** a hrách (Just dance, Wii games...), gesta ruky pro přeskočení písničky na Android 10 (Google Pixel). Rozpoznávání osob a objektů - pro bezpečnost nebo klasifikaci.

Módy komunikace

Nejzákladnějším dělením obrazové komunikace člověka s počítačem je dělení podle aktivity uživatele:

- **Aktivní komunikace** - Uživatel **řídí** činnost počítače - činnost počítače záleží na **vůli uživatele**. Například panel nástrojů v grafickém editoru, unixový terminál...
- **Pasivní komunikace** - Uživatel **odpovídá** na dotazy počítače.
Dialogová/modální okna.

Je zřejmé, že komunikace s počítačem se **nebude trvale** odehrávat jen **pasivně** nebo jen **aktivně**, a že je vhodné oba způsoby **kombinovat**. **Aktivní** komunikaci je nutné používat tam, kde uživatel **musí rozhodnout**, jakou činnost má počítač vykonávat a kdy není možné rozhodnutí obsluhy předvídat. **Pasivní** komunikace je naopak vhodná tehdy, je-li příští **činnost** počítače **známá**, a obsluha jen zadává údaje nebo rozhoduje mezi několik málo možnými variantami, které lze přesně určit.

Vlastnost	Aktivní komunikace	Pasivní komunikace
Efektivní práce	? dle aplikace	- často neefektivní
Učení	- obtížné	+ snadné
Tvůrčí práce	+ snadná	- obtížná
Možnost chyby	- velká	+ malá

Módy komunikace - Stav ve kterém počítač reaguje jedinečným způsobem na vstup od uživatele (změny módů - změny chování počítače na vstupy, na klávesnici **Caps Lock, Insert**, grafický editor a levé tlačítko myši může kreslit, mazat, přesouvat objekty). Obecně čím méně modů tím lípe. Typické módy (např. Vim - i pro editaci):

- Zadání příkazu v příkazovém jazyce
- Odpověď na dotaz (potvrzení akce - např. smazání).
- Editace textu
- Reakce na chybu

Přímá manipulace (Drag and Drop, Look and Feel)

Umožňuje interakci mezi uživatelem a objekty zobrazenými na obrazovce - podporuje **přirozené chování** uživatele. **Zjednodušuje ovládání** počítače pro

neškolené uživatele a zlepšuje jejich dojem při práci s PC. Zjednoduší nároky na zkušeného uživatele, který nemusí vynakládat úsilí např. pro přesun souboru, ale provede jej intuitivně pomocí **Drag and Drop**. **Look and Feel** - uživatelské rozhraní se dá používat **intuitivním způsobem** vycházejícím z podobnosti s prací s předměty v běžném životě.

Systémy řízené událostmi

Systémy, které se zabývají **detekcí, zpracováním a reakcí** na události. Tok programu je řízen různými událostmi (vstup z periferií - **zmáčknutí klávesy, pohyb myši**). Program naslouchá na tyto akce, a nějak reaguje (např. v JS pomocí listeners: onclick, onhover, onkeypress). U hardware je toto prováděno pomocí **přerušení**. Tento přístup je velmi využívaný. Programy, které takto pracují, běží v nekonečné smyčce (z té vystoupí po události signalizující ukončení) a čekají na příchod události (HW přerušení) nebo sami kontrolují její vznik (Polling), nebo probíhá formou zasílání zpráv. Detekuje se o jakou se jedná událost a spustí se její obsluha, po dokončení obsluhy se program dostává zpět do stavu, ve kterém čeká na další událost.

- QT: signal and slots,
- callbacks

Standardní prvky rozhraní

Standardní prvky GUI obvykle označujeme pod zkratkou **WIMP**:

- **Window - okno**: reprezentují spuštěné programy:
 - **primární**: hlavní okno aplikace, obsahuje menu, více nezávislých funkcí,
 - **sekundární**: okno pro zpracování vedlejších, rozšiřujících či doplňkových funkcí, pevný rozměr, nemá min/maximalizaci, menší než primární okno. Může být nazývané jako **modální**, protože **mění mód/režim** chování aplikace/systému, mění pracovní postup. Může vyžadovat interakci uživatele, než vrátí řízení rodičovskému oknu (např. dialogová okna). Slouží k upozornění uživatele a blokování aplikace/systému pro získání klíčových informací (uživatel musí dokončit práci v modálním okně)
 - **modální** - dialog v rámci aplikace,
 - **systémově modální** - Mají prioritu nad aplikacemi (např. nedostatečná práva pro provedení akce, **potvrzení instalace**).
 - **nemodální** - Neprioritní. Uživatel může mít okno otevřené a stále pracovat s aplikací (výběr barvy štětce v malování například).
- **Icon - ikona**: reprezentují zkratky sloužící k provedení určité činnosti (například spuštění programu).
- **Menu**: textové nebo z ikon složené **nabídky**, ze kterých je možné jednu vybrat a provést tak určitou akci.

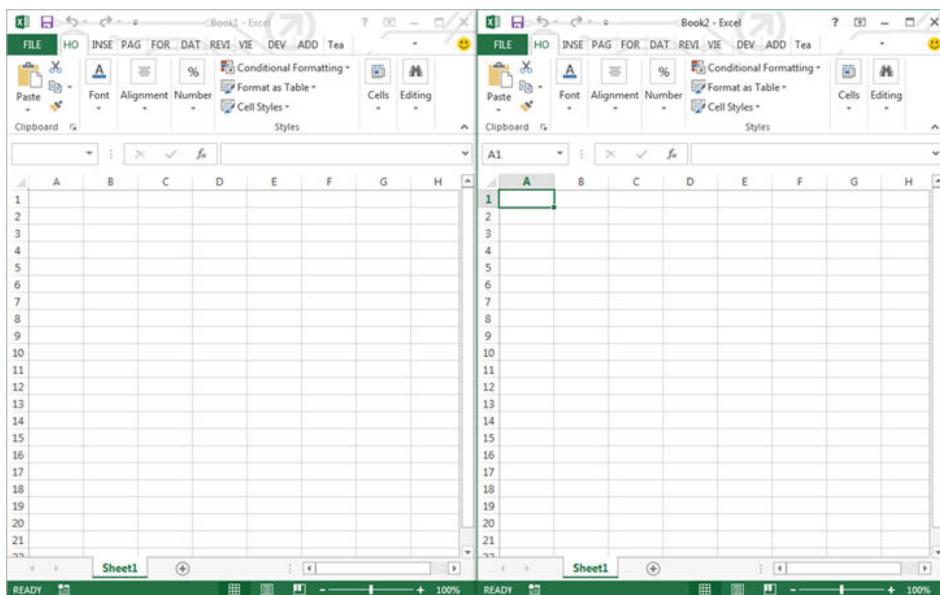
- **Pointer - ukazatel:** pohybující se grafický symbol reprezentující pohyb fyzického zařízení (myš), pomocí něhož uživatel vybírá ikony, položky v menu nebo data.

Další prvku UI

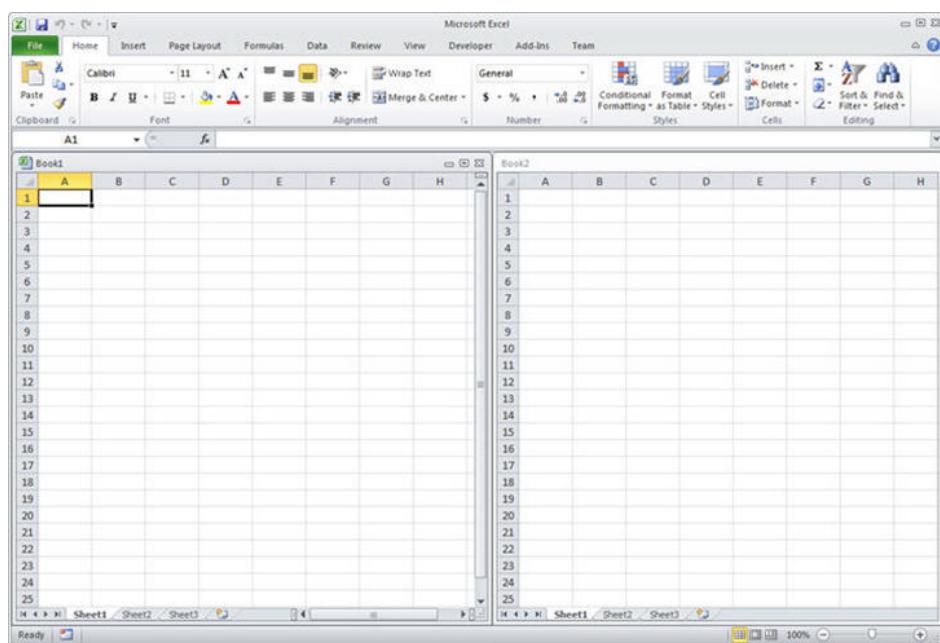
- tlačítka,
- přepínače,
- popisky,
- checkboxy,
- seznamy,
- slidery,
- výběr souboru,
- vstupní pole pro text

Režim oken aplikace

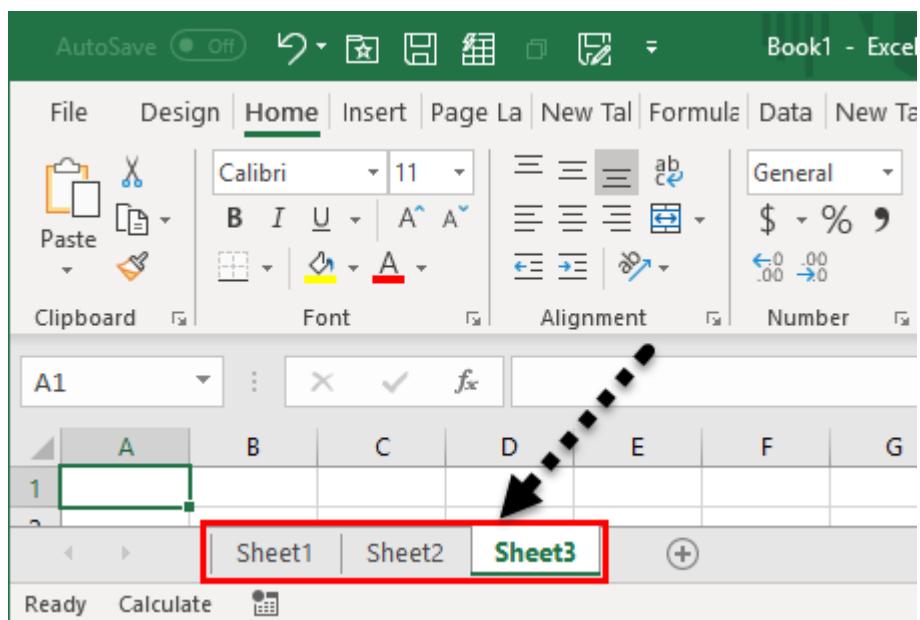
- **Single-Document Interface (SDI):** Jedno primární okno, několik sekundárních. Jednoznačný vztah okna s objektem. Přehledné, srozumitelné.
Každé okno má své menu. Několik instancí aplikace v liště OS.
(Excel 2013)



- **Multiple-Document Interface (MDI):** Jedna aplikace obsahuje více oken. Práce s jedním objektem z více pohledů (otevřený ve více oknech) nebo práce s více objekty současně (možnost při práci s jedním nahlížet na druhý). Menší přehlednost a obtížnější zvládnutí.
(Excel 2010)
- **Tabbed Document Interface (TDI)** - prohlížeče, editory zdrojových kódů, ...



Dnes často používané. Řízené SDI - SDI doplněné o řídící okno, které obsahuje menu a seznam otevřených objektů (záložek). Kooperativní SDI - některé funkce mohou ovlivnit obsah i jiných oken.
(jednotlivé dokumenty v záložkách)



Odkazy:

https://www.fit.vutbr.cz/study/courses/ITU/private/lectures/zaklady/itu-zaklady_GUI_a_historie.pdf

14. Spektrální analýza spojитých a diskrétních signálů.

Signál je záměrný fyzikální jev, nesoucí informaci o nějaké aktuální události. Jedná se o časově či prostorově proměnnou prostředí (**fyzikální veličina**).

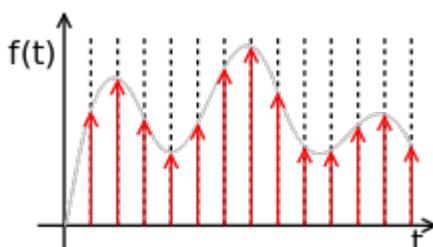
Spojitý (analogový)

Vyskytují se **v reálném světě** (například zvukové vlny), mají pro každý časový okamžik určitou hodnotu, což tvoří souvislou (spojitou) křivku. Lze je zapsat funkcí. Značíme je $x(t)$ a hodnoty spojitého signálu jsou z množiny **reálných čísel**.

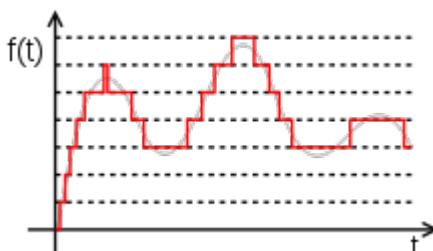
Diskrétní

Jeho okamžitá hodnota se nemění spojite s časem ale **skokově**. Lze je ukládat a zpracovávat na **číslicových** počítačích. Značíme je $x[t]$ a hodnoty diskrétních signálů jsou pouze z množiny **celých čísel** a navíc nabývají některé z **konečného počtu hodnot**. Pro získání diskrétního signálu **vzorkujeme** signály kolem nás, které jsou spojité. Následně signály **kvantujeme** - zaokrouhlujeme na čísla z oboru hodnot signálu (respektive na čísla, která dokáže reprezentovat v PC).

- **Vzorkování:** Periodicky zaznamenáváme hodnotu spojitého signálu. Periode zaznamenávání spojitého signálu musí být aspoň dvakrát menší, než nejmenší perioda některé ze složek spojitého signálu (**Nyquistův–Shannonův vzorkovací teorém**, jinak nelze signál rekonstruovat).



- **Kvantování:** diskretizace oboru hodnot signálu (převod z reálných čísel na celá). Je to obecně proces ztrátový a nevratný.



Deterministické signály

Pro **každý spojitý čas či diskrétní čas** přesně víme, jakou **hodnotu** bude signál **mít**. Lze je definovat funkcí.

Náhodné signály

Nelze je popsat rovnicí. Pro část (t) nebo [n] **nikdy** přesně **nevíme**, jaká bude jejich **hodnota**. Popisují se pomocí parametrů jako **střední hodnota** nebo **rozptyl**.

Periodické

Průběh signálu se opakuje s určitou periodou. $s(t+T) = s(t)$.

- **Harmonické** - Nejjednodušejí definované periodické signaly. Mají tvar $x(t) = A \cdot \cos(\omega t + \varphi_0)$,
 - A - Amplituda. Maximální hodnota periodické funkce.
 - ω - úhlový nebo kruhový kmitočet [rad/s] pro spojitý a [rad] pro diskrétní.
 - φ - počáteční fáze [rad]. Posunutí funkce po ose x.

Spektrální analýza

Spektrální analýzou signálů zjišťujeme jejich **frekvenční charakteristiku** (zajímají nás):

- **kde** jsou frekvenční komponenty signálu (na jakých frekvencích se signál nachází)
- **kolik** je na které frekvenci signálu (amplituda)
- jak je na které frekvenci signál **posunutý** (fázový posun).

Rozklad signálu na **sinusovky** a **cosinusovky**. Převod signálu z časové oblasti do frekvenční oblasti. Výsledkem spektrální analyzy je **spektrum**.

Fourierova řada Fourier Series

Každý periodický signál lze vyjádřit jako součet sinusovek a cosinusovek.

Frekvence jednotlivých sinusovek a cosinusovek jsou **celočíselným násobkem** frekvence **původního** signálu. **Fázi** a **amplitudu** každé harmonické složky (sinusovky a cosinusovky), lze získat **furiérovým rozkladem** (výpočet koeficientů a_n a b_n). Fourierova řada tedy umožňuje popsat původní periodický signál součtem **sinusovek a kosinusovek**.

$$f(t) = a_0 + \sum_{n=1}^N a_n \cos n\omega_0 t + \sum_{n=1}^N b_n \sin n\omega_0 t$$

$$\begin{aligned}
a_0 &= \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) dx, & a_0 &= \frac{2}{T} \int_0^T f(t) dt, \\
a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos nx dx \quad (n = 1, 2, 3, \dots) & a_k &= \frac{2}{T} \int_0^T f(t) \cos(k\omega t) dt, \quad k \geq 1, \\
b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin nx dx \quad (n = 1, 2, 3, \dots) & b_k &= \frac{2}{T} \int_0^T f(t) \sin(k\omega t) dt, \quad k \geq 1.
\end{aligned}$$

$$\cos(\omega) + i \sin(\omega) = \left(\cos\left(\frac{\omega}{n}\right) + i \sin\left(\frac{\omega}{n}\right) \right)^n$$

Euler now applies the limit $n \rightarrow \infty$:

$$\cos(\omega) + i \sin(\omega) = \lim_{n \rightarrow \infty} \left(\cos\left(\frac{\omega}{n}\right) + i \sin\left(\frac{\omega}{n}\right) \right)^n$$

using small angle approximations $\cos(x) \approx 1$ and $\sin(x) \approx x$:

$$\cos(\omega) + i \sin(\omega) = \lim_{n \rightarrow \infty} \left(1 + \frac{i\omega}{n} \right)^n = e^{i\omega}.$$

In the last line he applied the limit representation $e^x = \lim_{n \rightarrow \infty} (1 + \frac{x}{n})^n$.

Fourierova řada může být také zapsána jako součet komplexních exponenciál.

- **Vstup:** Spojitý periodický signál (funkce).
- **Výstup:** Koeficienty určující **amplitudy** a **fáze** komplexních exponenciál na **násobcích** základní **frekvence** - diskrétní hodnoty. Z **koeficientů** můžeme následně původní signál **znovu sestavit** pomocí vzorce pro **FŘ**.

Důležité je, že **FŘ** reprezentuje pořád ten **samý signál**, my ale z toho původního potřebujeme znát právě jeho **koeficienty**, které nám o něm dívají informace.

$$\boxed{x(t) = \sum_{k=-\infty}^{+\infty} c_k e^{jk\omega_1 t} \quad \boxed{c_k = \frac{1}{T_1} \int_{T_1} x(t) e^{-jk\omega_1 t} dt,}}$$

Vzorec pro výpočet fourierovy řady
- vzorec pro výpočet koeficientů Fourierovy řady

Diskrétní Fourierova řada - Discrete Fourier Series

Diskrétní Fourierova řada je tvořena **konečným počtem** součtů sinusovek a kosinusovek. Jedná se o aproximaci FŘ. **Signál musí** být stále **periodický**. Umožňuje nám provádět výpočty v praxi, protože data na počítačích nejsou **nikdy spojité** a navíc nedokázeme provést nekonečné množství součtů.

- Vstup: **Diskrétní periodický** signál s periodou **N**.

- Výstup: Koeficienty určující **amplitudu** a **fáze** komplexních exponenciál na **násobcích základní frekvence**, které se **periodicky** opakují s periodou **N**.

The periodic sequence x can be represented

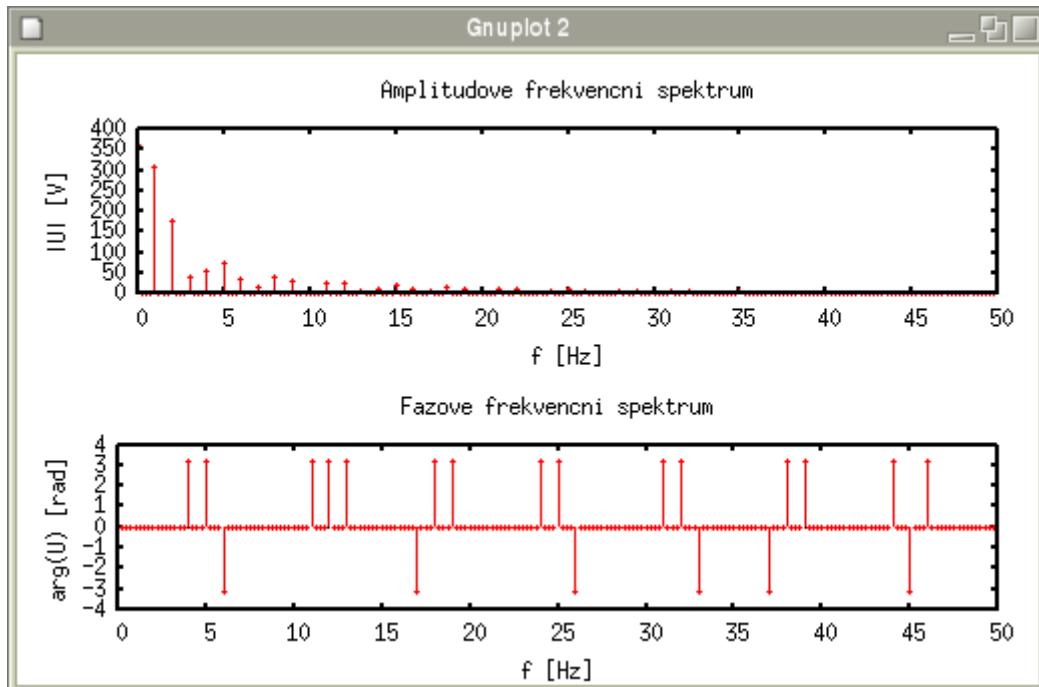
as a sum of N complex exponentials with frequencies $k\frac{2\pi}{N}$, where $k = 0, 1, \dots, N-1$:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{jk\frac{2\pi}{N}n} \quad (1)$$

DFS coefficients are obtained as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}.$$

for all times n , where $X[k] \in \mathbb{C}$ is the k^{th} DFS coefficient corresponding to the complex exponential sequence $\{e^{jk\frac{2\pi}{N}n}\}$.

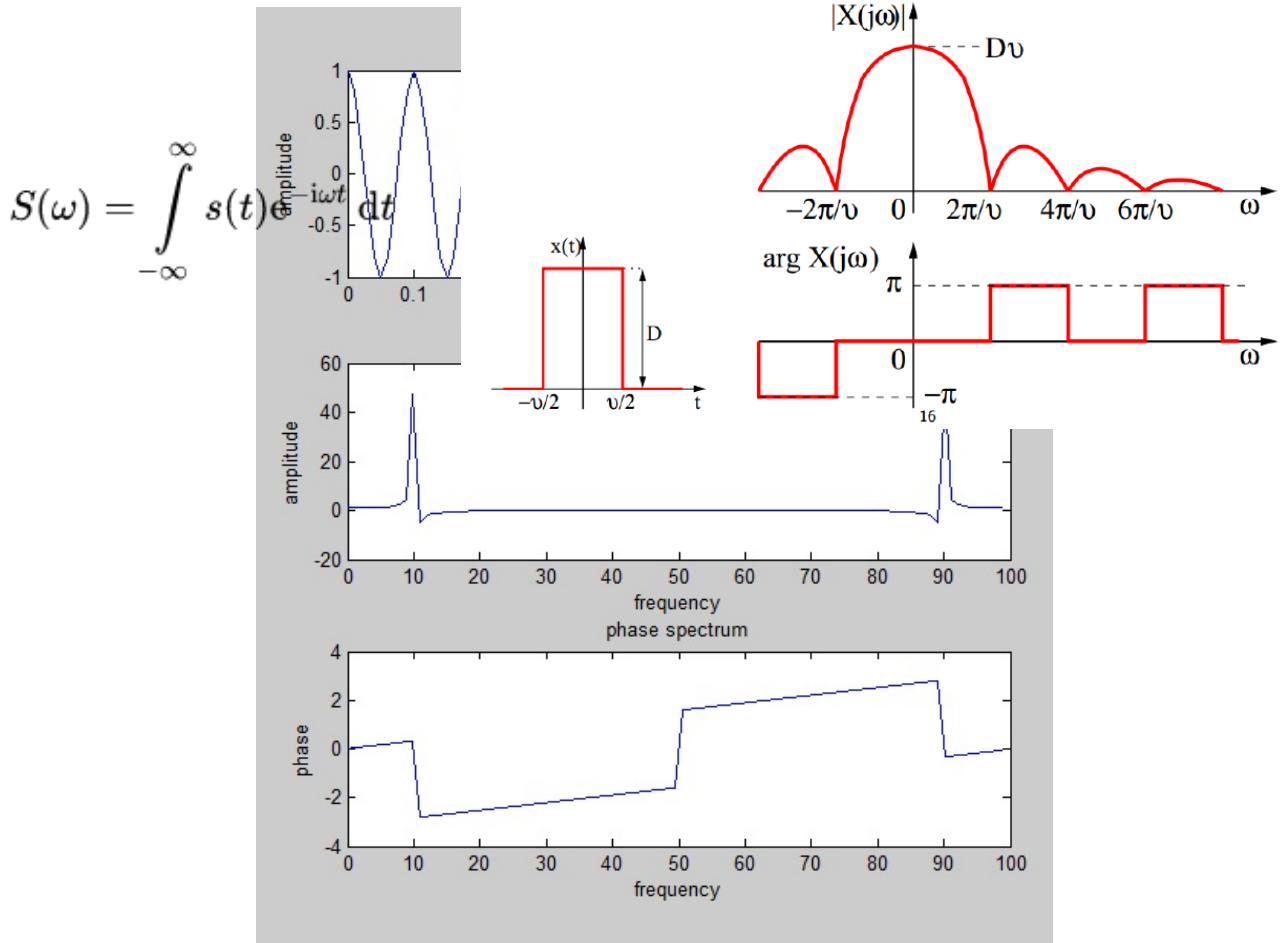


Fourierova transformace - Fourier Transform

Zobecnění FŘ pro **neperiodické** signály. Slouží pro **převod** (transformaci) signálů z **prostorové** nebo **časové** oblasti **do** oblasti **frekvenční**. Umožňuje dekomponovat signál na **jednotlivé frekvence**, které ho tvoří.

- **Vstup** - Obecný spojitý signál (**nemusí** být periodický).

- **Výstup** - je řada komplexních čísel, z nichž každé odpovídá frekvenci, amplitudě a fázi ve výsledném frekvenčním spektru. (někdy lze využít pro získání koeficientů FŘ <https://lpsa.swarthmore.edu/Fourier/Xforms/FXFS.html>, pokud obsahuje jednu periodu signálu a jinak 0 - lze ztotožnit se získáváním koeficientů FŘ)



Fourierova transformace s diskrétním časem - Discrete Time Fourier Transform

Zobecňuje DFŘ pro neperiodické signály. Slouží pro převod (transformaci) signálů z prostorové nebo časové oblasti do oblasti frekvenční.

- **Vstup:** Nekonečno vzorků obecného diskrétního signálu (nemusí být periodický).
- **Výstup:** Spojitá periodická funkce ve frekvenční oblasti, což není na PC použitelné. Diskrétní je pouze v případě periodického signálu.

$$X_{2\pi}(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-i\omega n}.$$

Diskrétní Fourierova transformace - Discrete Fourier Transform

Slouží pro **převod** (transformaci) signálů z **prostorové** nebo **časové** oblasti **do** oblasti **frekvenční**. **Vzorkuje výstup DTFT** (fourierova transformace s diskrétním časem), který je spojitý. Pracuje s **kvadratickou O(n^2)** časovou složitostí.

- **Vstup:** N vzorků obecného neperiodického signálu (DFT bere těchto N vzorků, jako by se periodicky opakovaly)
- **Výstup:** N komplexních koeficientů (komplexních exponenciál) které určují **amplitudy a fáze frekvence**. Pro PC tedy ideální situace.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N} kn} \\ &= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi}{N} kn\right) - i \cdot \sin\left(\frac{2\pi}{N} kn\right) \right], \end{aligned}$$

Fast Fourier Transform

modifikace DFT, která je rychlejší, pracuje s **linearitmickou O(n*logn)** časovou složitostí. Lze počítat pouze pro **počty vzorků**, které jsou rovny mocnině 2 (2^n). Jedná se dnes o jeden z **nejpoužívanějších algoritmů vůbec**. Např.:

- **Komprese** JPEG, MP3, ...: Fungují na principu výpočtu FFT a následně **odstranění frekvencí**, které jsou zastoupeny **minimálně** - což je většina, třeba i **99%**. Jedná se o ztrátovou kompresi (nelze praktici rozeznat), ale **nezbytnou**.
- **derivace**
- **filtrace dat**: pomocí FFT nalezneme frekvence, které tvoří rušení a odstraníme je.
- **konvoluce**: pomocí FFT se převedou signály **do frekvenční domény**, **vynásobí se a převedou se zpět** pomocí IFFT (inverzní FFT). Běžná konvoluce je v **O(n^2)**, konvoluce pomocí FFT v **O(n*logn)**.

Pomůcka

platí že **periodicitu v časové** doméně způsobuje **diskrétnost ve frekvenční** doméně a **diskrétnost v časové** doméně způsobuje **periodicitu ve frekvenční** doméně.

FŘ je **periodická spojitá** (v čase) → její koeficienty jsou **diskrétní neperiodické** (ve frekvenci),

DFR je **periodická diskrétní** (v čase) → její koeficienty jsou **diskrétní periodické** (ve frekvenci), perioda u bou je **stejná**.

FT není periodická spojitá (v čase) → koeficienty jsou vyjádřeny spojitou neperiodickou funkcí (ve frekvenci),

DTFT není periodická diskrétní (v čase) → koeficienty jsou vyjádřeny spojitou periodickou funkcí (ve frekvenci)

DTF je pseudo periodická diskrétní (v čase) → koeficienty jsou diskrétní a periodické (ve frekvenci), ale vždy je jich stejný počet jako v čase (takže délka jedné periody - původního signálu).

Signály **pomalu** měnící se v čase budou mít **úzké** spektrum ve frekvenci, signály **rychle** měnící se v čase budou mít **široké** spektrum ve frekvenci

Užitečné signály

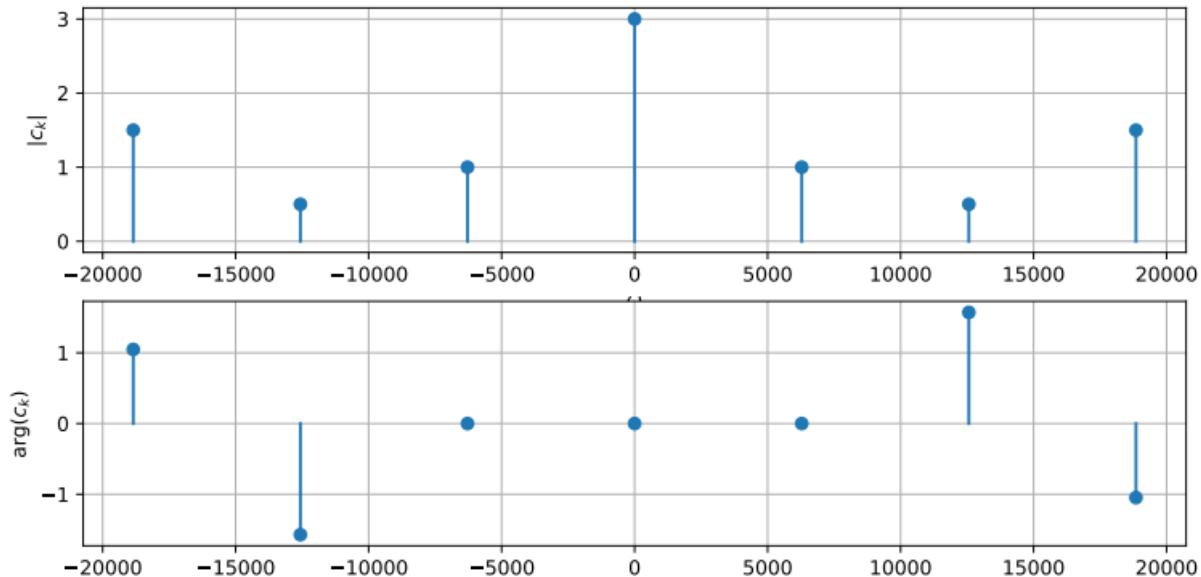
- několik cosinusovek:

$$x(t) = 3 + 2 \cos(2000\pi t) + 1 \cos(4000\pi t + \frac{\pi}{2}) + 3 \cos(6000\pi t - \frac{\pi}{3})$$

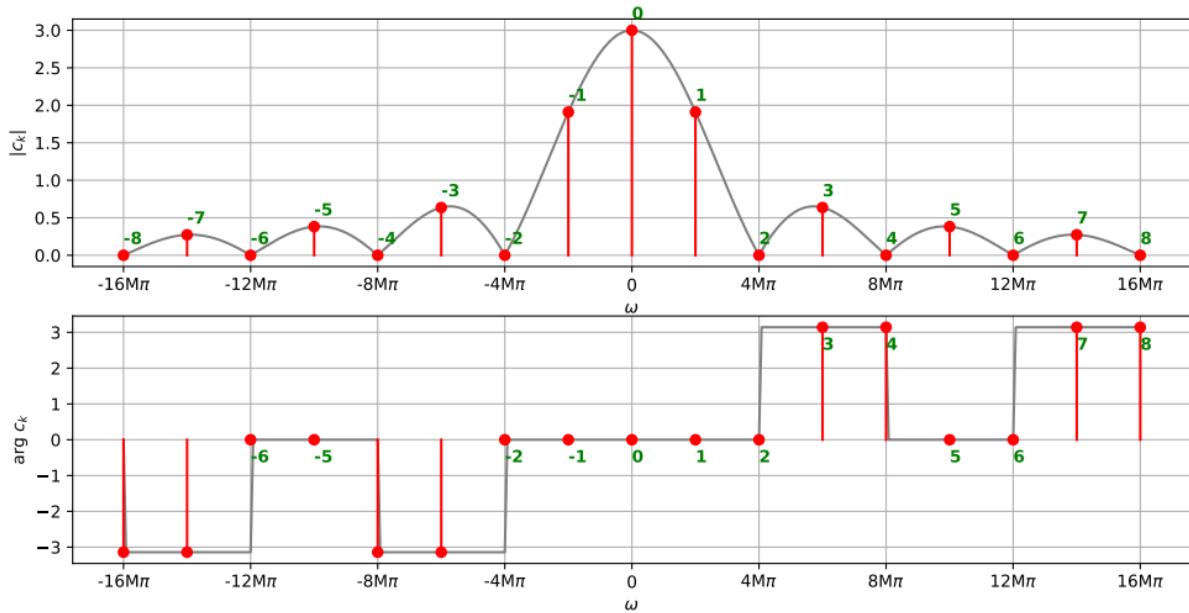
koeficienty získáme z převodního vzorečku (**půlka amplitudy a stejná, resp. opačná fáze**)

$$c_k = \frac{C_k}{2} e^{j\phi_k}, \quad c_{-k} = \frac{C_k}{2} e^{-j\phi_k}.$$

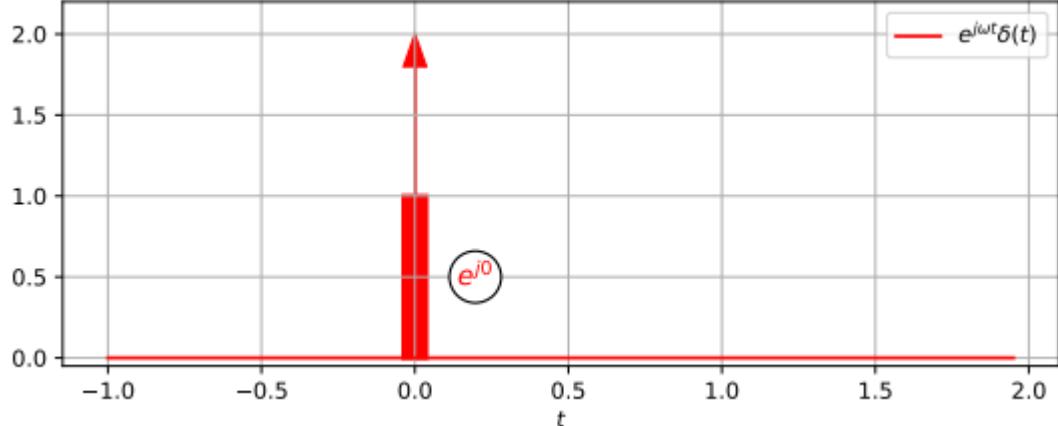
$$c_0 = 3, \quad c_1 = 1, \quad c_{-1} = 1, \quad c_2 = \frac{1}{2} e^{j\frac{\pi}{2}}, \quad c_{-2} = \frac{1}{2} e^{-j\frac{\pi}{2}}, \quad c_3 = 1.5 e^{-j\frac{\pi}{3}}, \quad c_{-3} = 1.5 e^{j\frac{\pi}{3}}.$$

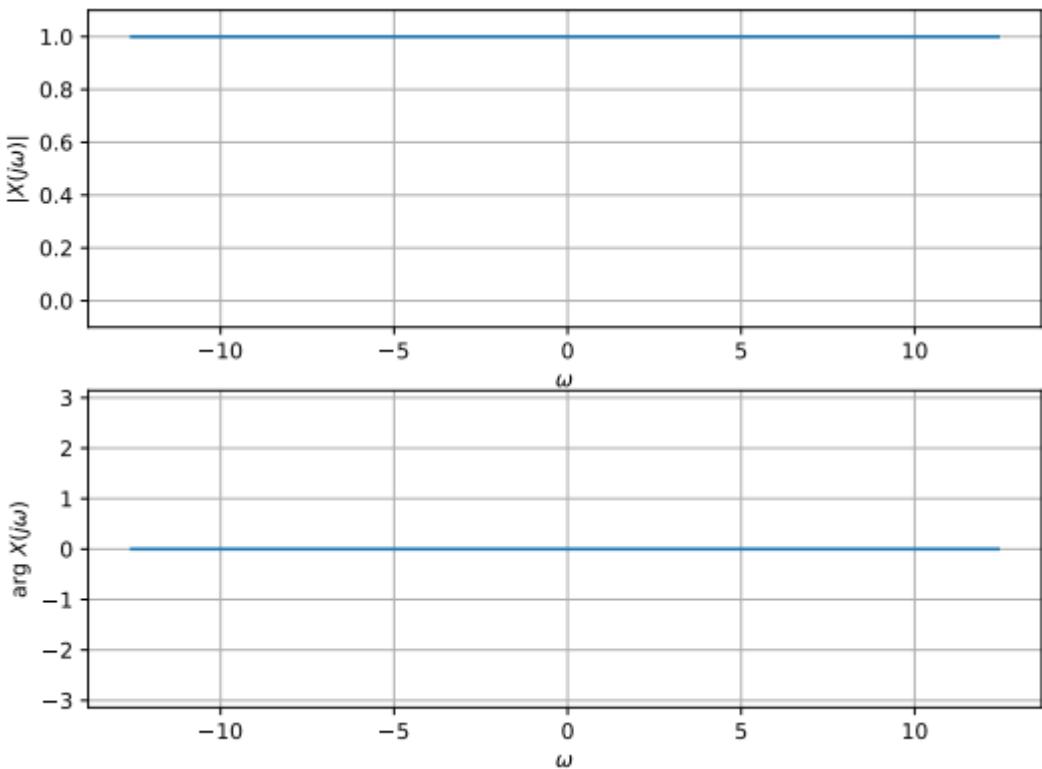


- **obdelník:** Velikost impulsů bude $D = 6$, základní perioda $T_1 = 1 \mu\text{s}$, šířka impulsu $\vartheta = 0.5 \mu\text{s}$. $\omega = 2\pi/\vartheta = 2\pi/(0.5 \times 10^{-6}) = 4\pi \times 10^{-6} = 4M\pi$. $\omega = 4M\pi$.



- **Diracův impuls $\delta(t)$:** "za 0 sekund dokáže vyskočit do nekonečna a zase se vrátit zpět"





Frekvence

- Normální nenormovaná frekvence - f [Hz]
- Normální normovaná frekvence - f / F_s [-]
- Kruhová nenormovaná frekvence - $\omega = 2\pi f$ [rad/s]
- Kruhová normovaná frekvence - $(2\pi f)/F_s$ [rad]

videa:

[But what is the Fourier Transform? A visual introduction.](#)

[How are the Fourier Series, Fourier Transform, DTFT, DFT, FFT, LT and ZT Related?](#)

[The Discrete Fourier Transform \(DFT\)](#)

[Image Compression and the FFT](#)

[3 Applications of the \(Fast\) Fourier Transform \(ft. Michael Kapralov\)](#)

15. Číslicové filtry (diferenční rovnice, impulsní odezva, přenosová funkce, frekvenční charakteristika)

Diferenční rovnice

Diferenční rovnice představují **rovnice o neznámé posloupnosti a jejích diferencích**. Jedná se o **diferenciální rovnice**, které jsou v **diskrétním čase**. Obecná diferenční rovnice vypadá následovně:

$$y[n] = \sum_{k=0}^Q b_k x[n-k] - \sum_{k=1}^P a_k y[n-k], \quad (1)$$

kde $x[n-k]$ jsou aktuální a zpožděně verze vstupu a $y[n-k]$ jsou zpožděné verze výstupu.

diferenční rovnice stejně jako diferenciální vyžadují počáteční podmínky. Pomocí diferenčních rovnic můžeme popisovat LTI (Linear time-invariant) systémy.

- **linearita:** musí platit aditivita a scaling nebo homogenita:

$$ax_1[n] + bx_2[n] \rightarrow ay_1[n] + by_2[n]$$

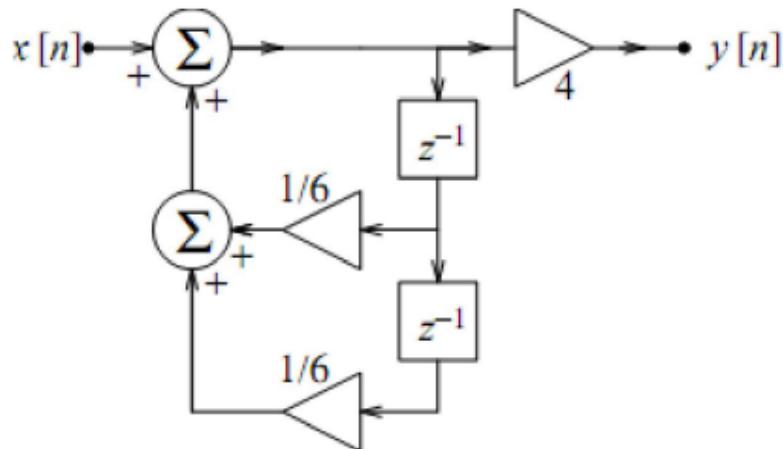
- **časová invariantnost:** systém **nemění** své **chování v čase**. Pokud systém zareagoval na signál x výstupem y , zareaguje na stejný signál x za deset sekund stejným signálem y .

Příklad popisu systému:

- aktuální výstup $y[n]$ je dán součtem minulého vstupu $1/2y[n-1]$, aktuálního vstupu $1/4x[n]$ a minulého vstupu $1/4x[n-1]$.

$$y[n] = \frac{1}{2}y[n-1] + \frac{1}{4}x[n] + \frac{1}{4}x[n-1]$$

- systém dán **diferenční** rovnicí $y[n] = 4x[n] + 1/6y[n-1] + 1/6y[n-2]$



Příklady systémů, které se chovají jako dolní propust (propouští malé frekvence).

- **průměr** posledních 6 vstupů:

$$y[n] = \frac{1}{6} \{ x[n] + x[n-1] + \dots + x[n-5] \}$$

- 'nabalující' se vstup:

$$y[n] = 0.95 y[n-1] + 0.05 x[n]$$

Příklady systémů, které se chovají jako horní propust (propouští velké frekvence).

- **rozdíl** posledních 6 vstupů:

$$y[n] = \frac{1}{6} \{ x[n] - x[n-1] + x[n-2] - \dots - x[n-5] \}$$

- 'nabalujející' se vstup:

$$y[n] = -0.95 y[n-1] + 0.05 x[n]$$

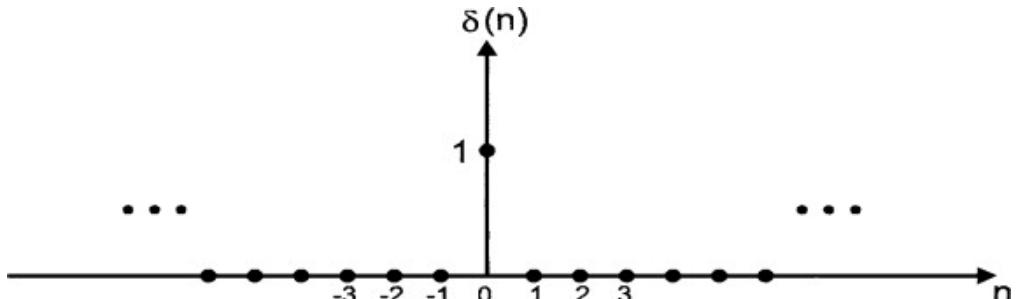
videa:

[Difference Equation Descriptions for Systems](#)

Impulsní odezva

Zkoumá **odezvu** systému na impuls. **Diskrétní impuls** je definován jako

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}$$



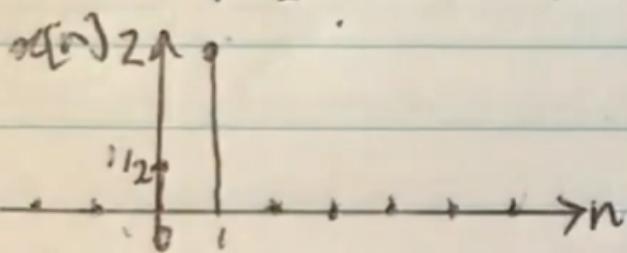
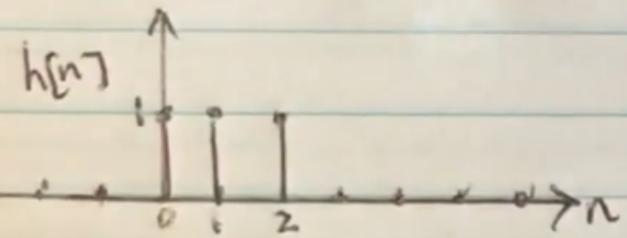
Jakýkoliv diskrétní signál může být reprezentován jako **vážený součet posunutých** diskrétních impulsů v čase. Impulsní odezva **definuje LTI** systémy a používá k tomu **konvoluci**. Impulsní odezva definuje systém, protože impuls obsahuje všechny frekvence (**ve frekvenční doméně je roven 1 pro všechny f**).

Konvoluce

Základní operace pracující se dvěma signály je definovaná vzorcem

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m], (f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]g[m].$$

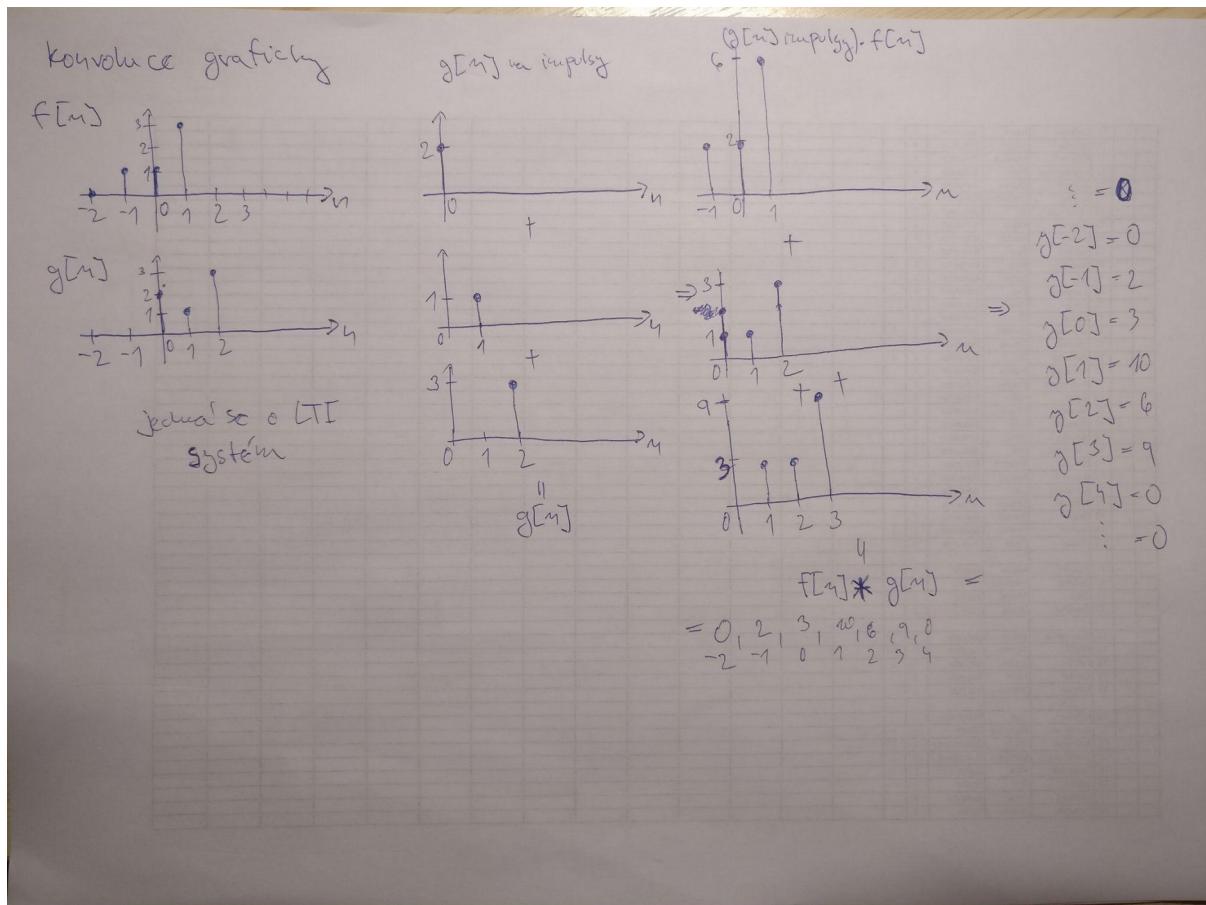
$$y[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k]$$



$$\begin{aligned} y[-1] &= x[0] h[-1] + x[1] h[-2] \\ &= 1 \cdot 0 + 2 \cdot 0 = 0 \end{aligned}$$

$$\begin{aligned} y[0] &= x[0] h[0] + x[1] h[-1] \\ &= 1 \cdot 1 + 2 \cdot 0 = 1 \end{aligned}$$

konvoluce graficky [Discrete Time Convolution Example](#):

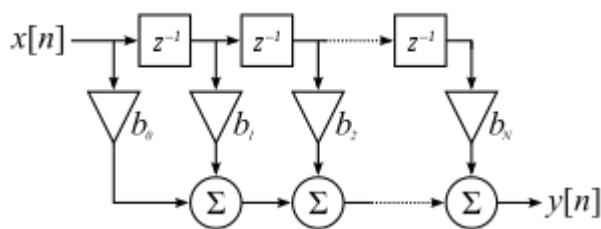


konvoluci lze ve **frekvenční doméně** provést pouze **násobením**.

FIR - Finite Impulse Response, IIR - Infinite Impulse Response

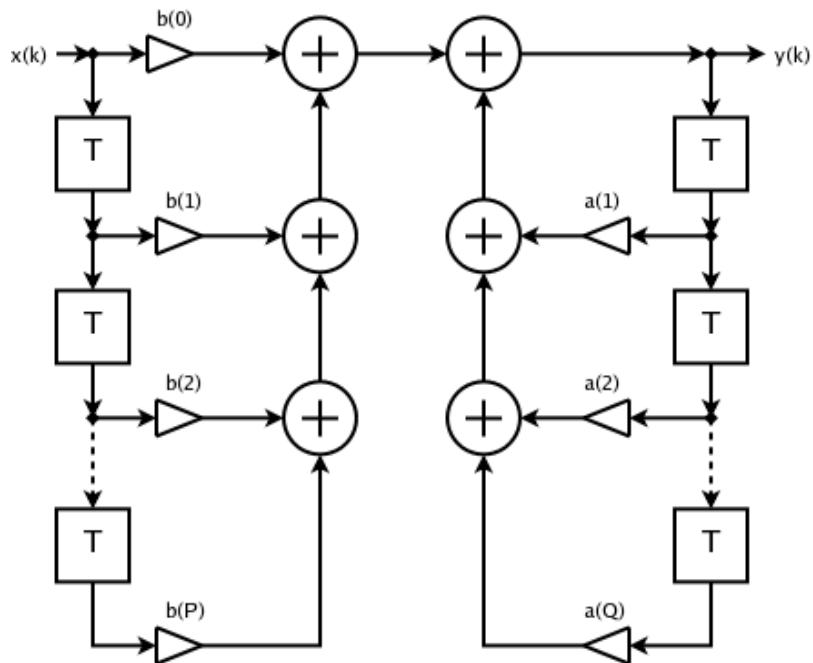
- **FIR:** impulsní odezva nebo odezva konečného vstupu má **konečnou délku**.
Tyto filtry **nesmí** mít ani jednu **zpětnou smyčku**.

$$y[n] = \sum_{k=0}^n b_k x[n-k]$$



- **IIR:** impulsní odezva a odezva konečného vstupu má **nekonečnou délku**.
Tyto filtry **musí** mít alespoň jednu **zpětnou smyčku**.

$$y[n] = \sum_{k=0}^m b_k x[n-k] - \sum_{k=s}^N a_k y[n-k]$$



Přenosová funkce

Přenosová funkce definuje LTI systém. Pokud chceme na nějaký signál aplikovat filter, tak jej tímto filtrem **vynásobíme** (konvoluce v časové doméně) ve frekvenční doméně (např. filtrem, který propouští pouze spodní frekvence). $X(z) * H(z) = Y(z)$ → Přenosová funkce je definována jako **poměr** (podíl) **výstupu** - $Y(z)$ ku **vstupu** - $X(z)$ ve **frekvenční oblasti**, když jsou **zanedbány počáteční podmínky**. Jinak řečeno při **nulových** počátečních podmínkách, jde o podíl **Fourierovy transformace** výstupu a vstupu, respektive **Z transformace** u signálů, pro které nelze provést **FT** (signály,

které **nejsou** v +- nekonečnu **0**, mají nekonečnou energii, Z transformace je násobí tak, aby **byly** v +- nekonečnu **0**). Přenosovou funkci lze také vyjádřit jako **FT impulsní odezvy** u FIR systémů (výstup je v nekonečnu 0), nebo jako **ZT** [Z Transform Region of Convergence Explained](#) IIR systémů (v nekonečnu nejsou 0). FT/ZT impulsní odezvy z toho důvodu, že $X(z)$ - vstup je **1** (FT jednotkového impulsu).

Z-Transform Example #1

Find $X(z)$ for $x[k] = \begin{cases} 1 & k=-1 \\ 3 & k=0 \\ 2 & k=1 \\ -1 & k=2 \\ 0 & \text{else} \end{cases}$

$$H(z) = \frac{Y(z)}{X(z)}$$

\Rightarrow Finite length signal.

The bilateral or two-sided Z-transform of a discrete-time signal $x[n]$

$$X(z) = \mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \quad (\text{Eq.1})$$

$$H(z) = \frac{\sum_{i=0}^P b_i z^{-i}}{1 + \sum_{j=1}^Q a_j z^{-j}}$$

$$\frac{z^{(-1)} + (3)z^0 + (2)z^1 + (-1)z^2}{z^3 + 2z^1 - z^2}$$

Considering that in most IIR filter designs coefficient a_0 is 1

Find DTFT of $x[k]$. $\Rightarrow X(\Omega) = e^{j\Omega} + 3 + 2e^{-j\Omega} - e^{-2j\Omega}$

Region of Convergence

[Z Transform Region of Convergence Explained](#) [Laplace Transform Region of Convergence Explained](#)

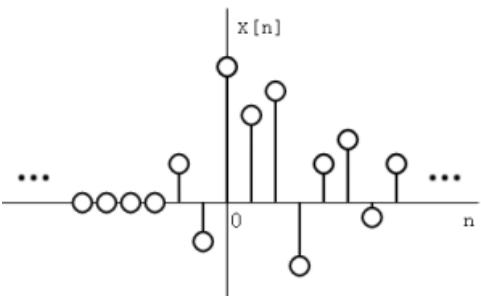


Figure 12.6.2: A right-sided sequence.

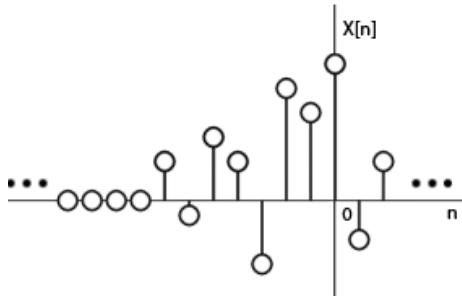


Figure 12.6.4: A left-sided sequence.

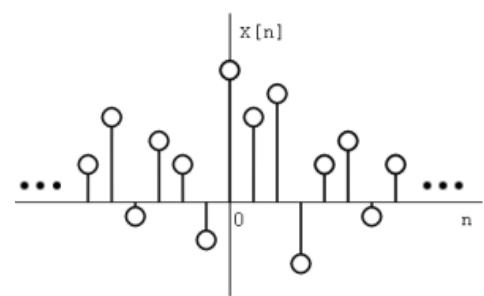


Figure 12.6.6: A two-sided sequence.

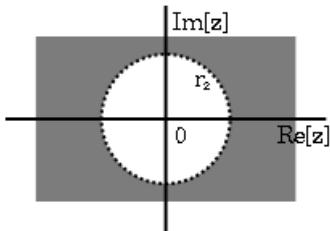


Figure 12.6.3: The ROC of a right-sided sequence.

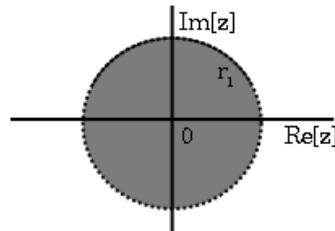


Figure 12.6.5: The ROC of a left-sided sequence.

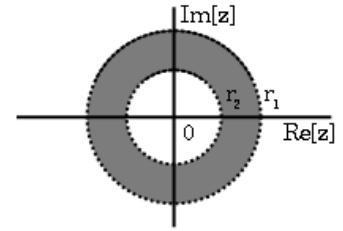
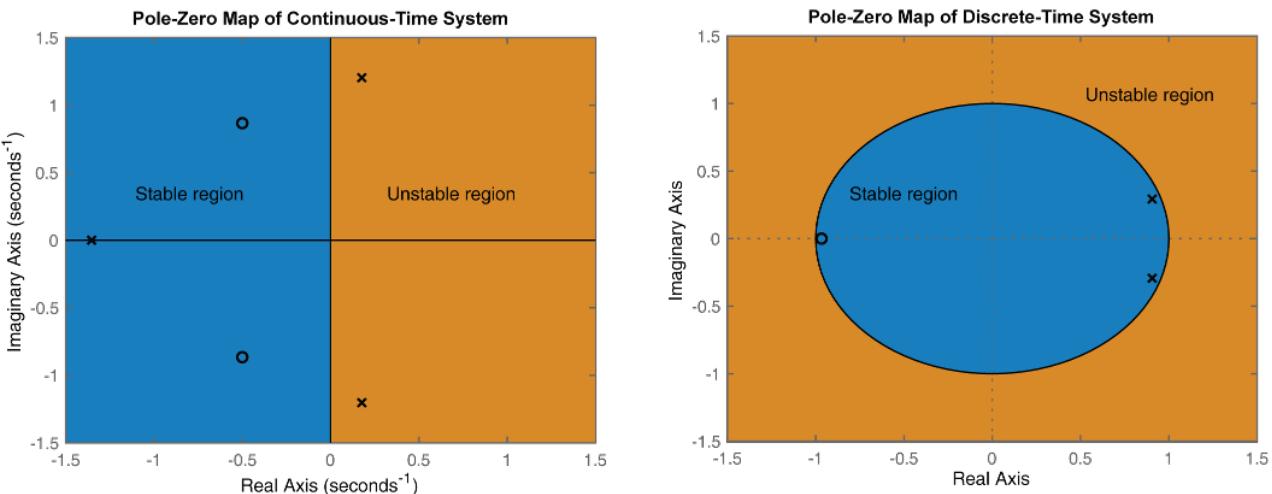


Figure 12.6.7: The ROC of a two-sided sequence.

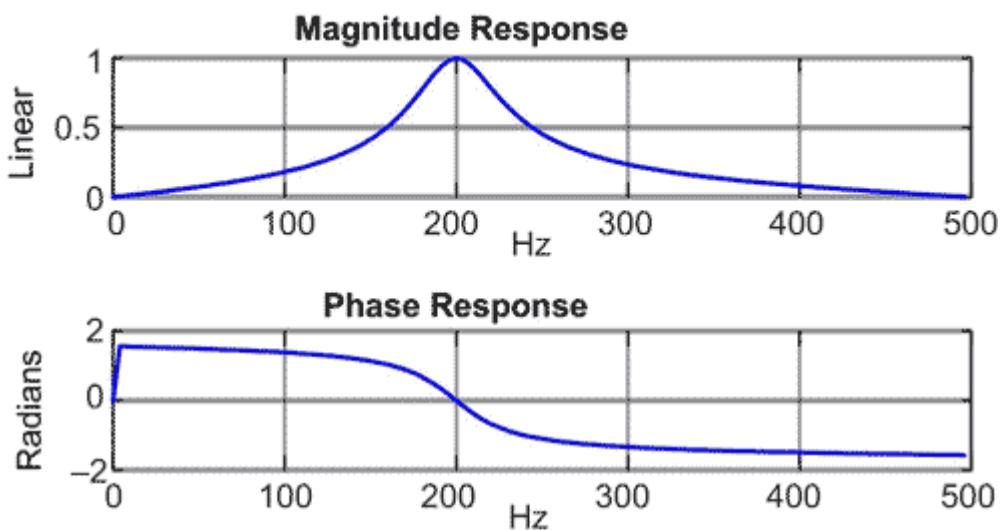
Stabilità

[How do Poles and Zeros affect the Laplace Transform and the Fourier Transform?](#)
[Frequency Response Magnitude and Poles and Zeros](#)



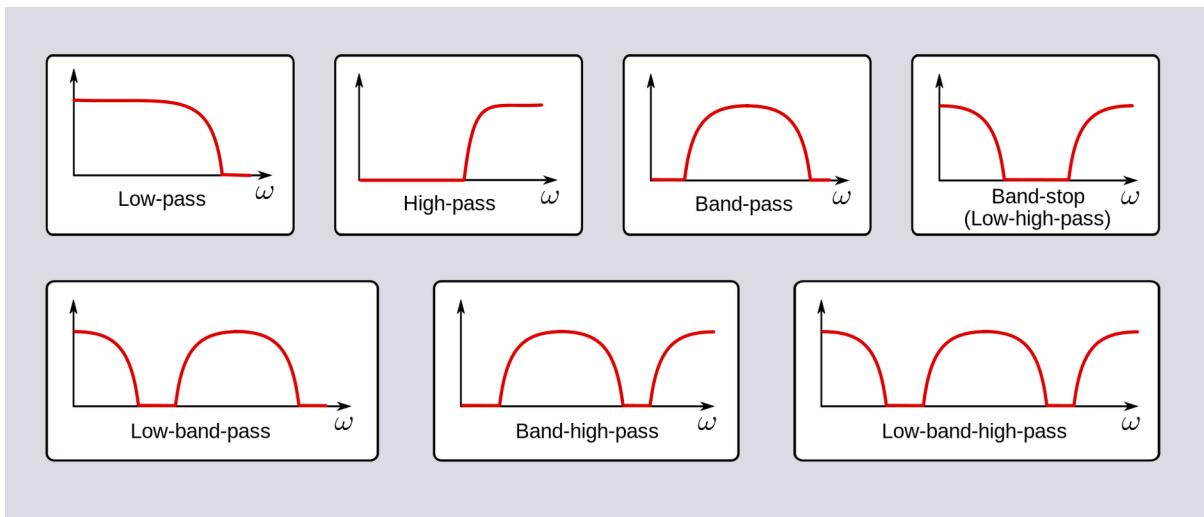
Frekvenční charakteristika

Frekvenční charakteristika popisuje systém **ve frekvenční doméně**, stejně jako **impulsní odezva** jej charakterizuje v **časové doméně**. Graficky ji vynášíme ve dvou grafech jako **amplitudovou $|H(\omega)|$** (jak zeslabuje/zesiluje) a **fázovou charakteristiku $\arg(H(\omega))$** (jak posouvá).



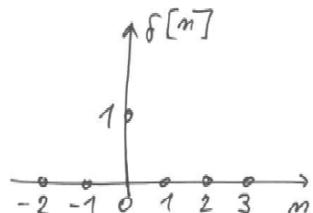
Dělení frekvenčních charakteristik

- **Dolní propust** - filtr, který propouští nízké frekvence.
- **Horní propust** - filtr, který propouští vysoké frekvence.
- **Pásmová propust** - filtr, který propouští signál jen určitých frekvencí.
- **Pásmová zádrž** - filtr, který nepropouští signál určitých frekvencí.

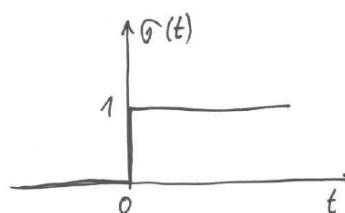


Komplexní čísla

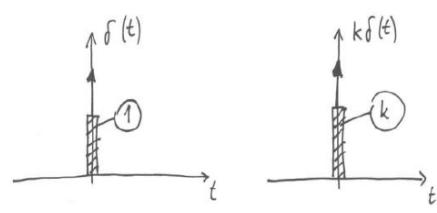
Jednotkový puls



Jednotkový skok



Diracův impuls



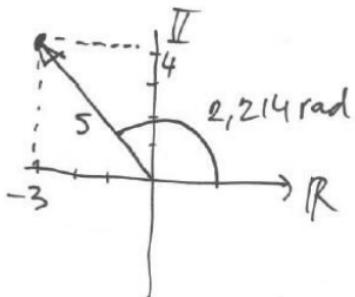
- Možné transformace signálu
 - Otočení časové osy - $s(-t)$
 - Zpozdění - $s(t-x)$, kde $x > 0$
 - Předběhnutí - $s(t+x)$, kde $x > 0$
 - Kontrakce - $s(m*t)$, kde $m > 1$
 - Dilatace - $s(t/m)$, kde $m > 1$
- Komplexní číslo - Složené z reálné části (osa x) a imaginární části (osa y), kde jednotka je odmocnina z -1, za kterou se substituuje i nebo j . Komplexní číslo je možné zapsat jako sumu těchto 2 hodnot $(42 + 3i)$ nebo vektorem: $z = |z|(\cos\phi + i\sin\phi)$, kde $|z|$ je délka vektoru a ϕ je úhel s kladnou osou x.

$$|z| = \sqrt{(a^2 + b^2)}; \varphi = \tan^{-1} \frac{b}{a}$$

Příklad 1

$$z = -3 + j4, \quad r = \sqrt{3^2 + 4^2} = 5, \quad \phi = \tan^{-1} \frac{4}{-3} = -0.92$$

což je špatně!. Správný úhel je $\phi = \pi - 0.92 = 2.214$ rad.



- **Exponenciální tvar komplexního čísla** - $z = r^* e^{j\phi}$
- **Komplexně sdružené číslo** - pro číslo $z = a + bi = re^{j\phi}$ existuje komplexně sdružené číslo $\bar{z} = a - bi = re^{-j\phi}$. Vznikne změnou znaménka imaginární části čísla. Např. $\overline{3 - 2i} = (3 - 2i)^* = 3 + 2i$. Součtem čísla a jeho komplexně sdruženého čísla vznikne reálné číslo. Funkce $y = e^{jx}$ je komplexní exponenciála součtem s komplexně sdruženou získáme kosinusovku.

$$\cos \phi = \frac{e^{j\phi} + e^{-j\phi}}{2} \quad \sin \phi = \frac{e^{j\phi} - e^{-j\phi}}{2j}$$

- **Obecná kosinusovka**

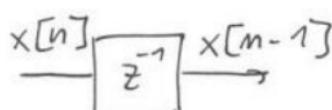
$$C_1 \cos(\omega_1 t) = \frac{C_1}{2} e^{j\omega_1 t} + \frac{C_1}{2} e^{-j\omega_1 t}$$

Filtr

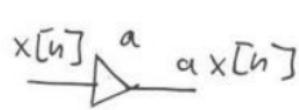
Filtr je lineárně časově invariantní (LTI) systém, který upravuje signál **ve frekvenční oblasti**. Je charakteristický **impulsní odezvou**. Výstup se získá konvolucí. Lze implementovat pomocí diferenční rovnice.

- **Základní bloky filtrů**

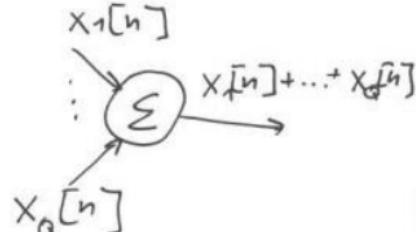
Zpožďovací článek



násobička



sčítáčka



15. Číslicové filtry (diferenční rovnice, impulsní odezva, přenosová funkce, frekvenční charakteristika) Ta otazka to celkem presne rika. Chci vedet:

jake jsou zakladni bloky cislicoveho filtru: zpozdeni, nasobeni, soucet.

- jak vypada schema obecneho IIR filtru s nerekurzivni (koeficienty b) a rekurzivni casti (koeficienty a).
- co z nej ustrihnut, aby se z nej stal FIR nebo ciste IIR filtr.
- jak schema zapsat diferencni rovnici.
- jak ji prevest na obrazovou formu (tedy jak ji z-transformovat): pomucka: konstanty zustanou konstanty, vsechna $x[n]$ se prepisou na $X(z)$, $y[n]$ na $Y(z)$ a kdyz je nekde zpozdeni o neco, musi se vyjadrit pomocí $z^{\{-neco\}}$
- jak z toho udelat prenosovou funkci (reseni: chcete dostat podil $Y(z)/X(z)$, vyjde Vam podil polynomu $B(z)/A(z)$, kde citatel zavisi na vstupni casti, jmenovatel na vystupni).
- jak z toho udelat kmitoctovou charakteristiku: nahradit z za $e^{\{j \omega\}}$, kde omega je norm. kruh. frekvence... mimochedom (viz minula otazka), ted uz by Vam melo byt jasne, proc je frekv. charakteristika cislic. filtru periodicka s F_s
- velice zhruba (bez rovnic) vedet, jak se frekv. charakteristika da spocitat nebo odhadnout rozlozenim citatele i jmenovatele na nuly a poly.

upozorneni: toto si pod otazkami predstavuju ji jako garant a ucitel ISS, u statnic ale muzete potkat dalsi lidi, kteři temto vecem dobre rozumi (celkem kdokoliv z recove a graficke skupiny, Fucik, Sekanina, a mnozi dalsi) a ti mohou mit lehce odlišnou interpretaci. Naucenim vysle uvedeneho ale rozhodne neprohlopitez.

16. Množiny, relace a zobrazení.

Množina

Matematická struktura **neopakujících** se objektů (prvků množiny), chápáných jako celek. Množina je jednoznačně určena svými prvky, ale nezáleží na jejich pořadí. Množiny mohou být prázdné, konečné a nekonečné.

Nekonečné množiny

- **Spočetná** - Taková množina, která je vzájemně **jednoznačná** (bijektivní zobrazení) s některou **podmnožinou přirozených čísel**.
- **Nespočetná** - Množina, kterou nelze jednoznačně vzájemně zobrazit na žádnou podmnožinu přirozených čísel (např. **reálná čísla**).

Popis množiny a značení

Množiny obvykle značíme velkými písmeny a jejich prvky písmeny malými. Je-li prvek **a** a prvkem množiny **B**, píšeme: $a \in B$. Způsoby zadávání:

- **Výčtem prvků**: $A = \{42; 1337; 94; 0\}$,
- **Predikátem**: $B = \{2^k \mid k \in \mathbb{N}\}$,
- **Intervalem**: $(a; b)$, $(a; b>$, $<a; b)$, $<a; b>$ - $a < b$,
- **Dobře známé množiny**: $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \emptyset$.

Potenční množina

Potenční množina množiny **X P(X)** je taková množina, která obsahuje všechny podmnožiny množiny **X**. Pokud je množina **X** konečná a její mohutnost $|X| = n$, pak je mohutnost potenční množiny **P(X)** rovna $|P(X)| = 2^n$. Např.:

$$\begin{aligned} A &= \{1, 2, 3\} \\ P(A) &= \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \\ &\text{(pozn. množina samotná je také podmnožinou sama sebe).} \end{aligned}$$

Prázdná množina (\emptyset nebo $\{\}$)

Množina, která neobsahuje žádné prvky.

Uzavřenosť množiny na operaci

Podmnožina $M \subset A$ je **uzavřená** vůči algebraické operaci pokud tato operace **vrátí** hodnotu z M . Tedy Provedením operace s **uzavřenou množinou k této operaci** získáme opět prvek **z této množiny**.

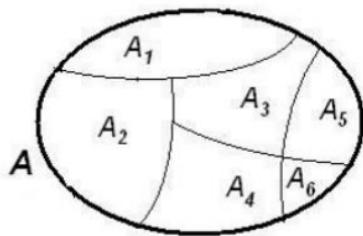
Podmnožina

Množina **B** je podmnožinou množiny **A**, pokud platí, že všechny prvky množiny B se **nachází** v množině **A**. Pak je možné množinu **A** označit také za **nadmnožinu**. Stav

“bytí podmnožinou” se také nazývá **inkluze**.

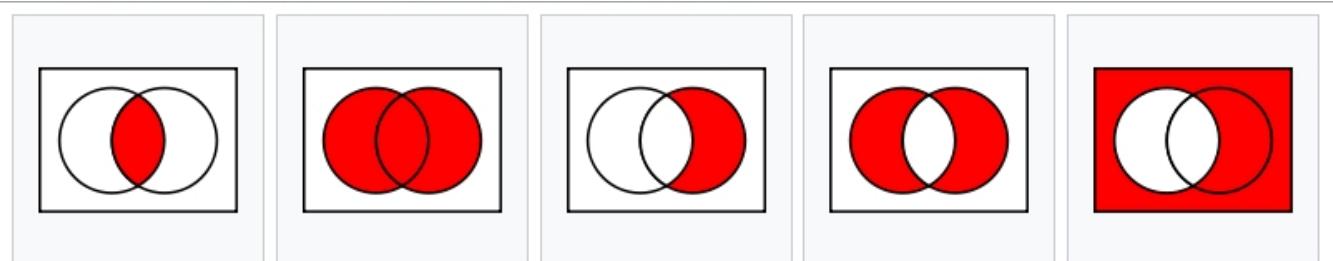
Rozklad množiny

Rozkladem množin vznikne **systém množin** (množina množin), kde sjednocením jejích prvků získáme původní rozloženou množinu a zároveň pokud pro libovolné dva prvky rozkladu platí $X \cap Y \neq \emptyset$, pak $X = Y$.



Operace s množinami

- **Průnik:** průnik dvou nebo více množin označuje taková množina, která obsahuje pouze ty prvky, které se nalézají ve všech těchto množinách.
- **Sjednocení:** sjednocení dvou nebo více množin označuje taková množina, která obsahuje každý prvek, který se nachází alespoň v jedné ze sjednocovaných množin, a žádné další prvky.
- **Rozdíl:** rozdíl dvou množin označuje taková množina, která obsahuje každý prvek, který se nachází v první z množin, ale nenachází se ve druhé z nich, a žádné další prvky.
- **Symetrická differenze** (symetrický rozdíl): symetrická differenze nebo symetrický rozdíl dvou množin označuje taková množina, která obsahuje všechny prvky z obou množin, které nejsou v jejich průniku. Značení:
 - $A \Delta B, A \setminus B, A \ominus B$.
- **Doplňek:** doplněk množiny A nebo **komplement** množiny A označuje množina A^c všech prvků, které nejsou v A a přitom v nějaké jiné (předem dané) množině jsou obsaženy (na obrázku v U). Aby bylo možné doplněk definovat, je třeba znát množinu, vzhledem ke které se doplněk počítá.



Průnik dvou množin
 $A \cap B$

Sjednocení dvou množin
 $A \cup B$

Rozdíl množin A (vlevo)
a B (vpravo)
 $A^c \cap B = B \setminus A$

Symetrická differenze dvou množin
 $A \Delta B$

Doplňek A v U
 $A^c = U \setminus A$

Vlastnosti operací

- **Komutativnost** - Vlastnost binární operace, která říká, že nezáleží na pořadí argumentů - $A \cap B = B \cap A$. U komutativní operace můžeme **zaměnit pořadí** hodnot a **nezměníme** tím **výsledek**.
- **Asociativita** - Vlastnost **binární operace**, která říká, že **nezáleží** na tom, jak použijeme **závorky** u výrazu, kde je více operandů. Nezáleží, v jakém pořadí budeme tedy tento výraz počítat - $(A \cap B) \cap C = A \cap (B \cap C)$.
- **Distributivita** - Vlastnost **binární operace vůči jiné binární operaci**, říkající, že můžeme tuto **operaci distribuovat** přes jinou operaci - $x*(y + z) = (x * y) + (x * z)$. Nebo $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$.

Relace

Jako relaci nebo n-ární relaci nazveme v matematice libovolný vztah mezi skupinou prvků jedné nebo více množin. Podle **arity** dělíme relace na **unární, binární, ternární a nární**.

Relace mezi množinami A_1, A_2, \dots, A_k , pro $k \in \mathbf{N}$, je libovolná podmnožina kartézského součinu $R \subseteq A_1 \times A_2 \times \dots \times A_k$. Pokud $A_1 = A_2 = \dots = A_k = A$, hovoříme o **k-ární** relaci **R** na **A**.

Unární relací

V relaci se nachází jeden prvek, např.:

- relace: **Být kladné číslo**. Ptáme se, je číslo kladné?
- relace: **Být pravdivý výrok**. Ptáme se, je výrok pravdivý?

Binární relace

Binární relace jsou relace, do kterých vstupují **dva** prvky množiny (množiny mohou být různé). Binární relace **R** mezi množinami **A, B** je libovolná podmnožina **R** **kartézského součinu** množin **A, B**. Označení: $[x, y] \in R$ nebo (xRy) . Pokud $A = B$, hovoříme o binární relaci na množině **A**. Množina **A** se nazývá **definičním oborem** relace **R** a množina **B oborem hodnot** relace **R**.

- **Symetrická**: pokud platí (pRq) , tak platí i (qRp) . Relace je symetrická pokud pro **p** a **q** z **X** platí, že **p** je v relaci s **q** a **q** je současně v relaci s **p**.
 - např. relace **R** = 'je sourozenec' - **Prázdná relace je symetrická**, tzn. platí to i když někdo nemá sourozence.

$$\forall a, b \in X : [a, b] \in R \implies [b, a] \in R$$

- **Antisimetrická** (slabě antisimetrická): Relace, ve které nenastává že by **p** bylo v relaci s **q** a zároveň **q** v relaci s **p**. Pokud tak nastane, **p** se rovná **q**. Jedná se např. o **R** = 'je menší nebo rovno' ((neostré) uspořádání). **Nejedná se o opak symetrie**. Relace může být **současně symetrická i asymetrická**, např. rovnost.

$$(\forall a, b \in X)(aRb \wedge bRa \Rightarrow a = b)$$

- **Asymetrická** (Silně antisymetrická): relace, která je **současně antisymetrická a ireflexivní**. Jedná se např. o $R = \text{'je menší než'}$ (ostrá nerovnost). Jediná relace, která je **současně symetrická a asymetrická**, je **prázdná** relace.

$$(\forall a, b \in \mathbf{X})(aRb \Rightarrow \neg(bRa))$$

- **Reflexivní**: pokud pro všechna x patřící do X platí **(xRx)**, jinak řečeno, pokud je každý prvek **v relaci sam se sebou**. **Prázdná** relace nad **prázdnou** množinou **je reflexivní**, ale **prázdná** relace nad **naprázdnou** relací **není reflexivní**.
 - např. $R = \text{'je stejný'}$, $R = \text{'je větší nebo rovno'}$, $R = \text{'je podmnožinou'}$

$$\forall a \in X : [a, a] \in R$$

- **Tranzitivní** - Pokud **(pRq)** a současně **(qRr)**, pak platí **(pRr)**. Pokud pro každé p, q, r z množiny X platí, že pokud je p v relaci s q a q v relaci s r , tak je i p v relaci s r . **Prázdná relace je tranzitivní**.
 - např. $R = \text{'je sourozenec'}$, $R = \text{'je vyšší'}$
 - relace $R = \text{'je kamarád'}$ **není tranzitivní**.

$$\forall a, b, c \in X : (([a, b] \in R \wedge [b, c] \in R) \implies [a, c] \in R)$$

Známé binární relace

- **Ekvivalence**: relace, která je současně **reflexivní**, **symetrická** a **tranzitivní**. Např. **rovnoběžnost**, **rovnost**, **podobnost trojúhelníků**.
- **Uspořádání**: relace, která je současně **reflexivní**, (slabě) **antisymetrická** a **tranzitivní**. Např. $R = \text{'větší nebo rovno než'}$ nebo $R = \text{'je podmnožinou'}$
- Ostré uspořádání: relace, která je současně **ireflexivní**, **asymetrická** a **tranzitivní**. Např. $R = \text{'větší než'}$ nebo $R = \text{'je vlastní podmnožinou'}$ (podmnožina množiny, která ji není rovna).

Inverzní relace

Inverzní binární relace je množina uspořádaných párů, která je přesnou inverzí (obrácením pořadí) množiny uspořádaných párů původní binární relace. Např.: pro $R = \{(1, a), (2, b), (3, c)\}$ je inverzní relace $R^{-1} = \{(a, 1), (b, 2), (c, 3)\}$

Uzávěry relací

- **Tranzitivní**: [Warshall's Algorithm \(Finding the Transitive Closure\)](#) Např. pro $\underline{R = \{(2, 1), (2, 3), (3, 1), (3, 4), (4, 1), (4, 3)\}}$ on set $A = \{1, 2, 3, 4\}$ je tranzitivní uzávěr $R^+ = \{(2, 1), (2, 3), (2, 4), (3, 1), (3, 3), (3, 4), (4, 1), (4, 3), (4, 4)\}$

- **Reflexivní:** reflexivní uzávěr binární relace \mathbf{R} na množině \mathbf{X} je nejmenší reflexivní relace \mathbf{S} na množině \mathbf{X} obsahující relaci \mathbf{R} .

$$S = R \cup \{(x, x) : x \in X\}$$

Např.: $X = \{1, 2, 3\}$, $R = \{(1, 1), (2, 3)\}$, $S = \{(1, 1), (2, 2), (3, 3), (2, 3)\}$

- **Symetrický:** symetrický uzávěr binární relace \mathbf{R} na množině \mathbf{X} je nejmenší symetrická relace \mathbf{S} na množině \mathbf{X} obsahující realaci \mathbf{R} .

$$S = R \cup \{(y, x) : (x, y) \in R\}.$$

Např.: $X = \{1, 2, 3\}$, $R = \{(1, 1), (2, 1), (2, 3)\}$, $S = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$

Ternární relace

Ternární relace jsou relace, do kterých vstupují **tři** prvky množiny (množiny mohou být různé). Ternární relace \mathbf{R} mezi množinami $\mathbf{A}, \mathbf{B}, \mathbf{C}$ je libovolná podmnožina \mathbf{R} **kartézského součinu** množin $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Označení: $[x, y, z] \mathbf{R}$ nebo $\mathbf{R}(x, y, z)$. Pokud $\mathbf{A} = \mathbf{B} = \mathbf{C}$, hovoříme o ternární relaci na množině \mathbf{A} .

- např. $R = \text{'je mezi'}$

Algebra

Množina, na které jsou definované nějaké **operace** a daná množina je vzhledem k těmto operacím **uzavřená**, tzn. že výsledkem operace nad prvky této množiny je vždy také prvek této množiny. Např. sčítání na množině přirozených čísel.

Kongruence

Kongruence je ekvivalence (reflexivní, symetrická a tranzitivní relace) na algebře (množina uzavřená vůči operacím). Jedná se např. o množinu **zbytkových tříd**.

Kartézský součin

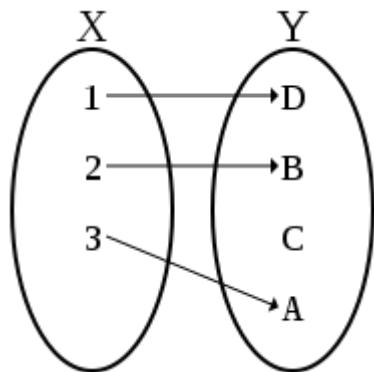
Kartézským součinem množin \mathbf{X} a \mathbf{Y} vznikne množina všech **uspořádaných dvojic**, ve kterých je první položka prvkem množiny \mathbf{X} a druhá prvkem množiny \mathbf{Y} .

$$X \times Y = \{(x, y) : x \in X \wedge y \in Y\}$$

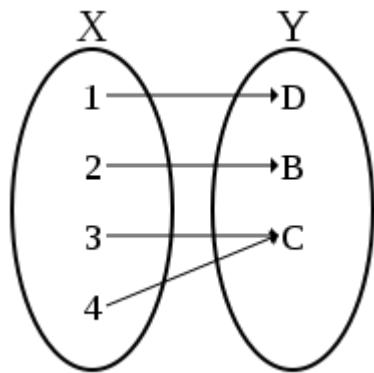
Zobrazení

Předpis, kterým se prvkům množiny \mathbf{X} přiřazuje **nejvýše jeden** prvek množiny \mathbf{Y} . Tedy zobrazení z množiny \mathbf{X} do množiny \mathbf{Y} . Prvky \mathbf{X} se nazývají **vzory** a prvky \mathbf{Y} se nazývají **obrazy**. Podobně jako u funkce musí každému prvku z \mathbf{X} být přiřazena nanejvýš jedna hodnota. **Zobrazení je speciálním případem binární relace, u které má každý vzor nejvýše jeden obraz.**

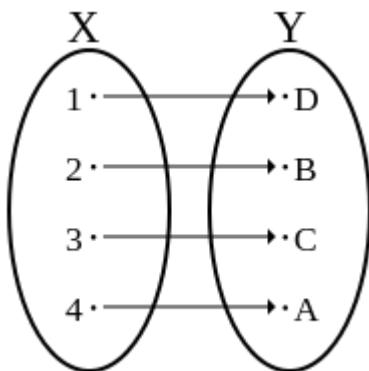
- **Injektivní (prosté) zobrazení:** Každý prvek z \mathbf{Y} má namapován nejvíce 1 prvek z \mathbf{X} .



- **Surjektivní zobrazení (zobrazení na):** Zobrazuje **na celou cílovou množinu** (**nezůstane volný prvek**, tedy každý prvek **Y** má namapovaný alespoň 1 prvek z **X**).



- **Bijektivní (vzájemně jednoznačné) zobrazení:** Zároveň **injektivní** a **surjektivní** zobrazení.



	surjective	non-surjective
injective	 bijective	 injective-only
non-injective	 surjective-only	 general

Inverzní zobrazení

Inverzní zobrazení k nějakému zobrazení $f: A \rightarrow B$ přiřazuje prvkům z množiny B prvky množiny A , tedy **obrazům** zobrazení f jejich **vzory**. Zobrazení f musí být **prosté** (injektivní).

Supremum a maximum

Nechť A je neprázdná **shora ohraničená** množina (množina, která nejde do nekonečna) reálných čísel. Číslo $\sup(A)$ se nazývá **supremum** množiny A , jestliže je **nejmenší horní závorou** množiny A . $\sup(A)$ musí být **větší nebo rovno** všem prvkům množiny A , nemusí být ale maximem. Např.: interval $X = (2, 3)$ má **supremum 3** ($\sup(X) = 3$), ale nemá maximum, protože se jedná o otevřený interval. $Y = (2, 3)$ má **supremum 3**, které je současně **maximem**.

Infimum a minimum

Nechť A je neprázdná **zdola ohraničená** množina (množina, která nezačíná v mínus nekonečnu) reálných čísel. Číslo $\inf(A)$ se nazývá **infimum** množiny A , jestliže je **největší dolní závorou** množiny A . $\inf(A)$ musí být **menší nebo rovno** všem prvkům množiny A , nemusí být ale minimem. Např.: interval $X = (2, 3)$ má **infimum 2** ($\inf(X) = 2$), ale nemá minimum, protože se jedná o otevřený interval. $Y = (2, 3)$ má **infimum 2**, které je současně **minimem**.

Svaz

Svaz je **uspořádaná množina**, která je doplněna o vlastnost, že pro **každé dva prvky** z daného svazu musí existovat **supremum a infimum**, které také naleží danému svazu.

Grupa

Grupou nazýváme množinu G spolu s binární operací (v tomto případě značenou $+$). Musí platit:

- Operace $+$ musí být na množině G **uzavřená**, tj. musí vracet prvek z množiny G .
- Operace $+$ musí být na množině G **asociativní**.
- **Existence neutrálního prvku**: Existuje prvek $e \in G$ (neutrální prvek) takový, že pro všechny ostatní prvky $a \in G$ platí: $a + e = e + a = a$.
- **Existence inverzních prvků**: Pro každý prvek $a \in G$ existuje prvek $b \in G$ takový, že $a + b = b + a = e$, kde $e \in G$ je **neutrální prvek**.

17. Diferenciální a integrální počet funkcí jedné a více proměnných.

Diferenciální počet

Matematická disciplína, která zkoumá **změny funkčních hodnot** v závislosti na změně nezávislé proměnné.

Spojitost funkce

Spojitost funkce je její důležitá vlastnost pro určování limit a derivací. Spojitá funkce je funkce, jejíž hodnoty se **mění plynule**, tedy při dostatečně malé změně hodnoty x se hodnota $f(x)$ změní libovolně málo. Intuitivní (ne zcela přesná) představa spojité funkce spočívá ve funkci, jejíž graf lze nakreslit jedním tahem, aniž by se tužka zvedla z papíru. Spojitost definujeme pomocí limity následovně:

- Funkce je v bodě x_0 definována (x_0 patří do definičního oboru).
- V bodě x_0 existuje limita funkce a je rovna právě funkční hodnotě v tomto bodě.

$$\lim_{x \rightarrow x_0} f(x) = f(x_0)$$

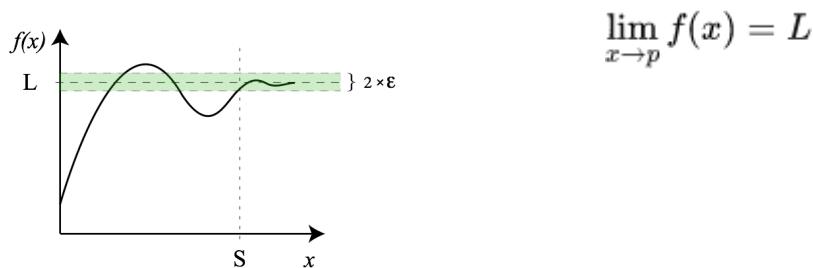
Pokud funkce není v bodě spojitá, může tato nespojitost být dvojího druhu:

- **prvního druhu:** Limity **zleva a zprava se nerovnají**, ale jsou **vlastní**.
- **druhého druhu:** Funkce má alespoň jednu **nevlastní** jednostrannou limitu nebo pokud alespoň jedna **limita neexistuje**.

Limita

Limita je matematická konstrukce vyjadřující, že se hodnoty zadané funkce blíží **libovolně blízko** k nějakému bodu. Právě tento bod je pak označován jako limita.

Funkce $f(x)$ má v bodě p limitu L , jestliž k **libovolně zvolenému okolí** bodu L **existuje okolí** bodu p tak, že pro všechna reálná $x \neq p$ tohoto okolí **náleží hodnoty** $f(x)$ zvolenému okolí **bodu L**. Značíme jako



Funkce má v bodě **maximálně jednu limitu**. Funkce $f(x)$ je spojitá v bodě p , právě když pro limitu v bodě p , platí $f(x) = f(p)$. Druhy limit:

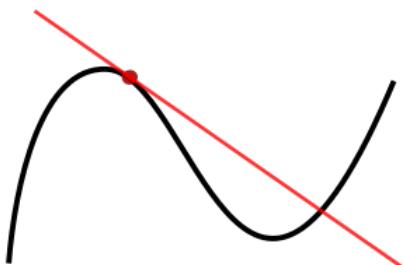
- **Vlastní** - Pokud hodnota neroste do +- nekonečna.
- **Nevlastní** - Limita roste do +- nekonečna.

- **Zprava** - Pokud se blížíme k bodu zprava na ose (z kladných čísel).
- **Zleva** - Pokud se blížíme k bodu zleva na ose (ze záporných čísel).

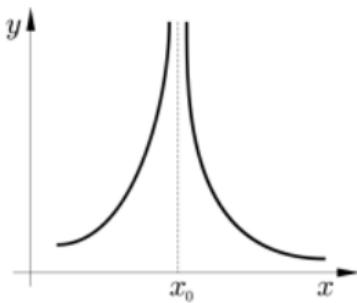
Derivace

Derivace funkce je **změna** (růst či pokles) její hodnoty **v poměru** ke **změně** jejího **argumentu**, pro velmi malé změny argumentu. V případě dvourozměrného grafu funkce $f(x)$ je **derivace této funkce** v libovolném bodě (pokud existuje) rovna **směrnici tečny tohoto grafu**. Například pokud funkce popisuje **dráhu tělesa v čase**, bude její **derivace** v určitém bodě udávat **okamžitou rychlosť**. Pokud popisuje **rychlosť**, bude **derivace udávat zrychlení**. Derivaci definujeme pomocí limity. Funkce $f(x)$ je v bodě x **diferencovatelná** (má derivaci), pokud v tomto bodě derivaci. Pokud limita v bodě x neexistuje nebo **je nevlastní**, pak **není derivace** funkce $f(x)$ v bodě x **definována**. Má-li funkce v daném bodě derivaci, pak je v tomto bodě i spojitá (naopak to **neplatí**).

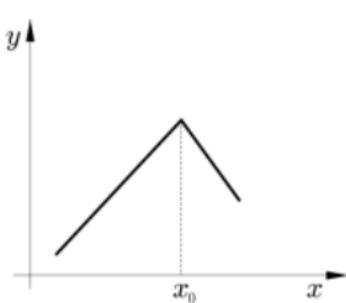
$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h} = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}$$



Derivaci používáme k **vyšetření průběhu funkce**. První derivací určujeme, jestli funkce v daném bodě roste, nebo klesá. V bodech, kde je **první derivace nulová**, nebo **neexistuje** hledáme **lokální extrémy** (minima a maxima). **Druhou derivací**



Derivace v x_0 neexistuje
a funkce je nespojitá.



Derivace v x_0 neexistuje
a funkce je spojitá.

Pravidla derivování

$$[f + g]' = f' + g'$$

$$[f - g]' = f' - g'$$

$$[fg]' = f'g + fg'$$

součinové pravidlo

$$\left[\frac{f}{g}\right]' = \frac{f'g - fg'}{g^2}$$

podílové pravidlo

$$[g(f)]' = g'(f) \cdot f'$$

řetízkové pravidlo

určujeme **konvexnost** (druhá derivace je kladná) a **konkávnost** (druhá derivace je záporná) funkcí. Body, ve kterých je druhá **derivace nulová nebo neexistuje**, jsou **inflexní body**. **Asymptoty grafu funkce** jsou přímky, ke které se graf funkce blíží v +/-nekonečnu

- **Bez směrnice:** podezřelé jsou hodnoty **vyloučené** z definičního oboru. Pro ověření je potřeba **spočítat limitu zleva a zprava** pro daný bod a pokud vyjdou **+nekonečno**, jedná se o **asymptotu bez směrnic**.
- **Se směrnicí:** Jedná se o asymptotu, která **není rovnoběžná s osou y**. Mohou existovat **maximálně 2** pro jednu funkci. ($y = kx + q$). Výpočet:

$$\lim_{x \rightarrow \pm\infty} [f(x) - (ax + b)] = 0$$

f	f'	$\mathcal{D}(f)$	$\mathcal{D}(f')$	Pozn.
const.	0	\mathbf{R}	• (tj. jako $\mathcal{D}(f)$)	
x^n	nx^{n-1}	\mathbf{R}	•	$n \in \mathbf{N}$
x^a	ax^{a-1}	$x > 0$	•	$a \in \mathbf{R}$
e^x	e^x	\mathbf{R}	•	
a^x	$a^x \ln a$	\mathbf{R}	•	$a > 0$
$\ln x$	$\frac{1}{x}$	$x > 0$	•	
$\log_a x$	$\frac{1}{x \ln a}$	$x > 0$	•	$a \in (0,1) \cup (1, +\infty)$
$\sin x$	$\cos x$	\mathbf{R}	•	
$\cos x$	$-\sin x$	\mathbf{R}	•	
$\operatorname{tg} x$	$\frac{1}{\cos^2 x}$	$x \neq \frac{\pi}{2} + k\pi$	•	
$\operatorname{cotg} x$	$-\frac{1}{\sin^2 x}$	$x \neq k\pi$	•	
$\arcsin x$	$\frac{1}{\sqrt{1-x^2}}$	$(-1,1)$	$(-1,1)$	v ± 1 : jen jednostranné derivace
$\arccos x$	$-\frac{1}{\sqrt{1-x^2}}$	$(-1,1)$	$(-1,1)$	v ± 1 : jen jednostranné derivace
$\operatorname{arctg} x$	$\frac{1}{1+x^2}$	\mathbf{R}	•	
$\operatorname{arccotg} x$	$-\frac{1}{1+x^2}$	\mathbf{R}	•	
$\sinh x$	$\cosh x$	\mathbf{R}	•	
$\cosh x$	$\sinh x$	\mathbf{R}	•	
$\operatorname{tgh} x$	$1 - \operatorname{tgh}^2 x$	\mathbf{R}	•	
$\operatorname{cotgh} x$	$1 - \operatorname{cotgh}^2 x$	$x \neq 0$	•	
$\operatorname{arg sinh} x$	$\frac{1}{\sqrt{x^2+1}}$	\mathbf{R}	•	
$\operatorname{arg cosh} x$	$\frac{1}{\sqrt{x^2-1}}$	$x > 1$	•	
$\operatorname{arg tgh} x$	$\frac{1}{1-x^2}$	$-1 < x < 1$	•	
$\operatorname{arg cotgh} x$	$\frac{1}{1-x^2}$	$ x > 1$	•	

Integrální počet

Integrální počet je část matematiky, která se zabývá především **integrací**. Integrály se využívají pro hledání **ploch**, **objemů** a délek **křivek**. Integraci můžeme také považovat **za proces sčítání**, dokonce tak lze definovat **určitý integrál**.

$$\int_a^b f(x) dx = \lim_{\delta x \rightarrow 0} \sum_{x=a}^b f(x) \delta x$$

Neurčitý integrál definujeme pomocí derivace takto: Nechť I je otevřený interval, $f(x)$ a $F(x)$ funkce na něm definované, pokud $F'(x) = f(x)$, pak se funkce $F(x)$ nazývá primitivní funkcí k funkci $f(x)$, nebo též **neurčitý integrál funkce $f(x)$ na intervalu I** .

$$\int f(x) dx = F(x) + c.$$

Neurčitý integrál

Operace integrování je **opačná** k operaci derivování. Integrál ale **není inverzní** funkci k derivaci, protože **derivováním ztrácíme** informaci o **konstantní** (stejnosměrné) složce derivované funkce. Abychom integrováním zderivované funkce získali funkci původní, musíme znát její hodnotu **alespoň v jednom bodě**. Pomocí této hodnoty můžeme **vypočítat konstantní (c)** složku.

Určitý integrál

Slouží pro výpočet **povrchu**, **objemu** nebo **obvodu** geometrického útvaru (výpočet je omezen na určitou část). Aby byla funkce integrovatelná nemusí být **spojitá** na celém intervalu nebo **spojitá po částech** (měla **konečně** mnoho bodů nespojitosti).

$$\int_a^b f(x)dx = [F(x)]_a^b = F(b) - F(a) \quad \int_a^b f(x)dx = - \int_b^a f(x)dx.$$

Postupy integrace

- **Metoda per partes:** využívá se pro integraci funkcí v součinovém tvaru dle vzorce (pro úspěšnou integraci velmi **záleží** na správném výběru **u** a **v**):

$$\int u \cdot v' = u \cdot v - \int u' \cdot v$$

$$\begin{bmatrix} u = x, & v' = \cos x \\ u' = 1, & v = \sin x \end{bmatrix}$$

Příklad integrace metodou per partes:

- **Substituční metoda:** Během integrování nahrazujeme část integrované

$$\int x \cdot \cos x \, dx$$

$$\int x \cdot \cos x = x \cdot \sin x - \int 1 \cdot \sin x$$

$$\begin{aligned}\int x \cdot \cos x &= x \cdot \sin x - \int 1 \cdot \sin x \\ &= x \cdot \sin x - (-\cos x) \\ &= x \cdot \sin x + \cos x + c\end{aligned}$$

funkce za jinou, provedeme integraci a vrátíme funkci zpět do původního tvaru. U **určitého** integrálu je **nutné přepočítat meze integrace**.

Příklad integrace substituční metodou:

$$\int \sin 6t \, dt \quad \int \sin 6t \, dt = \left[\frac{x}{dx} \int f(\phi(t)) \cdot \phi'(t) \, dt \right] = \int f(x) \cdot x' \, dx$$

Příklad integrace určitého integrálu:

$$\int_0^{\pi/2} e^{\sin(x)} \cos(x) \, dx = \left| \begin{array}{l} y = \sin(x) \\ dy = \cos(x) \, dx \\ x = 0 \mapsto y = \sin(0) = 0 \\ x = \pi/2 \mapsto y = \sin(\pi/2) = 1 \end{array} \right| = \int_0^1 e^y \, dy = [e^y]_0^1 = e - 1.$$

$$\frac{1}{6} \cdot \int \sin x \, dx = \frac{1}{6} \cdot (-\cos x)$$

f	$\int f \frac{dx}{x}$	Pozn.	Kde
x^n	$\frac{x^{n+1}}{n+1}$	$n \in \mathbf{Z}, n \neq -1$	$x \in \mathbf{R}$ pro $n \geq 0, x \in \mathbf{R} \setminus \{0\}$ pro $n < 0$
x^a	$\frac{x^{a+1}}{a+1}$	$a \in \mathbf{R} \setminus \mathbf{Z}$	$x \in (0, +\infty)$
$\frac{1}{x}$	$\ln x $		$x \in \mathbf{R} \setminus \{0\}$
e^x	e^x		$x \in \mathbf{R}$
a^x	$\frac{a^x}{\ln a}$	$a > 0, a \neq 1$	$x \in \mathbf{R}$
$\sin x$	$-\cos x$		$x \in \mathbf{R}$
$\cos x$	$\sin x$		$x \in \mathbf{R}$
$\operatorname{tg} x$	$-\ln \cos x $		$x \in \mathbf{R} \setminus \{\frac{\pi}{2} + k\pi, k \in \mathbf{Z}\}$
$\operatorname{cotg} x$	$\ln \sin x $		$x \in \mathbf{R} \setminus \{k\pi, k \in \mathbf{Z}\}$
$\frac{1}{\csc^2 x}$	$\operatorname{tg} x$		$x \in \mathbf{R} \setminus \{\frac{\pi}{2} + k\pi, k \in \mathbf{Z}\}$
$\frac{1}{\sin^2 x}$	$-\operatorname{cotg} x$		$x \in \mathbf{R} \setminus \{k\pi, k \in \mathbf{Z}\}$
$\arcsin x$	$x \arcsin x + \sqrt{1-x^2}$		$x \in (-1, 1)$
$\arccos x$	$x \arccos x - \sqrt{1-x^2}$		$x \in (-1, 1)$
$\operatorname{arctg} x$	$x \operatorname{arctg} x - \frac{1}{2} \ln(x^2 + 1)$		$x \in \mathbf{R}$
$\operatorname{arccotg} x$	$x \operatorname{arccotg} x + \frac{1}{2} \ln(x^2 + 1)$		$x \in \mathbf{R}$
$\frac{1}{\sqrt{1-x^2}}$	$\arcsin x$		$x \in (-1, 1)$
$\frac{1}{\sqrt{1-x^2}}$	$-\arccos x$		$x \in (-1, 1)$
$\frac{1}{1+x^2}$	$\operatorname{arctg} x$		$x \in \mathbf{R}$
$\frac{1}{1+x^2}$	$-\operatorname{arccotg} x$		$x \in \mathbf{R}$
$\sinh x$	$\cosh x$		$x \in \mathbf{R}$
$\cosh x$	$\sinh x$		$x \in \mathbf{R}$
$\operatorname{tgh} x$	$\ln(\cosh x)$		$x \in \mathbf{R}$
$\operatorname{cotgh} x$	$\ln \sinh x $		$x \in \mathbf{R} \setminus \{0\}$
$\frac{1}{\sqrt{x^2+1}}$	$\operatorname{argsinh} x$		$x \in \mathbf{R}$
$\frac{1}{\sqrt{x^2-1}}$	$\operatorname{sign} x \arg \cosh x $		$ x > 1$
$\frac{1}{1-x^2}$	$\arg \operatorname{tgh} x$	Pozor na def. obor!	$-1 < x < 1$
$\frac{1}{1-x^2}$	$\arg \operatorname{cotgh} x$	Pozor na def. obor!	$ x > 1$

Více proměnných

Parciální derivace

Parciální derivace funkce o **více proměnných** je její derivace vzhledem **k jedné z těchto proměnných**, přičemž s **ostatními** proměnnými se zachází jako s **konstantami**. **Většinou** platí:

$$F''_{yx} = F''_{xy}$$

Pomocí parciálních derivací se např. určuje tečná rovina grafu dvou proměnných

$$z = f(x_0, y_0) + f'_x(x_0, y_0)(x - x_0) + f'_y(x_0, y_0)(y - y_0)$$

Gradient (parciální derivace)

Jedná se o **vektor prvních parciálních derivací** dle **všech** proměnných funkce, který určuje směr **největšího růstu** funkce (respektive největšího poklesu, pokud jej vezmeme záporně). Délka vektoru gradientu je nárůst veličiny f na intervalu jednotkové délky.

$$\vec{\text{grad}} F = (F'_x; F'_y)$$

Umožňuje vypočítat **derivaci** funkce více proměnných ve **směru nějakého vektoru**, viz [19 - Gradient a jeho využití \(MAT - Diferenciální počet funkcí více proměnných\)](#). Směrová derivace se vypočítá jako skalární součin vektoru \mathbf{u} (vektor udávající směr) a gradientu v bodě \mathbf{A} (lze vyjádřit i obecně a poté dosadit libovolný bod).

$$s = \vec{\text{grad}} F_A \cdot \vec{u}_{\text{jed}}$$

Gradient je kolmý na křivky (u funkcí 2 proměnných) a plochy (u funkcí 3 proměnných) o **stejně funkční hodnotě - hladiny funkce**. Jedná se např. o **vrstevnice** na mapách.

V bodech, kde je gradient **nulový vektor**, se mohou nacházet **lokální extrémy** funkce více proměnných nebo **sedlové body**. Jedná se o stacionární body. Lokální **maximum** se v bodě nachází, pokud existuje i gradient druhé parciální derivace a jeho **hodnota je menší než 0**, respektive je zde lokální **minimum**, pokud je **hodnota větší než 0**.

$d^2f(A) < 0$, funkce f má v bodě A **ostré lokální maximum**,

$d^2f(A) > 0$, funkce f má v bodě A **ostré lokální minimum**.

$$d^2f(A) = \frac{\partial^2 f}{\partial x^2}(A)dx^2 + 2\frac{\partial^2 f}{\partial x \partial y}(A)dxdy + \frac{\partial^2 f}{\partial y^2}(A)dy^2.$$
$$d^2f(A) = (dx \ dy) \cdot \begin{pmatrix} \frac{\partial^2 f}{\partial x^2}(A) & \frac{\partial^2 f}{\partial x \partial y}(A) \\ \frac{\partial^2 f}{\partial x \partial y}(A) & \frac{\partial^2 f}{\partial y^2}(A) \end{pmatrix} \cdot \begin{pmatrix} dx \\ dy \end{pmatrix}$$

Dvojný integrál

Jedná se o integrál funkce **dvou proměnných**. Používá se např. pro výpočet **objemu**, výpočet **hmotnosti nehomogenní plochy**, výpočet **povrchu**. Při výpočtu se snažíme **dvojný** integrál převést na **integrál dvojnásobný**, tj. **dva jednoduché** integrály a integraci provádět postupně (Fubiniova věta). Hranicí vnitřního integrálu budou tvořit funkce.

$$\iint_I f(x, y) dx dy = \int_a^b \left(\int_{f(x)}^{g(x)} f(x, y) dy \right) dx \quad \iint_I xy dx dy = \int_0^2 \left(\int_{x^2}^{2x} xy dy \right) dx$$

Trojný integrál

Jedná se o integrál funkce **tří proměnných**. Používá se např. pro výpočet **objemu** tělesa ohrazeného třemi křivkami, výpočet jeho **hmotnosti**, na základě proměnlivé

hustoty, **statické momenty**, aj.

Výpočet **trojného integrálu** opět provádíme jeho **převodem na trojnásobný integrál**.

$$V = \iiint_M 1 \, dx dy dz \quad m = \iiint_M \rho_{x,y,z} \, dx dy dz$$

$$\iiint_B f(x, y, z) \, dx dy dz = \int_a^b \left(\int_{\varphi(x)}^{\psi(x)} \left(\int_{\Phi(x,y)}^{\Psi(x,y)} f(x, y, z) \, dz \right) dy \right) dx.$$

Příklady dvojného a trojného integrálu

- **dvojný integrál na čtverci:** $M = \langle 0, 3 \rangle \times \langle 0, 1 \rangle$

$$\begin{aligned} \int_M \frac{x^2}{3+y^2} \, dA &= \int_0^3 \int_0^1 \frac{x^2}{3+y^2} \, dy \, dx = \int_0^3 \left[\frac{x^2}{\sqrt{3}} \arctg \frac{y}{\sqrt{3}} \right]_{y=0}^{y=1} \, dx = \\ &= \frac{\pi\sqrt{3}}{18} \int_0^3 x^2 \, dx = \frac{\pi\sqrt{3}}{2}. \end{aligned}$$

- **dvojný integrál - Fubiniova věta:**

Řešení: M je ohraničená přímkou $y = -x$ a parabolou $y = x - x^2$ (Obr. 1).

Souřadnice průsečíků obou křivek získáme řešením soustavy dvou rovnic

$$\begin{aligned} y &= -x, \\ y &= x - x^2. \end{aligned}$$

Řešením této soustavy zjistíme, že křivky se protnou v bodech $(0, 0)$ a $(2, -2)$. Funkce $f(x, y) = xy$ je na M spojitá a je zřejmé, že pro libovolné $x \in \langle 0, 2 \rangle$ je $-x \leq y \leq x - x^2$. Užitím Fubiniových vět pak dostaváme

$$\begin{aligned} \int_M xy \, dA &= \int_0^2 \int_{-x}^{x-x^2} xy \, dy \, dx = \int_0^2 \left[\frac{1}{2} xy^2 \right]_{y=-x}^{y=x-x^2} \, dx = \\ &= \frac{1}{2} \int_0^2 (x(x-x^2)^2 - x^3) \, dx = -\frac{16}{15}. \end{aligned}$$

- **trojný integrál na čtverci:**

$$\begin{aligned}
\int_W (2x - y + z) \, dV &= \int_0^1 \int_1^2 \int_2^3 (2x - y + z) \, dz \, dy \, dx = \\
&= \int_0^1 \int_1^2 \left[2xz - yz + \frac{z^2}{2} \right]_{z=2}^{z=3} \, dy \, dx = \int_0^1 \int_1^2 \left(2x - y + \frac{5}{2} \right) \, dy \, dx = \\
&= \int_0^1 \left[2xy - \frac{y^2}{2} + \frac{5}{2}y \right]_{y=1}^{y=2} \, dx = \int_0^1 (2x + 1) \, dx = 2.
\end{aligned}$$

- **trojný integrál - Fubiniova věta:**

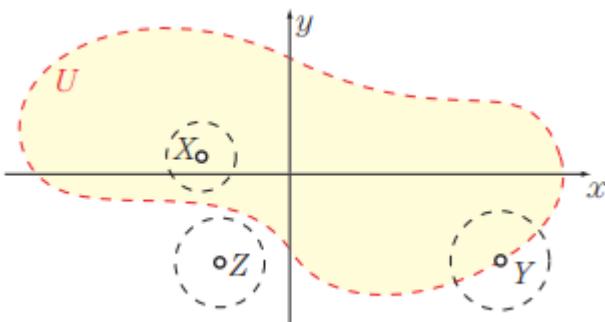
Zapišeme-li množinu M ve tvaru

$$M = \{(x, y) \in \mathbb{R}^2 : 0 \leq x \leq 1 \wedge 0 \leq y \leq 1 - x\},$$

můžeme použitím Fubiniové věty pro dvojný integrál náš trojný integrál převést na trojnásobný integrál. Potom

$$\begin{aligned}
\int_W \frac{1}{1+x+y} \, dV &= \int_0^1 \int_0^{1-x} \int_0^{1-x-y} \frac{1}{1+x+y} \, dz \, dy \, dx = \\
&= \int_0^1 \int_0^{1-x} \left[\frac{z}{1+x+y} \right]_{z=0}^{z=1-x-y} \, dy \, dx = \int_0^1 \int_0^{1-x} \frac{1-x-y}{1+x+y} \, dy \, dx = \\
&= \int_0^1 [2 \ln(1+x+y) - y]_{y=0}^{y=1-x} \, dx = \int_0^1 (2 \ln 2 - 2 \ln(x+1) + x - 1) \, dx = \\
&\quad = \frac{3}{2} - 2 \ln 2.
\end{aligned}$$

Limity funkcí více proměnných



- Pokud máme funkci spojitou v limitním bodě, pak lze hodnotu limity získat pouhým dosazením bodu do funkčního předpisu.

Příklad 3.1.1. Vypočtěte:

$$\lim_{(x,y) \rightarrow (1,2)} \frac{x^3y - xy^3 + 1}{(x-y)^2}.$$

Řešení: Do lomeného výrazu dosadíme bod $(1, 2)$:

$$\lim_{(x,y) \rightarrow (1,2)} \frac{x^3y - xy^3 + 1}{(x-y)^2} = \frac{2 - 2^3 + 1}{(-1)^2} = -5.$$

Limity funkcí více proměnných fungují na **podobném** principu jako limity funkce jedné proměnné. Limita funkce **existuje**, když hodnoty **libovolně malého okolí** bodu **spadají do ohraničeného pásu funkčních hodnot**. U funkce **dvou** proměnných bude okolím **kruh**, u tří proměnných **koule**. Limitu funkce více proměnných v bodě $p \in \mathbf{U}$ lze počítat pouze, pokud je tento bod **bodem hromadným** (X a Y na obr.) - bod jehož **každé prstencové** okolí má s množinou **U neprázdný průnik**. Body, které toto nesplňují, jsou body **izolované** (Z na obr.) a v nich **limitu funkce více proměnných počítat nelze**.

- **Otevřená množina** - Neobsahuje žádný bod ze své hranice.
- **Uzavřená množina** - Obsahuje všechny body své hranice.
- **Vnitřní bod** - Bod uvnitř množiny.
- **Hraniční bod** - Bod na hranici množiny.
- **Hranice množiny** - Množina všech hraničních bodů.

Další info

L'Hôpitalovo pravidlo

Umožňuje v některých případech vypočítat limitu podílu dvou funkcí. Říká, že limita podílu dvou funkcí, které splňují jisté předpoklady, je rovna limitě podílu derivací těchto funkcí.

$$\lim_{x \rightarrow a} \frac{f(x)}{g(x)} = \lim_{x \rightarrow a} \frac{f'(x)}{g'(x)}$$

Taylorova řada

Mocninná řada, vyjádřená jako **suma derivací funkce v bodě**. Pokud se jedná o rozvoj v **okolí bodu 0**, mluvíme o **Maclaurinově řadě**. **Taylorovy** a **Maclaurinovy** se využívají k approximaci funkcí.

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \dots = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$$

Odkazy:

- [Neurcity integral](#)
- [Integrace substitucí](#)
- [Spojitost funkce](#)
- [Funkce více proměnných](#)
- [VZORCE PRO INTEGROVÁNÍ](#)
- Derivace: [Tabulky](#)

18. Číselné soustavy a převody mezi nimi.

Číselné soustavy vyjadřují způsob reprezentace čísel. Podle způsobu určení hodnoty čísla z dané reprezentace rozlišujeme dva hlavní druhy číselných soustav: poziční číselné soustavy a nepoziční číselné soustavy. **Dnes** se obvykle používají **soustavy poziční**. Zápis čísla dané soustavy je posloupností symbolů, které se nazývají číslice.

Nepoziční číselná soustava

Číselné soustavy, ve kterých **není** hodnota číslice **dána jejím umístěním** v dané sekvenci číslic. Tyto způsoby zápisu čísel se **dnes** již téměř **nepoužívají** a jsou považovány za zastaralé.

Římská číselná soustava (římské číslice)

Římská číslice	I	V	X	L	C	D	M
Hodnota	1	5	10	50	100	500	1000

Římané psali číslici **4 jako IIII**, **40 jako XXXX** atd. tento zápis opravdu umožňuje zápis **číslic na libovolné pozice**. Pravidlo pro odečtení, že 4 se zapisuje jako IV, 40 jako XL atd., se začalo používat až ve středověku, komplikuje ale umístění číslic (největší se tak psala vlevo, nejmenší vpravo). Např. číslo **78** se běžně zapisuje jako **LXXVIII**. Lze ale zapsat také např. jako XXVLIII, XXIIILV - uvažujeme pravidlo odečtení a libovolně, pokud jej neuvažujeme např. IXLXVI, IIVILXL, ... Poziční zápis v římské číselné soustavě není možný, protože neexistuje symbol pro **nulu**.

- **Převod z římské do desítkové:** jednotlivé číslice vynásobíme jejich hodnotou a poté sečteme. Např. **LXXVI = 50 + 10 + 5 + 10 + 1 = 76**.
- **Převod z desítkové do římské:** musíme najít první číslici, jejíž hodnota je menší, než hodnota čísla v desítkové soustavě. Zapsat ji tolikrát, kolikrát se do čísla vejde, od čísla odečít zapsanou hodnotu a pokračovat další číslici. Např. převod čísla **32**: L - nevleze se, X - vleze se (3x), V - nevleze se, I - vleze se (2x) → **XXXII**.

Poziční číselné soustavy

Převládající způsob písemné reprezentace čísel. V tomto způsobu zápisu čísel je **hodnota každé číslice dána její pozicí v sekvenci symbolů**. Všechny poziční soustavy **musí** mít symbol pro **nulu**. Celá část je oddělena od zlomkové speciálním znakem (zpravidla řádovou čárkou či tečkou). Nejběžnější poziční číselnou soustavou je soustava **desítková**. Pro zápis hodnoty čísla v libovolné soustavě můžeme použít polynomiální zápis.

$$(132)_{16} = 2 \cdot 16^0 + 3 \cdot 16^1 + 1 \cdot 16^2 = 2 + 48 + 256 = 306$$

$$A = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_i \cdot 10^i + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0$$

Základ (báze, radix) číselné soustavy

Číslo definující **maximální počet číslic**, které jsou v dané soustavě k **dispozici**.

- **dvojková** (binární, **r=2**) – přímá implementace v digitálních elektronických obvodech (použitím logických členů).
- **osmičková** (oktální, oktalová, **r=8**)
- **desítková** (decimální, dekadická, **r=10**) – nejpoužívanější v běžném životě
- **dvanáctková (r=12)** – dnes málo používaná, ale dodnes z ní zbyly názvy prvních dvou řádů – **tucet** a veletucet.
- **šestnáctková** (hexadecimální, **r=16**) – používá se v oblasti informatiky, pro číslice 10 až 15 se používají písmena **A** až **F**.
- **šedesátková (r=60)** – používá se k měření času pro zlomky hodiny; číslice se obvykle zapisují desítkovou soustavou jako 00 až 59 a řády se oddělují **dvojtečkou**. Staré názvy prvních dvou řádů jsou **kopa** a **velekopa**.

Převody mezi pozičními soustavami

Běžný postup při převodu čísel mezi dvěma číselnými soustavami je **převod přes desítkovou soustavu**. Pokud však **základ jedné soustavy je mocninou základu soustavy druhé**, lze postupovat i **přímo**.

Substituční metoda převodu soustav

Lze použít pro libovolnou soustavu, pro člověka je však nejjednodušší pro převod do desítkové soustavy.

- Číslo se vyjádří polynomiálním zápisem v cílové soustavě (dekadická).
- Vypočítají se členy polynomu a sečtou.

$$\begin{aligned} & (10011,011)_2 \\ & = (1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3})_2 \\ & = (16 + 2 + 1 + 0,25 + 0,125)_{10} \\ & = (19,375)_{10} \end{aligned}$$

Metoda dělení základem

Vhodná pro převod celých čísel.

$$(109)_{10} / 2 = 54 \text{ zb. } 1(LSB)$$

$$(54)_{10} / 2 = 27 \text{ zb. } 0$$

$$(27)_{10} / 2 = 13 \text{ zb. } 1$$

$$(13)_{10} / 2 = 6 \text{ zb. } 1$$

$$(6)_{10} / 2 = 3 \text{ zb. } 0$$

$$(3)_{10} / 2 = 1 \text{ zb. } 1$$

$$(1)_{10} / 2 = 0 \text{ zb. } 1(MSB)$$

$$(109)_{10} = (1101101)_2$$

Protože dělení je pro člověka náročná operace, lze použít způsob, kdy hledáme největší mocninu cílové soustavy, která je menší než převáděná hodnota, zjistíme, kolikrát se do převáděného čísla vejde a odečteme tento násobek od převedeného čísla - získanou číslici zapíšeme na správné pozici čísla v soustavě, do které převádíme. Postup opakujeme, dokud nevyčerpáme všechny pozice cílového čísla. (Vhodné zejména u **dvojkové** soustavy, ve které známe hodnoty jednotlivých mocnin 2)

Metoda násobení základem

Vhodná pro **převod desetinných čísel**. Číslo se násobí základem soustavy, do které ho převádíme. Po každém kroku se sepíše celočíselná část. Končíme po **dosažení 0** nebo **požadované přesnosti**.

$$(0,6875)_{10} \cdot 2 = 1,375 = 1 + 0,375 = b_{-1} + 0,375 \text{ (MSB)}$$

$$(0,375)_{10} \cdot 2 = 0,75 = 0 + 0,75 = b_{-2} + 0,75$$

$$(0,75)_{10} \cdot 2 = 1,5 = 1 + 0,5 = b_{-3} + 0,5$$

$$(0,5)_{10} \cdot 2 = 1,0 = 1 + 0,0 = b_{-4} + 0,0 \text{ (LSB)}$$

$$(0,6875)_{10} = (0,1011)_2$$

Substituční převod desetinných čísel

Postupuje se stejně jako s celými čísly, ale tentokrát je lepší čísla odečítat a jak se dostaneme na desetinnou čárku, použijí se **záporné mocniny**.

$$32,625 - 2^5 = 0,625 \text{ (1)}$$

$$0,625 - 2^4 = 0,625 \text{ (0)}$$

$$0,625 - 2^3 = 0,625 \text{ (0)}$$

$$0,625 - 2^2 = 0,625 \text{ (0)}$$

$$0,625 - 2^1 = 0,625 \text{ (0)}$$

$$0,625 - 2^0 = 0,625 \text{ (0)}$$

desetinná čárka

$$0,625 - 2^{-1} = 0,125 \text{ (1)}$$

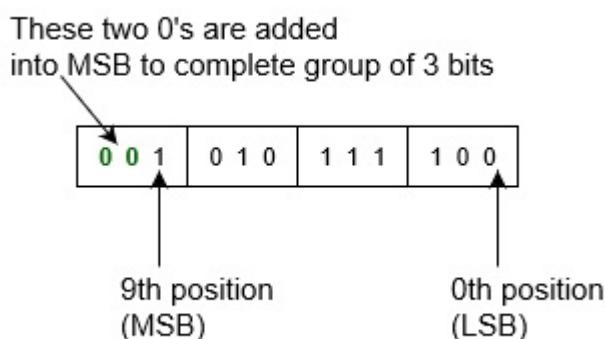
$$0,125 - 2^{-2} = 0,125 \text{ (0)}$$

$$0,125 - 2^{-3} = 0 \quad (1)$$

Výsledek bude tedy 100000,101

Převod mezi soustavami s mocninným rozdílem základů

- **Převod mezi dvojkovou a osmičkovou soustavou:** na zapsání všech osmičkových číslic stačí přesně **3 bity**. Stačí číslo rozdělit na **trojice** (číslo se doplní na násobek 3 přidáním nul před číslo - nezmění hodnotu) a každou zapsat jako **osmičkovou číslici**, respektive z osmičkové číslice vytvářet **trojice binárních číslic**. Např. převod binárního čísla 1010111100:



$$(1010111100)_2 = (001|010|111|100)_2 = (1|2|7|4)_8 = (1274)_8$$

$$(736)_8 = (7|3|6)_8 = (111|011|110)_2 = (111011110)_2$$

- **Převod mezi dvojkovou a šestnáctkovou soustavou:** na zapsání všech šestnáctkových číslic stačí přesně **4 bity**. Stačí číslo rozdělit na **nibly** (čtveřice) a každý zapsat jako **šestnáctkovou číslici**, respektive z šestnáctkové číslice vytvářet **nibly** (čtveřice binárních číslic).
 $(0001|0011|0010|1010)_2 = (1|3|2|A)_{16}$
 $(8|A|F)_{16} = (1000|1010|1111)_2$

19. Výroková logika a predikátová logika. Syntaxe a sémantika výrokové logiky. Splnitelnost a platnost. Logická ekvivalence a logický důsledek. Normální formy. Jazyk predikátové logiky prvního řádu. Syntaxe, termy a formule, volné a vázané proměnné.

Výroková logika

Výroková logika tvoří formální odvozovací systém, ze syntaktických a odvozovacích pravidel. <https://www.fit.vutbr.cz/~lengal/idm-2021/vyrokova-logika.pdf>

Syntax výrokové logiky

Syntax je dán abecedou a gramatikou:

- **abeceda:** Je tvořena spočetně nekonečnou množinou výrokových proměnných, logickými spojkami (logické symboly $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$) a logickými konstantami (**0** a **1**). $X = \{x, y, z, \dots, x_1, x_2, \dots, 0, 1, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$
- **gramatika:** vyjadřuje pravidla, jak můžeme tvořit formule výrokové logiky.
 - Je-li x výroková proměnná, tj. $x \in X$, pak **x, 0 a 1 jsou formule**.
 - Jsou-li ϕ a ψ formule, pak jsou formule i $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ a $(\phi \leftrightarrow \psi)$.
 - Formule výrokové logiky jsou právě všechny konečné řetězce získané pomocí předchozích dvou pravidel.

Pozor, výroková logika **neuvážuje** bez rozšíření **precedenci** jednotlivých **logických symbolů**.

Sémantika výrokové logiky

Sémantika formule určuje její **význam** a určuje, jakou **pravdivostní hodnotu** formule nabude pro **jednotlivá ohodnocení proměnných**. Tato hodnota se často definuje pomocí pravdivostní tabulky. Pravdivostní tabulka pro dvě formule ϕ a ψ říká, jaká bude výsledná hodnota formule získané aplikací dané **logické spojky** na ϕ a ψ .

φ	ψ	$\neg\varphi$	$\varphi \vee \psi$	$\varphi \wedge \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
0	0	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	1	0	0	0
1	1	0	1	1	1	1

Splňuje, splnitelnost, platnost, nesplňuje, neplatnost, nesplnitelnost

V následujících bodech uvažujeme ohodnocení proměnných $I : X \rightarrow \{0, 1\}$.

- $I \models \varphi$
- **I splňuje ϕ** - $I \models \phi$: zjišťujeme na základě daného ohodnocení proměnných I formule ϕ . Pokud je pro toto ohodnocení hodnota $\phi 1$, je I modelem formule ϕ a formuli splňuje.

- **splnitelná**: Formule je splnitelná, pokud existuje nějaké ohodnocení proměnných I takové, že $I \models \phi$ (pro nějaké ohodnocení I I splňuje ϕ). Z pravdivostní tabulky poznáme **splnitelnou formuli** tak, že se ve sloupci značící ϕ vyskytuje **alespoň jednou hodnota 1**.

- $I \models \varphi$
- **platná, tautologie, $\models \phi$** : formule je platná (tautologie) pokud je splněna libovolným ohodnocením proměnných. Pomocí pravdivostní tabulky můžeme platnou formuli poznat tak, že v **posledním sloupci** (sloupec s ϕ) tabulky jsou **samé hodnoty 1**.

- $I \not\models \varphi$
- **I nesplňuje ϕ** - $I \not\models \phi$: zjišťujeme na základě daného ohodnocení proměnných I formule ϕ . Pokud je pro toto ohodnocení hodnota $\phi 0$, není I modelem formule ϕ a formuli nesplňuje.

- $I \not\models \varphi$
- **neplatná, $\not\models \phi$** : formule ϕ je neplatná pokud existuje **ohodnocení** proměnných, které je **nesplňuje**. Pomocí pravdivostní tabulky takovou formuli poznáme tak, že by v posledním sloupci (sloupec s ϕ) je **alespoň jedna hodnota 0**.
 - **nesplnitelná, kontradikce**: formule je nesplnitelná, pokud **není splnitelná**, tj. ve sloupci s ϕ pravdivostní tabulky jsou **samé 0**.

$$\varphi_1: (x \wedge y) \rightarrow x$$

φ_1 je *platná i splnitelná*

$$\varphi_2: (x \vee y) \rightarrow x$$

φ_2 je *neplatná, ale splnitelná*

$$\varphi_3: (x \vee y) \wedge \neg(x \vee y)$$

φ_3 je *neplatná a nesplnitelná*

x	y	$x \wedge y$	$(x \wedge y) \rightarrow x$	$x \vee y$	$(x \vee y) \rightarrow x$	$\neg(x \vee y)$	$(x \vee y) \wedge \neg(x \vee y)$
0	0	0	1	0	1	1	0
0	1	0	1	1	0	0	0
1	0	0	1	1	1	0	0
1	1	1	1	1	1	0	0

Logická ekvivalence a logický důsledek

- $\varphi \Leftrightarrow \psi$
- **logická ekvivalence**: Dvě formule ϕ a ψ jsou (logicky) **ekvivalentní**, zapisováno $\phi \Leftrightarrow \psi$, pokud pro **všechna ohodnocení** proměnných I platí, že I je modelem (splňuje) ϕ právě tehdy, když I je modelem (splňuje) ψ .
 - **logický důsledek**: Formule ψ je **logickým důsledkem** formule ϕ , zapisováno $\phi \Rightarrow \psi$, tehdy, když pro **každé ohodnocení** proměnných I platí, že je-li I modelem (splňuje) ϕ , pak je I rovněž modelem (splňuje) ψ .

Neplést si s logickými spojkami implikace a bikondicionál, i když je význam podobný. Implikace a bikondicionál se používají uvnitř formulí, logický ekvivalence logický důsledek mezi formulemi.

Normální formy

Jedná se o formule, které splňují jistá syntaktická omezení.

Negační normální forma (NNF)

Formule je v NNF, pokud:

1. obsahuje jen následující logické spojky: **0, 1, \neg , \wedge , \vee**
2. **negace \neg** se vyskytuje jen **před proměnnými**.

Jedná se o literály (proměnné nebo negace proměnných) spojené konjunkcemi a disjunkcemi. Formule v NNF:

$$\begin{aligned} & x \wedge \neg y, \\ & (x \vee y) \wedge (\neg z \vee (\neg x \wedge \neg y)) \\ & x. \end{aligned}$$

Postup převodu obecné formule na formuli v NNF:

1. **Přepíšeme** postupně všechny **bikondicionály** \leftrightarrow ve formuli **za implikace**.

$$x \leftrightarrow y \rightsquigarrow (x \rightarrow y) \wedge (y \rightarrow x)$$

2. **Přepíšeme** postupně všechny **implikace** \rightarrow ve formuli **za negaci a**

$$x \rightarrow y \rightsquigarrow \neg x \vee y$$

disjunkci.

3. Pomocí **De Morganových** zákonů postupně **přesuneme negaci** co **nejhlouběji**.
4. Kdykoliv to jde, **eliminujeme dvojitou negaci** (dvojitá negace v NNF není povolena).

$$\neg(x \wedge y) \rightsquigarrow \neg x \vee \neg y$$

$$\neg(x \vee y) \rightsquigarrow \neg x \wedge \neg y$$

$$\neg\neg x \rightsquigarrow x$$

povolena).

Disjunktivní normální forma (DNF)

Formule je v DNF (sum of products, SoP) v případě, že je zapsána jako disjunkce konjunkcí literálů (konjunkce jsou uvnitř závorek, disjunkce na nejvyšší úrovni). V případě DNF se **konjunkcí** literálů se říká **klauzule**. Používá se následující známení, kde **i,j** je literál:

$$\bigvee_i \bigwedge_j \ell_{i,j}$$

Příklady formulí v DNF:

$$\begin{aligned}
 & (x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge z), \\
 & (x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge \neg x_5) \vee (x_2 \wedge \neg x_3 \wedge x_5), \\
 & (x \wedge y) \vee \neg z, \\
 & x \wedge \neg y, \\
 & x \vee \neg y, \\
 & 1. \\
 & 0.
 \end{aligned}$$

Postup převodu obecné formule na formuli v DNF:

1. Převedeme obecnou formuli na formuli v NNF.
2. Formuli v NNF převedeme do tvaru, kde jsou **všechny konjunkce pod disjunkcemi** pomocí **distributivního zákona**.

$$x \wedge (y \vee z) \rightsquigarrow (x \wedge y) \vee (x \wedge z)$$

Konjunktivní normální forma (CNF)

Formule je v CNF (product of sums, PoS) v případě, že je zapsána jako konjunkce disjunkcí literálů (disjunkce jsou uvnitř závorek, konjunkce na nejvyšší úrovni). V případě CNF se **disjunkcí** literálů se říká **klaузule**. Používá se následující znáčení, kde **li,j** je literál:

$$\bigwedge_i \bigvee_j \ell_{i,j}$$

Příklady formulí v CNF:

$$\begin{aligned}
 & (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee \neg y \vee z), \\
 & (x_1 \vee x_2 \vee x_3 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_3 \vee x_5), \\
 & (x \vee y) \wedge \neg z, \\
 & x \vee \neg y, \\
 & x \wedge \neg y, \\
 & 0. \\
 & 1.
 \end{aligned}$$

Postup převodu obecné formule na formuli v CNF:

1. Převedeme obecnou formuli na formuli v NNF.
2. Formuli v NNF převedeme do tvaru, kde jsou všechny **disjunkce pod konjunkcemi** pomocí **distributivního zákona**.

$$x \vee (y \wedge z) \rightsquigarrow (x \vee y) \wedge (x \vee z)$$

Jazyk predikátové logiky prvního řádu

Oproti výrokové logice **poskytuje** predikátová logika mnohem bohatší vyjadřovací prostředky. <https://www.fit.vutbr.cz/~lengal/idm-2021/predikatova-logika.pdf>

Abeceda predikátové logiky

1. **logické spojky**: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow,$
2. **proměnné**: $x, y, z, \dots, x_1, x_2, \dots \in X$, kde X je spočetně nekonečná množina proměnných
3. **kvantifikátory**: $\exists, \forall,$
4. **závorky**: $),(),$
5. **funkční symboly**: $f_1, f_2, \dots \in F (+_{/2}, \sin_{/1}, e_{/0}),$
6. **predikátové symboly**: $p_1, p_2, \dots (<_{/2}, \text{isnan}_{/1}),$
7. **predikátový symbol rovnosti**: $=_{/2}.$

Funkční a predikátové symboly mají **aritu**, která říká, s kolika parametry (operandy) daný symbol pracuje.

Gramatika predikátové logiky

- **termy**:
 - Je-li x proměnná ($x \in X$), pak řetězec „ x “ je **term**.
 - Je-li f funkční symbol s aritou n a t_1, \dots, t_n jsou **termy**, pak i řetězec „ $f(t_1, \dots, t_n)$ “ je term.
- Jazyk teorie grup se signaturou $\langle \mathcal{F} = \{\cdot_{/2}, e_{/0}\}, \mathcal{P} = \emptyset \rangle$. Příklady termů v tomto jazyce jsou následující:
 - „ $(z \cdot (x \cdot e)) \cdot y$ “,
 - „ e “,
 - „ x “ a
 - „ $e \cdot e$ “.
- **formule**:
 - Je-li p predikátový symbol s aritou n a t_1, \dots, t_n jsou **termy**, potom je řetězec „ $p(t_1, \dots, t_n)$ “ **formule** (toto platí i pro „vestavěný“ binární predikátový symbol $=$). Formuli tohoto tvaru říkáme **atomická formule**.
 - Jsou-li ϕ a ψ **formule**, pak jsou **formule** i řetězce „ $(\neg\phi)$ “, „ $(\phi \wedge \psi)$ “, „ $(\phi \vee \psi)$ “, „ $(\phi \rightarrow \psi)$ “ a „ $(\phi \leftrightarrow \psi)$ “.
 - Je-li ϕ formule a $x \in X$ proměnná, pak jsou formule i řetězce „ $(\exists x\phi)$ “ a „ $(\forall x\phi)$ “.

Jazyk teorie uspořádání se signaturou $\langle \mathcal{F} = \emptyset, \mathcal{P} = \{\leq_2\} \rangle$:

- $\forall x(x \leq x)$
- $\forall x\forall y((x \leq y \wedge y \leq x) \rightarrow x = y)$
- $\forall x\forall y\forall z((x \leq y \wedge y \leq z) \rightarrow x \leq z)$
- $\forall x\forall y(x \leq y \vee y \leq x)$

Syntax predikátové logiky

Syntax predikátové logiky tvoří jí abeceda a gramatika. Navíc funkční a predikátové symboly nejsou **pevně zafixovány**, ale lze je chápout jako *parametr* jazyka, který si volíme podle toho, co **chceme** v logice **vyjádřit**. Jedná se o signaturu jazyka predikátové logiky, která je dána jako dvojice **<množina funkčních symbolů, množina predikátových symbolů>**. Příklady jazyků predikátové ligiky.

1. Jazyk teorie uspořádání se signaturou $\langle \mathcal{F} = \emptyset, \mathcal{P} = \{\leq_2\} \rangle$:

- $\forall x(x \leq x)$
- $\forall x\forall y((x \leq y \wedge y \leq x) \rightarrow x = y)$
- $\forall x\forall y\forall z((x \leq y \wedge y \leq z) \rightarrow x \leq z)$
- $\forall x\forall y(x \leq y \vee y \leq x)$

2. Jazyk teorie grup se signaturou $\langle \mathcal{F} = \{\cdot_2, e_0\}, \mathcal{P} = \emptyset \rangle$:

- $\forall x(x \cdot e = x \wedge e \cdot x = x)$
- $\forall x\exists y(x \cdot y = e \wedge y \cdot x = e)$

3. Jazyk teorie množin se signaturou $\langle \mathcal{F} = \emptyset, \mathcal{P} = \{\in_2\} \rangle$:

- $\forall u(u \in x \rightarrow u \in y)$
- $\forall x\exists y\forall z(\forall u(u \in z \rightarrow u \in x) \rightarrow z \in y)$

4. Jazyk teorie polí se signaturou $\langle \mathcal{F} = \{read_{/2}, write_{/3}\}, \mathcal{P} = \emptyset \rangle$:

- $\forall x\forall y(\forall i(read(x, i) = read(y, i)) \rightarrow x = y)$

5. Jazyk elementární (tzv. Peanovy) aritmetiky se signaturou $\langle \mathcal{F} = \{0_0, S_{/1}, +_{/2}, \cdot_{/2}\}, \mathcal{P} = \emptyset \rangle$:

- $\forall x\forall y\exists z(x + y = z)$
- $\forall x\forall y(x \cdot S(y) = x \cdot y + x)$
- $\exists x\forall y(\neg(x = S(y)))$

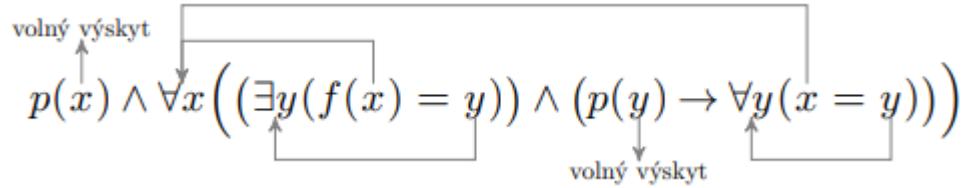
Vázané proměnné

Výskyt proměnné je ve formuli vázaný, pokud se nachází v oboru platnosti kvantifikátoru \exists nebo \forall . Pokud je výskyt proměnné vázaný, pak je vázaný nejbližším kvantifikátorem nad sebou. Příklad oborů platnosti jednotlivých kvantifikátorů:

$$p(x) \wedge \forall x \left((\exists y(f(x) = y)) \wedge (p(y) \rightarrow \forall y(x = y)) \right)$$

Volné proměnné

Volné proměnné jsou takové, které **nejsou vázané žádným kvantifikátorem**. Z předchozí formule jsou to tyto:



Proměnná je ve formuli **volná**, pokud v ní **alespoň jeden volný výskyt**. Formuli s **volnými proměnnými** říkáme **výroková forma**, formuli **bez volných proměnných** říkáme **uzavřená formule** nebo také **výrok**.

Sémantika predikátové logiky

Sémantika predikátové logiky je podstatně **komplikovanější** než u výrokové logiky. V predikátové logice musíme proměnným **přiřazovat hodnoty** z nějakého univerza a musíme **interpretovat funkční a predikátové symboly** (provádět **realizaci** jazyka). Např. abychom určili, jestli formule platí, musíme znát:

$$\forall x (f(y) < x)$$

- jakou hodnotu bude mít proměnná **y**,
- jaké všechny prvky bude uvažovat **kvantifikátor „ $\forall x$ “**,
- jaká je **sémantika** funkčního symbolu „**f/1**“ a
- jaká je **sémantika** predikátového symbolu „ **$</2$** “.

Příklady různých realizací jazyka:

Uvažujme jazyk predikátové logiky se signaturou $\langle \mathcal{F} = \{+_{/2}\}, \mathcal{P} = \emptyset \rangle$. Následují příklady realizací tohoto jazyka:

1. Realizace I_1 modelující sčítání přirozených čísel, kde
 - $D_{I_1} = \mathbb{N}$ a
 - $I_1(+)=+_{\mathbb{N}}$ (tj. sčítání přirozených čísel: $+_{\mathbb{N}} = \{(0, 0) \mapsto 0, (0, 1) \mapsto 1, (1, 0) \mapsto 1, (0, 2) \mapsto 2, \dots\}$).
2. Realizace I_2 modelující sčítání tříprvkových vektorů reálných čísel, kde
 - $D_{I_2} = \mathbb{R}^3$ a
 - $I_2(+)=\{([x_1, y_1, z_1], [x_2, y_2, z_2]) \mapsto [x_1 +_{\mathbb{R}} x_2, y_1 +_{\mathbb{R}} y_2, z_1 +_{\mathbb{R}} z_2] \mid x_1, x_2, y_1, y_2, z_1, z_2 \in \mathbb{R}\}$, kde $+_{\mathbb{R}}$ je sčítání reálných čísel.
3. Realizace I_3 modelující spojení (supremum) v Booleově algebře nad $\{0, 1\}$.
 - $D_{I_3} = \{0, 1\}$ a
 - $I_3(+)=\{(0, 0) \mapsto 0, (0, 1) \mapsto 1, (1, 0) \mapsto 1, (1, 1) \mapsto 1\}$.
4. Realizace I_4 modelující konkatenaci řetězců v jazyce Python, kde
 - $D_{I_4} = \text{str}$ (datový typ pro řetězec v jazyce Python) a
 - $I_4(+)=\{("ab", "cd") \mapsto "abcd", ("foo", "bar") \mapsto "foobar", \dots\}$

20. Boolovy algebry.

Booleova algebra je algebraická struktura se dvěma binárními (\wedge , \vee) a jedním unárním operátorem (\neg). Jedná se o **distributivní komplementární svaz**.

Booleova algebra je **šestice** (B , \wedge , \vee , \neg , 0 , 1), kde:

- B je neprázdná množina,
- $0 \in B$ a je nejmenší prvek,
- $1 \in B$ a je největší prvek,
- \wedge je **binární operace průsek** (infimum),
- \vee je **binární operace spojení** (supremum),
- \neg je **unární operace doplňku**.

Axiomy

- **Komutativita:** Binární operace u níž **nezáleží na pořadí prvků**.

$$x \vee y = y \vee x \quad x \wedge y = y \wedge x$$

- $| x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ aci.

Operaci **průsek** je možné **distribuovat přes** operaci **spojení** a operaci **spojení přes** operaci **průsek**.

- **Asociativita:** u operace nezáleží, jak jsou použity závorky u více operandů (někde nebývá jako axiom).

$$a \vee (b \vee c) = (a \vee b) \vee c \quad a \wedge (b \wedge c) = (a \wedge b) \wedge c$$

- **Neutralita 0 a 1:** **0** je neutrální prvek pro operaci **spojení** a **1** je neutrální

$$x \vee 0 = x \quad x \wedge 1 = x$$

prvek pro operaci **průsek**.

- **Komplementárnost:** existence doplňku (komplimentu)
- **Absorpce:** (někde nebývá jako axiom)

$$x \vee \neg x = 1$$

$$x \wedge \neg x = 0$$

$$a \vee (a \wedge b) = a$$

$$a \wedge (a \vee b) = a$$

Vlastnosti

- **Agresivita nuly** - Při průseku jakéhokoliv prvku s 0, je výsledek 0.

$$x \wedge 0 = 0$$

- **Agresivita jedničky** - Při spojení jakéhokoliv prvku s 1, je výsledek 1.

$$x \vee 1 = 1$$

- **Idempotence** - Opakovaným použitím na nějaký vstup vznikne stejný výstup.

Tedy jakákoliv binární operace prvku se sebou samým má za výsledek ten stejný prvek.

$$x \vee x = x$$

$$x \wedge x = x$$

- **Absorpce negace**
 $x \vee (\neg x \wedge y) = x \vee y, x \wedge (\neg x \vee y) = x \wedge y$
- **Dvojitá negace**
 $\neg(\neg x) = x$
- **0 a 1 jsou vzájemně komplementární**
 $\neg 0 = 1, \neg 1 = 0$
- **De Morganovy zákony**
 $\neg x \wedge \neg y = \neg(x \vee y), \neg x \vee \neg y = \neg(x \wedge y)$

Příklady Booleových algeber

- **triviální algebry:**
 - **0 = 1** (obsahují pouze jeden prvek)
 - všechny operace dávají stejný výsledek
- **algebry výroků:**
 - **0 - false** (nepravda), **1 - true** (pravda),
 - **\wedge - konjunkce, \vee - disjunkce, \neg - negace**
- **množinová algebra:**
 - **0 - prázdná množina (\emptyset), 1 - univerzum (U)**,
 - **\wedge - průnik, \vee - sjednocení, \neg - doplněk**
- **algebry elektrických obvodů:**
 - **0 - low** (log. 0), **1 - high** (log. 1)
 - **\wedge - AND, \vee - OR, \neg - NOT**

Funkce

Na Booleově algebrách lze realizovat **$2^2 \cdot 2^2 = 16$** funkcí.

Funkce	Název funkce	Logický člen	Algebraický výraz
f_0	konstanta		0
f_1	logický součet	NEBO(OR,ODER)	$a + b$
f_2	implikace		$\bar{a} + b$
f_3	implikace		$a + \bar{b}$
f_4	Shefferova funkce	NAND	$\bar{a} + \bar{b} = \overline{ab}$
f_5	logický součin	A (AND,UND)	ab
f_6	inhibice		$\bar{a}b$
f_7	inhibice		$a\bar{b}$
f_8	Pierceova funkce	NOR	$\bar{a} \cdot \bar{b} = \overline{a + b}$
f_9	identita		a
f_{10}	identita		b
f_{11}	ekvivalence		$ab + \bar{a}\bar{b}$
f_{12}	neekvivalence	Exclusive OR	$\bar{a}b + a\bar{b}$
f_{13}	negace	NE (NOT,NICHT)	\bar{a}
f_{14}	negace	NE (NOT,NICHT)	\bar{b}
f_{15}	konstanta		1

- **NOT** - Negace (doplňek)

Funkce	$Y = \bar{A}$	
Značení		Pravdivostní tabulka
norma	symbol	
ANSI/MIL	A ——————→ Q	
IEC	A ——————→ Y	$\begin{array}{ c c } \hline X(A) & Y \\ \hline 0 & 1 \\ 1 & 0 \\ \hline \end{array}$
DIN	—————→	

- **Opakovač** - Realizuje funkci **identity**. Lze použít jako **buffer**.

Funkce	$Y = A$	
Značení		Pravdivostní tabulka
norma	symbol	
ANSI/MIL	A ——————→ Q	
IEC	A ——————→ Y	$\begin{array}{ c c } \hline X(A) & Y \\ \hline 0 & 0 \\ 1 & 1 \\ \hline \end{array}$
DIN	—————→	

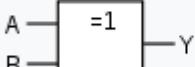
- **OR** - Disjunkce (spojení)

Funkce	$Y = A + B$	
Značení		Pravdivostní tabulka
norma	symbol	
ANSI/MIL	A ——————→ Q	
IEC	A ——————→ Y	$\begin{array}{ c c c } \hline X_1(A) & X_2(B) & Y \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ \hline \end{array}$
DIN	—————→	

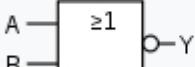
- **AND** - Konjunkce (průnik)

Funkce	$\mathbf{Y} = \mathbf{A} \cdot \mathbf{B}$		
Značení		Pravdivostní tabulka	
norma	symbol	$X_1(A)$	$X_2(B)$
ANSI/MIL		$X_1(A)$	$X_2(B)$
IEC		0	0
DIN		0	1
		1	0
		1	1

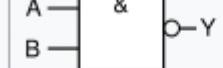
- **XOR** - negovaná ekvivalence - **Exkluzivní disjunkce**

Funkce	$\mathbf{Y} = \mathbf{A} \oplus \mathbf{B} = \overline{\mathbf{A}} \cdot \mathbf{B} + \mathbf{A} \cdot \overline{\mathbf{B}}$		
Značení		Pravdivostní tabulka	
norma	symbol	$X_1(A)$	$X_2(B)$
ANSI/MIL		$X_1(A)$	$X_2(B)$
IEC		0	0
DIN		0	1
		1	0
		1	1

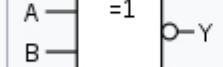
- **NOR** - Negovaná disjunkce - **Peirceova funkce**

Funkce	$\mathbf{Y} = \overline{\mathbf{A} + \mathbf{B}} = \overline{\mathbf{A}} \cdot \overline{\mathbf{B}}$		
Značení		Pravdivostní tabulka	
norma	symbol	$X_1(A)$	$X_2(B)$
ANSI/MIL		$X_1(A)$	$X_2(B)$
IEC		0	0
DIN		0	1
		1	0
		1	1

- **NAND** - Negovaná konjunkce - **Shefferova funkce**

Funkce	$Y = \overline{A \cdot B} = \overline{A} + \overline{B}$		
Značení		Pravdivostní tabulka	
norma	symbol		
ANSI/MIL		$X_1(A)$	$X_2(B)$
IEC		0	0
DIN		0	1
		1	0
		1	1

- **XNOR** - negovaná exkluzivní disjunkce - **Ekvivalence**

Funkce	$Y = \overline{A \oplus B} = A \cdot B + \overline{A} \cdot \overline{B}$		
Značení		Pravdivostní tabulka	
norma	symbol		
ANSI/MIL		$X_1(A)$	$X_2(B)$
IEC		0	0
DIN		0	1
		1	0
		1	1

Shefferova funkce

Pomocí Shefferovy funkce lze vyjádřit všechny ostatní funkce Booleovy algebry, viz:

$$\neg p \equiv \neg(p \wedge p)$$

$$p \wedge q \equiv \neg(\neg(p \wedge q)) \equiv \neg(\neg(p \wedge q) \wedge \neg(p \wedge q))$$

$$p \vee q \equiv \neg(\neg p \wedge \neg q) \equiv \neg(\neg(p \wedge p) \wedge \neg(q \wedge q))$$

$$p \rightarrow q \equiv \neg p \vee q \equiv \neg(p \wedge \neg q) \equiv \neg(p \wedge \neg(p \wedge q))$$

Peirceova funkce

Pomocí Piercovy funkce lze také vyjádřit všechny ostatní funkce Booleovy algebry, viz:

$$\neg p \equiv \neg(p \vee p)$$

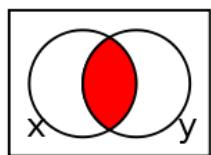
$$p \wedge q \equiv \neg(\neg p \vee \neg q) \equiv \neg(\neg(p \vee p) \vee \neg(q \vee q))$$

$$p \vee q \equiv \neg(\neg(p \vee q)) \equiv \neg(\neg(p \vee q) \vee \neg(p \vee q))$$

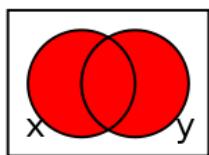
$$p \rightarrow q \equiv \neg p \vee q \equiv \neg(p \vee p) \vee q \equiv \neg(\neg(\neg(p \vee p) \vee q) \vee \neg(\neg(p \vee p) \vee q))$$

Vennovy diagramy

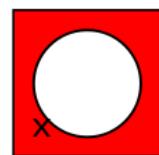
Grafické znázornění příslušnosti prvků do množin.



$$x \wedge y$$



$$x \vee y$$



$$\neg x$$

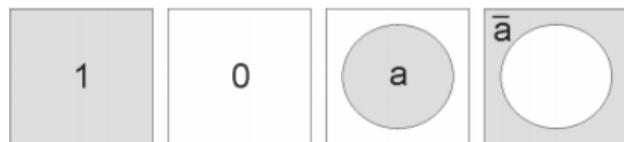


Diagram illustrating Boolean addition ($a + (b \cdot c)$):

On the left, three overlapping circles labeled 'a', 'b', and 'c' are shown. The regions 'a' and ' $(b \cdot c)$ ' are shaded grey, while the intersection 'a \cap (b \cap c)' is white. This represents the expression $a + (b \cdot c)$.

Diagram illustrating Boolean multiplication ($(a+b) \cdot (a+c)$):

On the left, two overlapping circles labeled 'a' and 'b' are shown. The regions '(a+b)' and '(a+c)' are shaded grey, while their intersection 'a' is white. This represents the expression $(a+b) \cdot (a+c)$.

Switching algebra

Jiný výraz pro booleovu algebru a vyjadřuje to, že v tomto systému (dvojkový systém) jsou prováděny všechny operace (**AND**, **OR**, **NOT**). Tedy použitím booleovy algebry děláme výpočty ve dvojkové soustavě.

21. Regulární jazyky a jejich modely (konečné automaty, regulární výrazy).

Regulární jazyk

Regulární jazyky jsou **nejjednodušší formální jazyky** v rámci **Chomského hierarchie**. Nad abecedou Σ je lze zavést následovně:

- prázdný jazyk \emptyset je regulární.
- pro každé $a, a \in \Sigma$, jazyk $\{ a \}$ je regulární.
- pokud A a B jsou regulární jazyky, jsou $A \cup B$ (sjednocení), $A \cdot B$ (konkatenace), a A^* (iterace, umožňuje jazyk prázdného řetězce $\{ \epsilon \}$) také regulární.
- žádné další jazyky regulární **nejsou**.

Dále platí, že jazyk jazyk je regulární, pokud:

- Existuje **konečný automat** (deterministický i nedeterministický), který **akceptuje** právě všechna slova z tohoto jazyka (**akceptuje/přijímá** tento jazyk).
- Existuje **regulární výraz**, který tento jazyk **značí**.
- Existuje **regulární gramatika**, která jej **generuje**.

Jazyk

Nechť Σ^* značí **množinu** všech **řetězců** nad Σ . Každá **podmnožina** $L \subseteq \Sigma^*$ je **jazyk** nad Σ . Tedy podmnožina řetězců abecedy. **Počet** všech **slov** jazyka je jeho **kardinalita**.

- **Konečný a nekonečný** - Jazyk L je **konečný**, pokud L obsahuje **konečný počet řetězců**, jinak je nekonečný.
- **Operace nad jazyky**
 - **Sjednocení** - Stejně jako u množin
 - **Průnik** - Stejně jako u množin
 - **Rozdíl** - Stejně jako u množin
 - **Doplňek** - Stejně jako u množin
 - **Konkatenace** - Nechť L_1 a L_2 jsou dva **jazyky** nad Σ . Konkatenace jazyků L_1 a L_2 je definována jako $L = L_1L_2 = \{xy: x \in L_1 \text{ a } y \in L_2\}$ $L = L_1L_2 = \{0, 1\}.\{2, 3\} = \{02, 03, 12, 13\}$. **Pozor** $L\{\epsilon\} = \{\epsilon\}L = L$, ale $\emptyset L = L\emptyset = \emptyset$.
 - **Reverzace** - Nechť L je jazyk nad abecedou Σ . Reverzace jazyka L , $\text{reverse}(L)$, je definována: $\text{reverse}(L) = \{\text{reverse}(x): x \in L\}$. Jedná se o převrácení slov abecedy. $\text{reverse}(\{02, 03, 12, 13\}) = \{20, 30, 12, 13\}$.

Abeceda

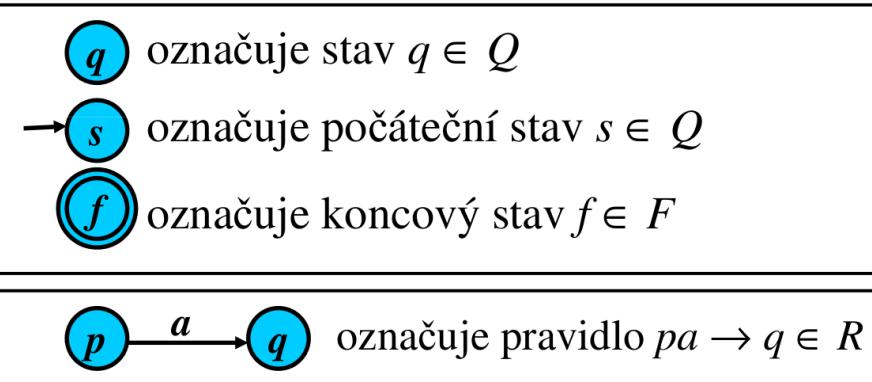
Konečná, neprázdná množina elementů, které nazýváme symboly.

Konečný automat (KA)

Konečný automat (KA) je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je **konečná množina stavů**,
- Σ je **vstupní abeceda**,
- R je **konečná množina pravidel** tvaru: $pa \rightarrow q$, kde $p, q \in Q, a \in \Sigma \cup \{\epsilon\}$,
- $s \in Q$ je **počáteční stav**,
- $F \subseteq Q$ je **množina koncových stavů**.

Graficky KA značíme následovně:



Přijímaný jazyk KA

Pokud s nějakou **vstupní posloupností znaků** se dostaneme až do **konečného stavu** automatu, pak automat tento **jazyk přijímá**.

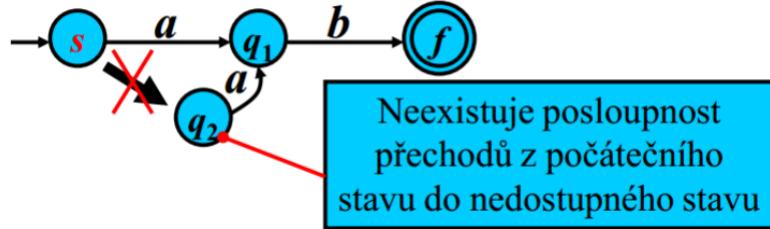
Ekvivalentní KA

Dva modely pro popis formálních jazyků (např. konečné automaty) jsou ekvivalentní, pokud specifikují **tentýž jazyk**.

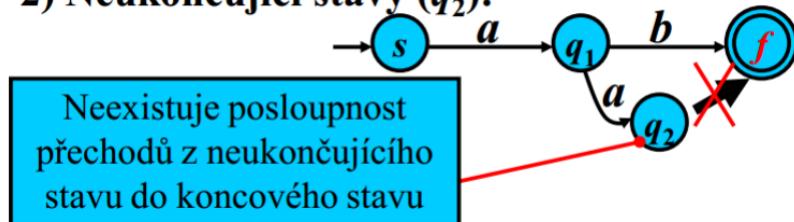
Typy konečných automatů

- **Nedeterministický**: Může obsahovat ϵ přechody a existují stavy, ze kterých lze s načtením **stejného symbolu** přejít do **více stavů**.
- **Bez epsilon přechodů** - Neobsahuje ϵ přechody.
- **Deterministický** - Neobsahuje ϵ přechody a z každého **stavu** může přejít **maximálně do jednoho** dalšího s načtením **stejného znaku**.
- **Úplný deterministický** - Pro **každý znak** abecedy existuje **právě jeden** přechod v **každém stavu** (nepotřebné směřují do **false stavu**). **Nemůže se zaseknout**.
- **Dobře specifikovaný** - **Nemá nedostupné** stavy a má maximálně **jeden neukončující** stav (false stav). Pro každý konečný automat existuje ekvivalentní dobré specifikovaný konečný automat.

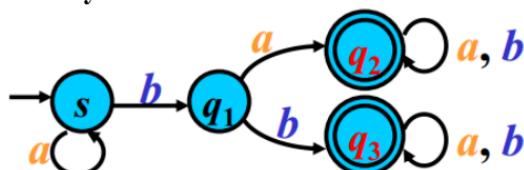
1) Nedostupné stavy (q_2):



2) Neukončující stavy (q_2):



- **Minimální** - Pokud obsahuje pouze rozlišitelné stavy. Pro dobré specifikovaný KA existuje právě jeden minimální KA



• **s** a **q_1** jsou **rozlišitelné**, protože např. pro $w = \text{a}$:

$$\begin{aligned} s\text{a} &\vdash s, s \notin F \\ q_1\text{a} &\vdash q_2, q_2 \in F \end{aligned}$$

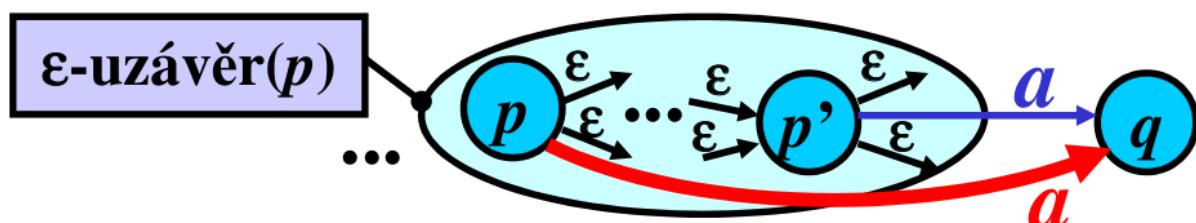
• **q_2** a **q_3** jsou **nerozlišitelné**, protože pro každé $w \in \Sigma^*$:

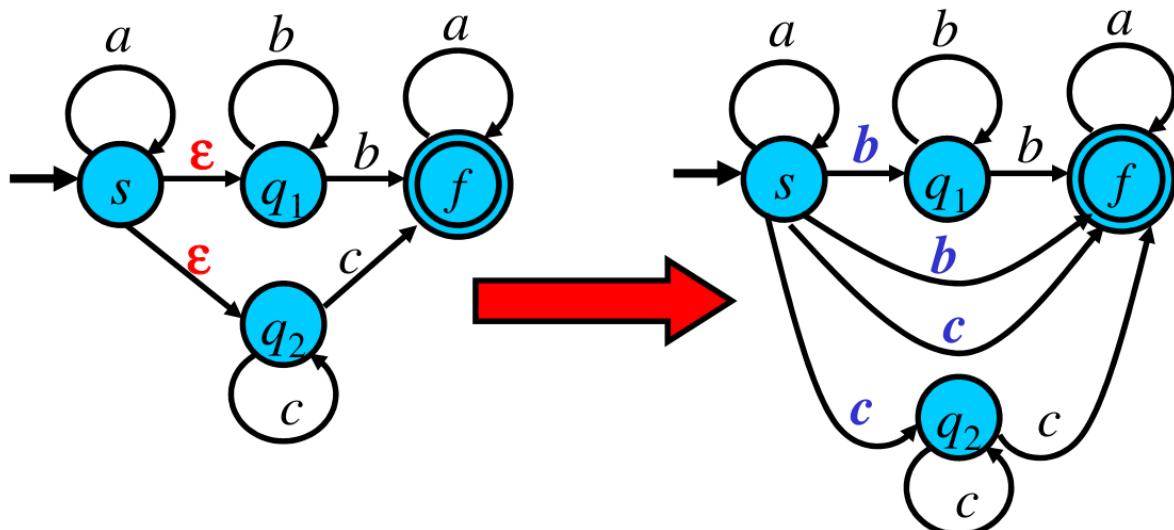
$$\begin{aligned} q_2w &\vdash^* q_2, q_2 \in F \\ q_3w &\vdash^* q_3, q_3 \in F \end{aligned}$$

Determinizace KA

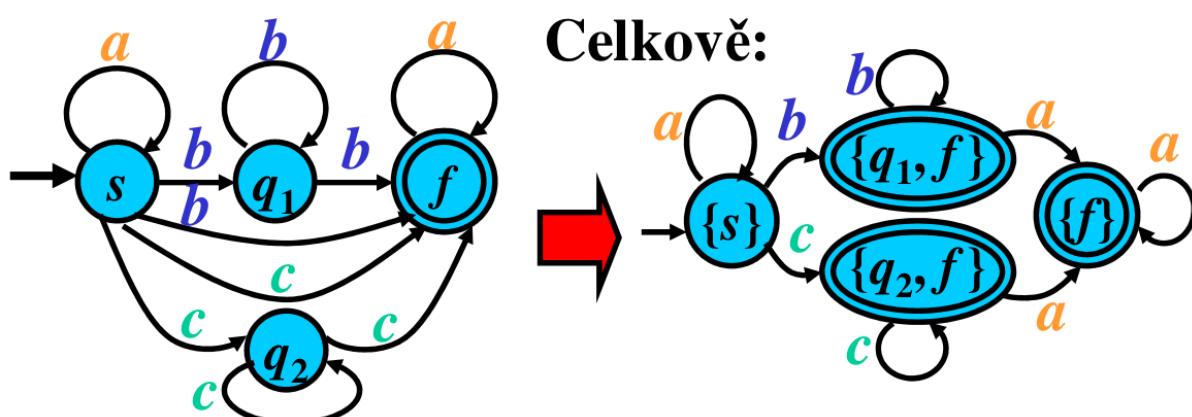
Převod nedeterministického KA na deterministický. Pokud má nedeterministický KA **n** stavů, má jeho deterministická varianta nejvíce **2^n stavů**. Každý nedeterministický automat má svoji deterministickou variantu. Postup:

1. Odstranění ϵ přechodů pomocí ϵ uzávěru - množina stavů, do kterých se můžeme dostat přečtením libovolného symbolu z abecedy.





- Z počátečního stavu vytváříme nové stavy tak, že tyto stavy jsou množinou stavů, do kterých se můžeme dostat přečtením libovolného symbolu (prázdné stavy neuvažujeme). Z takto vzniklých nových stavů postupujeme dále obdobně (je nutné dát pozor, aby byly brány v potaz všechny výstupy vzniklé množiny stavů). Za nové koncové stavy označíme ty, které obsahují



alespoň jeden původní koncový stav.

Shrnutí:

	KA	KA bez ε-přech.	DKA	Úplný KA	DSKA
Počet všech pravidel tvaru $p \rightarrow q$, kde $p, q \in Q$	0-n	0	0	0	0
Počet pravidel tvaru $pa \rightarrow q$, pro libovolné $p \in Q$ a libovolné $a \in \Sigma$	0-n	0-n	0-1	1	1
Počet všech nedostupných stavů	0-n	0-n	0-n	0-n	0
Počet všech neukončujících stavů	0-n	0-n	0-n	0-n	0-1
Počet všech možných těchto automatů pro jeden regulární jazyk	∞	∞	∞	∞	∞

Regulární výraz (RV)

Jedná se o výrazy s operátory “.”, “+”, “*”, které značí v tomto pořadí **konkatenaci**, **sjednocení** a **iteraci**. Nechť Σ je abeceda. Regulární výrazy nad abecedou Σ a **jazyky**, které **značí**, jsou definovány následovně:

- \emptyset je **RV** značící prázdnou množinu (**prázdný jazyk**),
- ϵ je **RV** značící jazyk $\{ \epsilon \}$,
- a , kde $a \in \Sigma$, je **RV** značící jazyk $\{ a \}$,
- Nechť r a s jsou regulární výrazy značící po řadě jazyky **L_r** a **L_s**, potom:
 - $(r.s)$ je RV značící jazyk $L = L_r L_s$,
 - $(r+s)$ je RV značící jazyk $L = L_r \cup L_s$,
 - (r^*) je RV značící jazyk $L = L_r^*$.

Závorky lze redukovat zavedením priority operátorů

Priority: $*$ > $.$ > $+$

RV $r.s$ může být zapsán jako rs

RV rr^* nebo r^*r může být zapsán jako r^+

Příklady RV:

$r_1 = ab + ba$	značí $L_1 = \{ab, ba\}$
$r_2 = a^+b^*$	značí $L_2 = \{a^n b^m : n \geq 1, m \geq 0\}$
$r_3 = ab(a + b)^*$	značí $L_3 = \{x : ab \text{ je prefix } x\}$
$r_4 = (a + b)^*ab(a + b)^*$	značí $L_4 = \{x : ab \text{ je podřetězec } x\}$

Prázdné slovo (ϵ)

Prázdný řetězec, který ale vyhovuje jazyku (je možné jej vygenerovat a přijmout).

Řetězec

Jakákoli posloupnost terminálních a neterminálních symbolů (znaků).

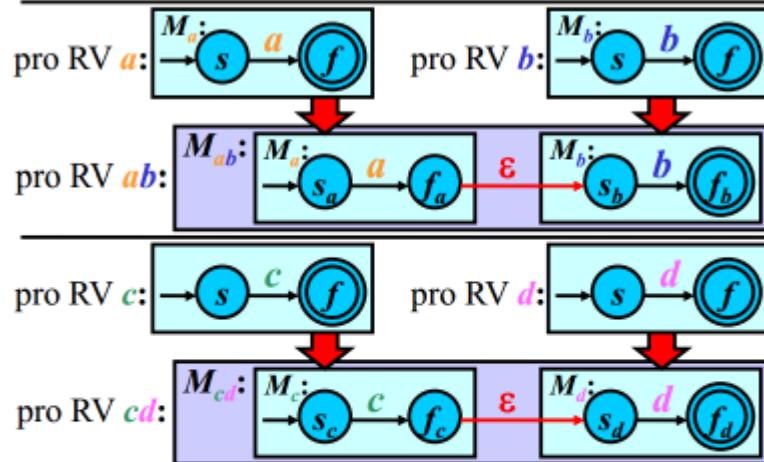
Nechť Σ je abeceda.

- ϵ je řetězec nad abecedou Σ
- pokud x je řetězec nad Σ a $s \in \Sigma$, potom xs je **řetězec** nad abecedou Σ

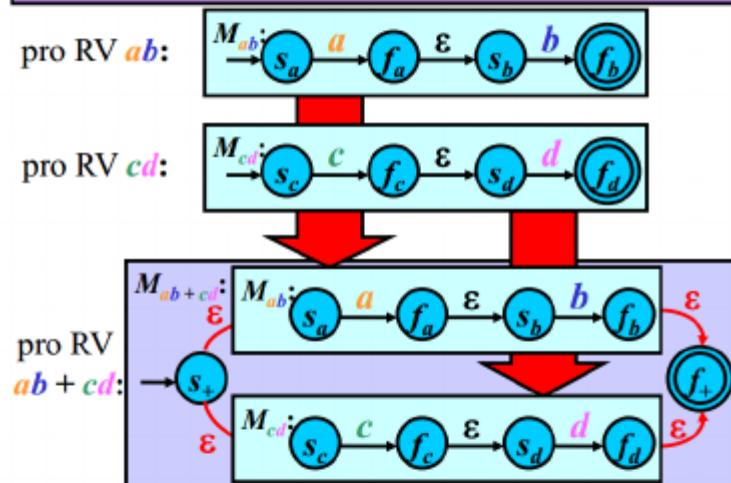
Převod regulárního výrazu na konečný automat

Převod z RV na KA: Příklad 1/3

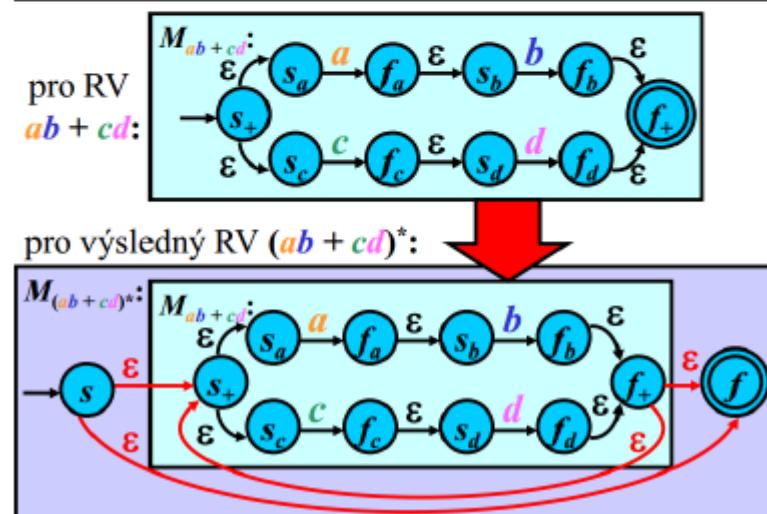
Převeďme RV $r = ((ab) + (cd))^*$ na ekvivalentní KA M



Převod z RV na KA: Příklad 2/3



Převod z RV na KA: Příklad 3/3



Regulární gramatika

Každá **regulární gramatika popisuje regulární jazyk** Je to čtveřice: **(T, N, P, S)**

- T - Konečná množina terminálních symbolů,
- N - Konečná množina neterminálních symbolů,
- P - Konečná množina pravidel, ve tvaru:
 - $X \rightarrow wY$ (w je řetězec)
 - $X \rightarrow w$
 - $S \rightarrow \epsilon$, poté se však nesmí S vyskytovat na pravé straně pravidla.
- S - Počáteční symbol.

Existují **pravé** ($X \rightarrow wY$) a **levé** ($X \rightarrow Yw$) **regulární gramatiky**, které jsou si **ekvivalentní**.

Příklady:

Příklad: $G = (\{a, b\}, \{A, B, C\}, P, A)$

$$\begin{aligned} P = \{ & \\ & \bullet A \rightarrow \epsilon \mid aB \\ & \bullet B \rightarrow bC \\ & \bullet C \rightarrow a \end{aligned}$$

$$\begin{aligned} A &\rightarrow Bb \\ B &\rightarrow \epsilon/Ba/Bb \end{aligned}$$

It derives the language that contains all the strings which end with b.
i.e. $L' = \{b, bb, ab, aab, bab, abb, bbb, \dots\}$

Pumping lemma

Používá se k dokázání, že jazyk **NENÍ REGULÁRNÍ** (používá se důkaz sporem), nemůže tedy dokázat, že jazyk je regulární. Nechť L je **RJ**. Pak existuje $k \geq 1$ takové, že: pokud $z \in L$ a $|z| \geq k$, pak existuje $u, v, w: z = uvw$,

- $v \neq \epsilon$
- $|uv| \leq k$
- pro každé $m \geq 0$, $uv^mw \in L$

[Pumping Lemma \(For Regular Languages\) | Example 1](#)

Odkazy

- Definice z IFJ: [IFJ Teorie](#)
- [Minimalizace a kanonizace, nedeterministické konečné automaty a determinizace](#)

22. Bezkontextové jazyky a jejich modely (zásobníkové automaty, bezkontextové gramatiky).

Bezkontextový jazyk (BKJ)

Jedná se o formální jazyk, pro který platí:

- Generuje jej **bezkontextová gramatika**,
- Přijímá (akceptuje) jej **zásobníkový automat**.

Nejznámějším bezkontextovým jazykem je jazyk $L(G) = \{(a^n)(b^n) : n \geq 0\}$

Bezkontextová gramatika

Slouží pro generování BKJ. Je to čtveřice $G = (N, T, P, S)$, kde:

- N - abeceda **neterminálů**,
- T - abeceda **terminálů**, přičemž $N \cap T = \emptyset$,
- P - konečná **množina pravidel** tvaru $A \rightarrow x$, kde $A \in N$, $x \in (N \cup T)^*$,
- S - počáteční neterminál, $S \in N$.

$$G = (N, T, P, S), \text{ kde } N = \{S\}, T = \{a, b\},$$

$$P = \{1: S \rightarrow aSb, 2: S \rightarrow \epsilon\}$$

$$S \Rightarrow \epsilon [2] \xrightarrow{} L(G) = \{a^n b^n : n \geq 0\}$$

$$S \Rightarrow aSb [1] \Rightarrow ab [2]$$

$$S \Rightarrow aSb [1] \Rightarrow aaSbb [1] \Rightarrow aabb [2]$$

⋮

$L = \{a^n b^n : n \geq 0\}$ je bezkontextový jazyk.

Deriva

Definice: Nechť $G = (N, T, P, S)$ je BKG.

Jazyk generovaný BKG G , $L(G)$, je definován:

$$L(G) = \{w : w \in T^*, S \xrightarrow{*} w\}$$

Definice: Nechť $G = (N, T, P, S)$ je BKG. Nechť

$u, v \in (N \cup T)^*$ a $p = A \rightarrow x \in P$. Potom, uAv

přímo derivuje uxv za použití p v G , zapsáno

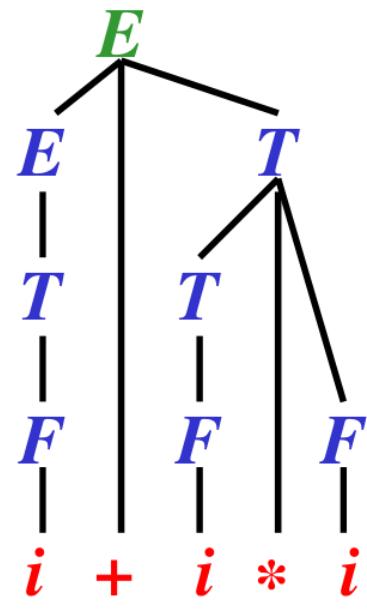
$uAv \Rightarrow uxv [p]$ nebo zjednodušeně $uAv \Rightarrow uxv$.

Jedná se o změnění řetězce použitím pravidla na neterminál. Pokud $uAv \Rightarrow uxv$ v G , můžeme říct, že G provádí derivační krok z uAv do uxv .

$$G = (N, T, P, \textcolor{green}{E}), \text{ kde } N = \{\textcolor{blue}{E}, \textcolor{blue}{F}, \textcolor{blue}{T}\}, T = \{\textcolor{red}{i}, +, *, (,)\},$$
$$P = \{ \begin{array}{lll} \textcolor{magenta}{1}: \textcolor{blue}{E} \rightarrow \textcolor{blue}{E} + \textcolor{blue}{T}, & \textcolor{magenta}{2}: \textcolor{blue}{E} \rightarrow \textcolor{blue}{T}, & \textcolor{magenta}{3}: \textcolor{blue}{T} \rightarrow \textcolor{blue}{T} * \textcolor{blue}{F}, \\ \textcolor{magenta}{4}: \textcolor{blue}{T} \rightarrow \textcolor{blue}{F}, & \textcolor{magenta}{5}: \textcolor{blue}{F} \rightarrow (\textcolor{blue}{E}), & \textcolor{magenta}{6}: \textcolor{blue}{F} \rightarrow \textcolor{red}{i} \end{array} \}$$

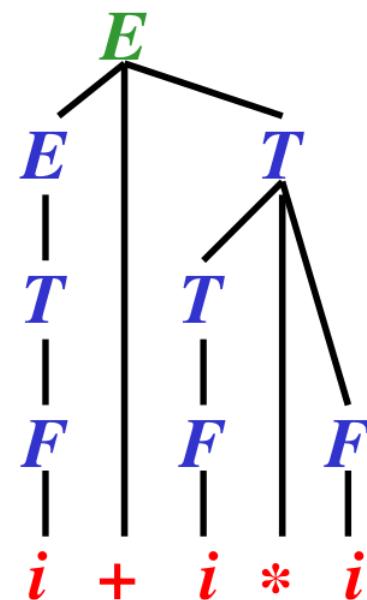
- **Nejlevější derivace:** Během nejlevějšího derivačního kroku je přepsán nejlevější neterminál.

$$\begin{aligned}
 \underline{E} &\Rightarrow_{lm} \underline{E} + T & [1] \\
 &\Rightarrow_{lm} \underline{T} + T & [2] \\
 &\Rightarrow_{lm} \underline{F} + T & [4] \\
 &\Rightarrow_{lm} i + \underline{T} & [6] \\
 &\Rightarrow_{lm} i + \underline{T} * F & [3] \\
 &\Rightarrow_{lm} i + \underline{F} * F & [4] \\
 &\Rightarrow_{lm} i + i * \underline{F} & [6] \\
 &\Rightarrow_{lm} i + i * i & [6]
 \end{aligned}$$



- **Nejpravější derivace:** Během nejpravějšího derivačního kroku je přepsán nejpravější neterminál.

$$\begin{aligned}
 \underline{E} &\Rightarrow_{rm} \underline{E} + \underline{T} & [1] \\
 &\Rightarrow_{rm} \underline{E} + T * \underline{F} & [3] \\
 &\Rightarrow_{rm} \underline{E} + \underline{T} * i & [6] \\
 &\Rightarrow_{rm} \underline{E} + \underline{F} * i & [4] \\
 &\Rightarrow_{rm} \underline{E} + i * i & [6] \\
 &\Rightarrow_{rm} \underline{T} + i * i & [2] \\
 &\Rightarrow_{rm} \underline{F} + i * i & [4] \\
 &\Rightarrow_{rm} i + i * i & [6]
 \end{aligned}$$



$A \rightarrow x$ znamená, že A má být přepsáno na x

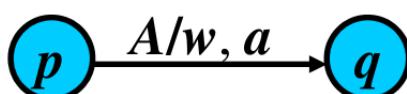
Zásobníkový automat

Konečný automat rozšířený o **zásobník**. Pro každou **BKG** existuje zásobníkový **automat** co ji přijímá. Zásobníkový automat (ZA) je **sedmice**:

$M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde:

- Q je konečná množina stavů,
- Σ je vstupní abeceda,
- Γ je zásobníková abeceda,
- R je konečná množina pravidel tvaru $Apa \rightarrow wq$, kde $A \in \Gamma$; $p, q \in Q$;
 $a \in \Sigma \cup \{\epsilon\}$, $w \in \Gamma^*$, (R je konečná relace z $\Gamma \times Q \times (\Sigma \cup \{\epsilon\})$ do $\Gamma^* \times Q$)
- $s \in Q$ je počáteční stav,
- $S \in \Gamma$ je počáteční symbol na zásobníku,
- $F \subseteq Q$ je množina koncových stavů.

Interpretace pravidel: $Apt \rightarrow wq \in R$ znamená, že pokud je aktuální stav p , aktuální symbol na vstupní pásce t a symbol na vrcholu zásobníku A , potom zásobníkový automat může přečíst t a na zásobníku nahradit A za w a přejít ze stavu p do q .

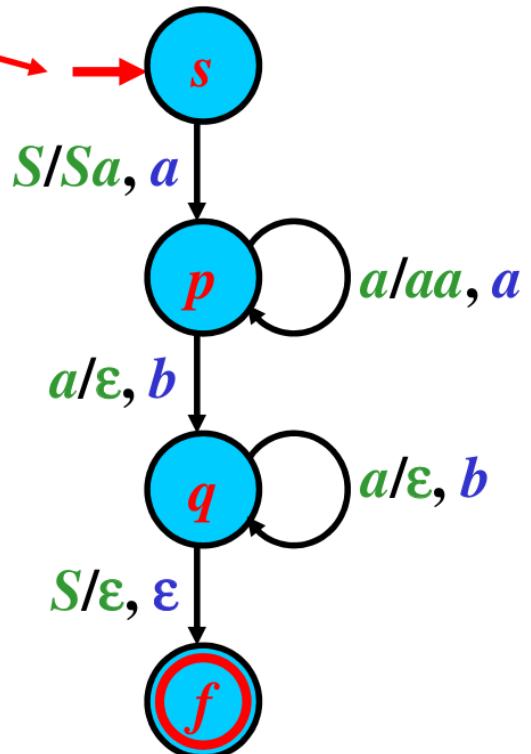


označuje $Apa \rightarrow wq \in R$

$$M = (Q, \Sigma, \Gamma, R, s, S, F)$$

kde:

- $Q = \{s, p, q, f\}$;
- $\Sigma = \{a, b\}$;
- $\Gamma = \{a, S\}$;
- $R = \{Ssa \rightarrow Sap, apa \rightarrow aap, apb \rightarrow q, aqb \rightarrow q, Sq \rightarrow f\}$
- $F = \{f\}$



Typy přijímaných jazyků ZA

- Přechodem do koncového stavu: ZA M přijímá jazyk L , pokud se čtením

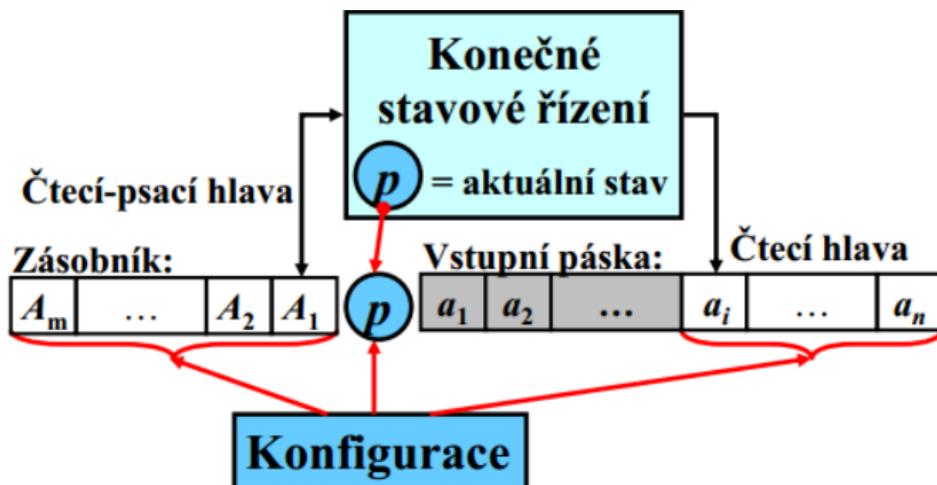
všech řetězců jazyka L dostane do koncového stavu.

- **Vyprázdněním zásobníků:** ZA M přijímá jazyk L , pokud se čtením všech řetězců jazyka L vyprázdní jeho zásobník.
- **Přechodem do koncového stavu a vyprázdněním zásobníku:** přijímá jazyk L , pokud se čtením všech řetězců jazyka L dostane do koncového stavu a současně se vyprázdní jeho zásobník.

Tyto tři typy ZA jsou si **ekvivalentní** a existují algoritmy pro **převody** mezi nimi.

Konfigurace ZA

Konfigurace ZA M je řetězec $\chi \in \Gamma^* Q \Sigma^*$. Jedná se o **aktuální stav zásobníku**, **aktuálního stavu** a **vstupní pásky**, jejíž část **nebyla** ještě přečtena.

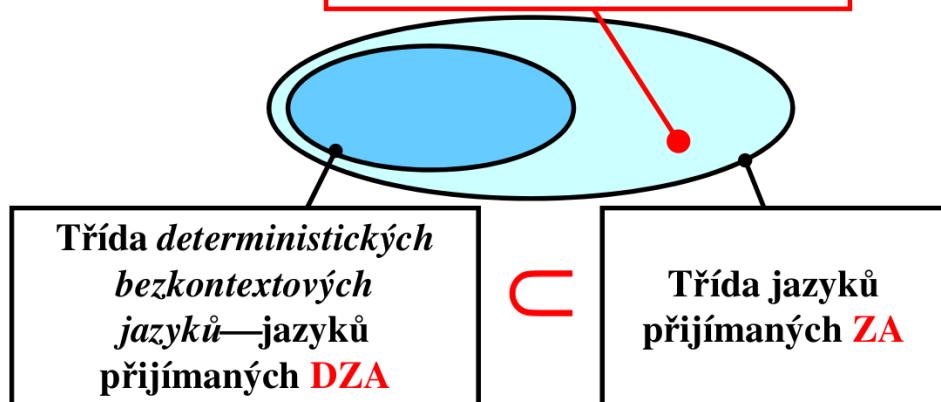


Deterministický zásobníkový automat (DZA)

Může provést z každé konfigurace **maximálně jeden přechod**. M je deterministický ZA, pokud pro každé pravidlo tvaru $Apa \rightarrow wq \in R$ platí, že množina $R - \{Apa \rightarrow wq\}$ (množinový rozdíl) neobsahuje **žádné pravidlo** s levou stranou **Apa** nebo **Ap**, jinak řešeno **levá strana** pravidel je v množině R vždy **pouze jednou** - je **unikátní**. DZA je **podmnožinou KA**, DZA nemusí přijímat některé jazyky, která KA přijímá → KA je **silnější** než DKA.

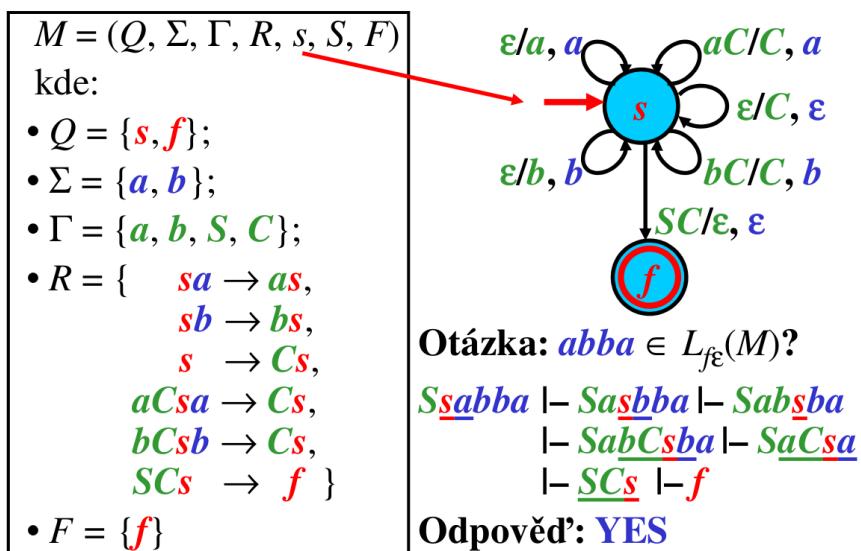
Ilustrace:

$$L = \{xy: x, y \in \Sigma^*, y = \text{reversal}(x)\}$$

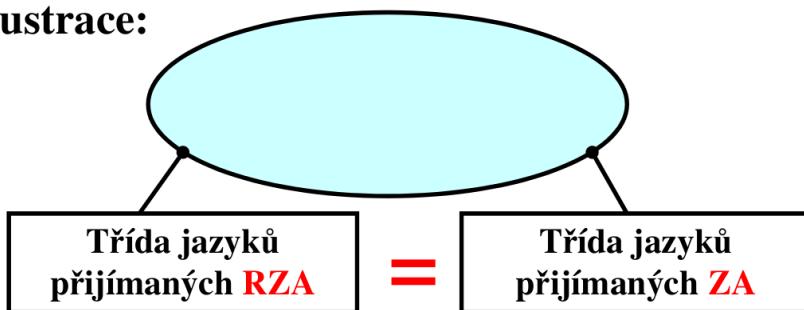


Rozšířený zásobníkový automat (RZA)

Z vrcholu zásobníku v **RZA** lze číst celý řetězec (v ZA to byl pouze jeden symbol). Pravidla jsou tak definována takto: **R** je konečná množina pravidel tvaru: $vpt \rightarrow wq$, kde $v, w \in \Gamma^*$; $p, q \in Q$; $t \in \Sigma \cup \{\epsilon\}$. RZA je ekvivalentní s ZA (jsou stejně silné). Mohou existovat RZA, které přijímají jazyky přechodem do koncového stavu, vyprázdněním zásobníku nebo vyprázdněním zásobníku a současně přechodem do koncového stavu. Opět jsou tyto typy RZA ekvivalentní. Následuje příklad RZA.



Illustrace:



RZA a ZA jako modely pro synt. analýzu

RZA nebo **ZA** mohou simulovat konstrukci derivačního stromu pro **BKG**. Používají se k tomu dva přístupy.

1) Shora dolů



Z S směrem ke vstupnímu řetězci

2) Zdola nahoru



Ze vstupního řetězce směrem k S

Převod z BKG na RZA (pro SA zdola nahoru)

1. obrázek je pro SA zdola nahoru, 2. pro SA shora dolů.

$$Q := \{s, f\};$$

$$\Sigma := T;$$

$$\Gamma := N \cup T \cup \{\#\};$$

Konstrukce množiny R :

- **for each** $a \in \Sigma$: přidej $sa \rightarrow as$ do R ;
- **for each** $A \rightarrow x \in P$: přidej $xs \rightarrow As$ do R ;
- přidej $\#Ss \rightarrow f$ do R ;

$$F := \{f\};$$

$$Q := \{s\};$$

$$\Sigma := T;$$

$$\Gamma := N \cup T;$$

Konstrukce množiny R :

- **for each** $a \in \Sigma$: přidej $asa \rightarrow s$ do R ;
- **for each** $A \rightarrow x \in P$: přidej $As \rightarrow ys$ do R , kde $y = \text{reversal}(x)$;

$$F := \emptyset;$$

$G = (N, T, P, S)$, kde:

$$N = \{S\}, T = \{(,)\}, P = \{S \rightarrow (S), S \rightarrow ()\}$$

Máme nalézt: RZA M , pro který platí: $L(G) = L(M)_f$

Např. (pořadí obrázků zůstává):

$M = (Q, \Sigma, \Gamma, R, s, \#, F)$, kde:

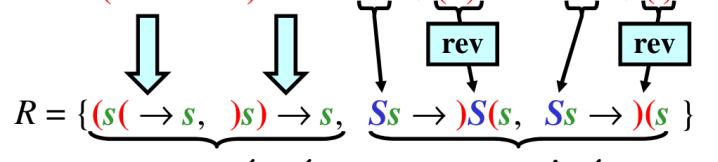
$$Q = \{s, f\}, \Sigma = T = \{(,)\}, \Gamma = \{(,), S, \#\}, F = \{f\}$$

$$R = \{s(\rightarrow (s, s) \rightarrow)s, (S)s \rightarrow Ss, ()s \rightarrow Ss, \#Ss \rightarrow f\}$$

$M = (Q, \Sigma, \Gamma, R, s, S, F)$ kde:

$$Q = \{s\}; \quad \Sigma = T = \{(,)\}; \quad \Gamma = N \cup T = \{S, (,)\}$$

$$\text{“(“} \in T \quad \text{“)”} \in T \quad S \rightarrow (S) \in P \quad S \rightarrow () \in P$$



Modely pro syntaktickou analýzu zdola nahoru $F = \emptyset$

RZA $M = (Q, \Sigma, \Gamma, R, s, \#, F)$ ($\#$ je počáteční symbol na zásobníku, s je počáteční stav)

1. **M** obsahuje **shiftovací pravidla**, která **přesouvají** vstupní symboly **na zásobník**. (tvorba: Pro každé $t \in \Sigma$: přidej $st \rightarrow ts$ do R .)
2. **M** obsahuje **redukční pravidla**, která **simulují aplikaci gramatických pravidel** pozpátku. (tvorba: Pro každé $A \rightarrow x \in P$ v G : přidej $xs \rightarrow As$ do R)
3. **M** také obsahuje **speciální pravidlo** $\#Ss \rightarrow f$, pomocí kterého provede **M** přechod do **koncového stavu**.

Modely pro syntaktickou analýzu shora dolů

ZA M = (Q, Σ, Γ, R, s, S, F) (F = prázdná množina)

1. **M** obsahuje **porovnávací pravidla**, která **porovnají** symbol z vrcholu **zásobníku** a aktuální symbol ze **vstupní pásky**. (tvorba: pro každé $a \in \Sigma$: přidej $asa \rightarrow s$ do R)
2. **M** obsahuje **expanzivní pravidla**, která **simulují gramatická pravidla**. (tvorba: pro každé $A \rightarrow t_1\dots t_n \in P$ v G , přidej $As \rightarrow t_n\dots t_1s$ do R ; = **reversal(t₁...t_n)**)

odkazy:

- Definice z IFJ: [IFJ Teorie](#)

23. Struktura překladače a charakteristika fází překladu (lexikální analýza, deterministická syntaktická analýza a generování kódu).

Překladač

Překladač čte **zdrojový program** překládá jej na **cílový program**.

- **Vstup** - Text zdrojového kódu ve zdrojovém jazyce (obvykle vyšší programovací jazyk - **C++, Go, C#, Java**).
- **Výstup** - Cílový program napsaný v cílovém jazyce (obvykle binární kód nebo bytecode), který je **funkčně ekvivalentní** se zdrojovým programem.

Fáze překladu

Jednotlivé části mohou často být spojené. Někdy se provádí více průchodů (definice funkce může být až za jejím voláním, optimalizace, ...)

1. **Lexikální analýza**: Lexikální analyzátor - **scanner**,
2. **Syntaktická analýza**: Syntaktický analyzátor - **parser**,
3. **Sémantická analýza**: Sémantický analyzátor,
4. **Generování vnitřního kódu**: Generátor vnitřního kódu,
5. **Optimalizace**: Optimalizátor,
6. **Generování cílového kódu**: Generátor cílového kódu.

Lexikální analýza



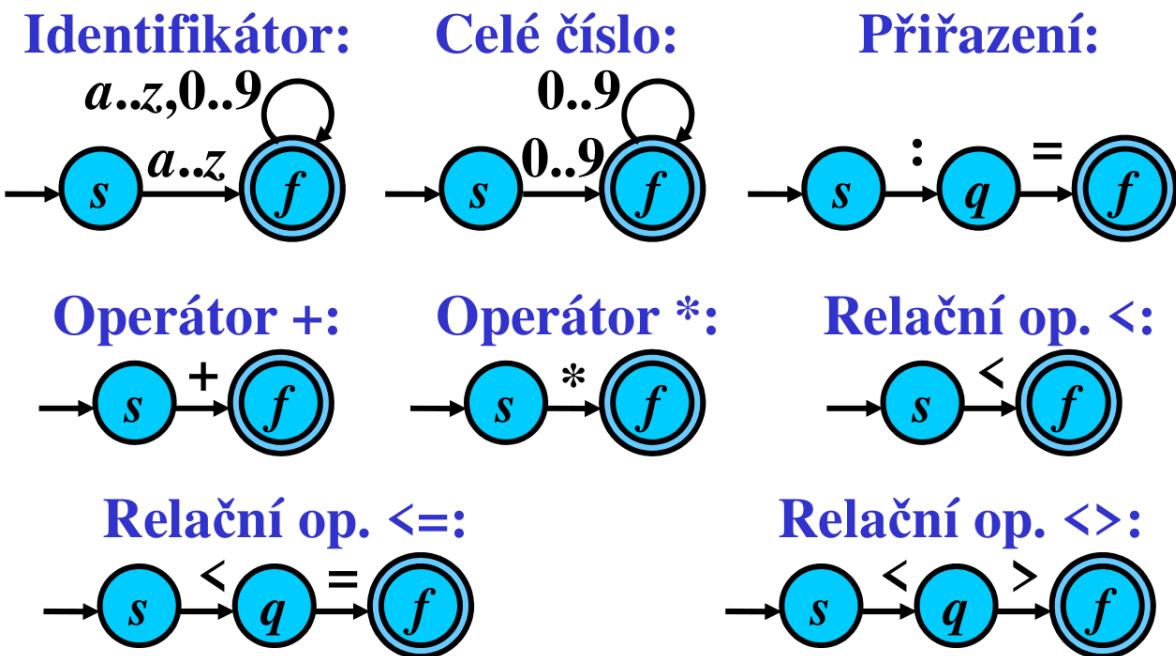
- **Vstup** - Text ve zdrojovém jazyce.
- **Výstup** - Řetězec tokenů.

Provádí **rozpoznávání a klasifikaci lexémů**, reprezentuje lexémy pomocí **tokenů**. Odstraňuje prázdné znaky a komentáře a komunikuje s tabulkou symbolů. Zdrojový program je rozdělen na **lexémy** = logicky oddělené lexikální jednotky (**identifikátory, čísla, klíčová slova, operátory, ...**). Každý lexém je reprezentován **tokenem**, který může mít **atributy** (u čísla je to jeho hodnota, u proměnné její název, u řetězce řetězec, ...). Jednotlivé lexémy jazyků jsou navrženy tak, aby je specifikovaly **regulární výrazy** a bylo je možné přijímat deterministickými konečnými automaty. Na **DKA** je tak založena **implementace** lexikální analýzy - **scanneru**.

Implementujeme pomocí **switch** statement ve **while** cyklu (může být vnořené). Klíčová slova a identifikátory se rozlišují podle **tabulky klíčových slov**. Informace (jméno, typ, konstantní hodnota, počet a typy parametrů v případě funkce, ...) o identifikátorech se uchovávají v **tabulce symbolů**, která má **zásobníkovou strukturu**, což umožňuje např. definovat globální a lokální proměnné.

1) Rozpoznávání jednotlivých lexémů pomocí DKA:

Příklad:

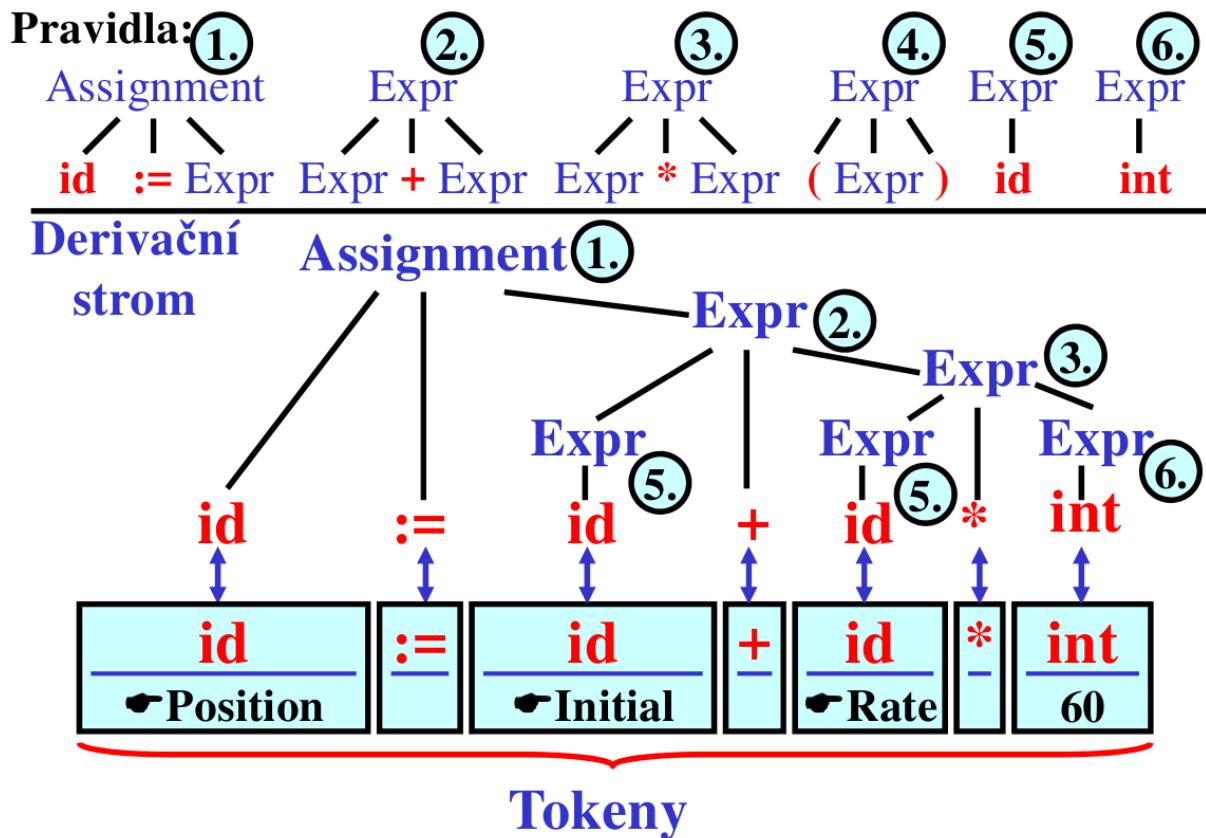


Syntaktická analýza

- **Vstup** - Řetězec tokenů.
- **Výstup** - Simulace konstrukce derivačního stromu.

Syntaktický analyzátor (**parser**) kontroluje, zda **řetězec tokenů** reprezentuje **syntakticky správně** napsaný program. Program je **správný**, pokud je k danému řetězci tokenů nalezen **derivační strom**, jinak správný není. Simulace konstrukce **derivačního stromu** je založena na **gramatických pravidlech**. Používají se dva přístupy, a to **shora dolů** a **zdola nahoru**. Pro syntaktickou analýzu se používají **deterministické zásobníkové automaty** (terminály jsou **tokeny**), respektive podmnožiny **bezkontextových gramatik** - **LL gramatiky** a **LR gramatiky** (BKG jsou silnější než LL a LR gramatiky)

- **první L**: čtení zleva doprava,
- **druhé L**: levá derivace (leftmost derivation) - nahrazuje se **nejlevější neterminál**.
- **druhé R**: pravá derivace (rightmost derivation) - nahrazuje se **nejpravější neterminál**.



Syntaktická analýza shora dolů

Syntaktická analýza shora dolů je založená na **LL gramatikách** a **LL tabulkách**. LL gramatika **bez ϵ** pravidel je BKG $G = (N, T, P, S)$, pro kterou navíc pro každé t, A platí, že $t \in T$, $A \in N$ a existuje maximálně jedno pravidlo $A \rightarrow X_1X_2...X_n \in P$ takové, že: $t \in \text{First}(X_1X_2...X_n)$.

- **First(x)** je množina všech terminálů, kterými může začínat **řetězec derivovatelný** z x , $x \in (N \cup T)^*$.

Konstrukce LL-tabulky

α	...	a	...
...			
A		$\alpha(A, a)$	
...			

Prav. r: $A \rightarrow X_1X_2...X_n \in P$
 $\text{First}(X_1) = \{a\}$ pokud $a \in \text{First}(X_1)$; jinak $\alpha(A, a)$ je prázdné \Rightarrow CHYBA

Vytvořme: LL tabulku

	id	int	:=	...
<prog>				
<st-list>	2. $\text{id} \in \text{First}(\text{<stat>})$			
<stat>		6. $\text{id} \in \text{First}(\text{id})$		
<it-list>				
<item>	10. $\text{id} \in \text{First}(\text{id})$			

Zbytek vytvoříme analogicky.

LL-gramatiky s **ϵ -pravidly** odstraní levé rekurze, ale vyžadují zavést další množiny **Empty**, **Follow** a **Predict**.

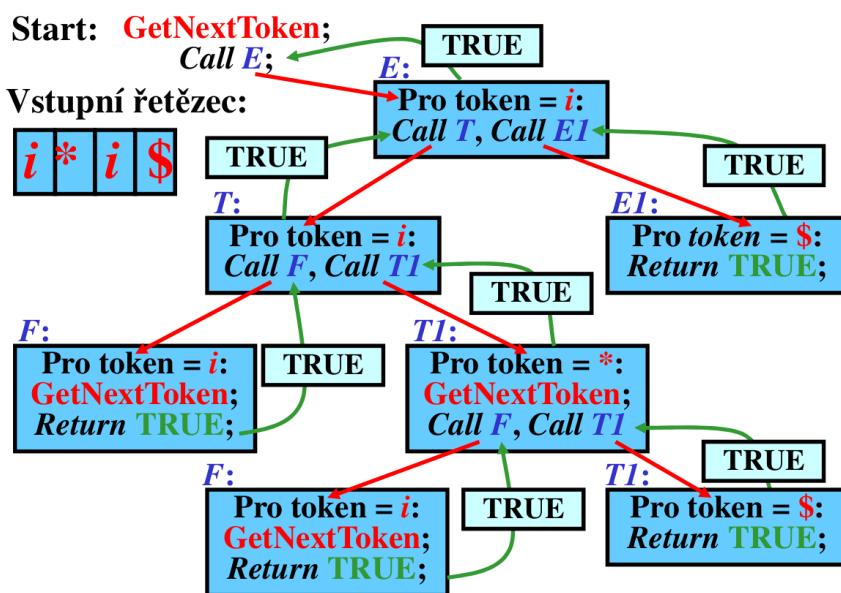
- Empty: **Empty(x)** je množina, která obsahuje jediný prvek ϵ ($\text{Empty}(x) = \{\epsilon\}$), pokud x derivuje ϵ , jinak je prázdná ($\text{Empty}(x) = \emptyset$).
- Follow: **Follow(A)** je množina všech **terminálů**, které se mohou vyskytovat vpravo od A (A je neterminál) ve větné formě.
- Predict: **Predict(A \rightarrow x)** je množina všech **terminálů**, které mohou být aktuálně nejlevěji vygenerovány, pokud pro libovolnou větnou formu použijeme pravidlo $A \rightarrow x$.

	<i>i</i>	+	*	()	\$
<i>E</i>	1			1		
<i>E'</i>		2			3	3
<i>T</i>	4			4		
<i>T'</i>		6	5		6	6
<i>F</i>	8			7		

$$\begin{array}{ll}
 1: E \rightarrow TE' & 5: T' \rightarrow *FT \\
 2: E' \rightarrow +TE' & 6: T' \rightarrow \epsilon \\
 3: E' \rightarrow \epsilon & 7: F \rightarrow (E) \\
 4: T \rightarrow FT' & 8: F \rightarrow i
 \end{array}$$

Implementace LL analyzátoru

- **Rekurzivní sestup:** každý **neterminál** je reprezentován **procedurou/funkcí**, která řídí jeho syntaktickou analýzu a může tak **rekurzivně** volat procedury jiných neterminálů dle pravidel. Např. pro **pravidlo $E \rightarrow TF$** volá procedura pro neterminál **E** nejdříve proceduru pro neterminál **T** a poté (pokud je úspěšná) volá proceduru pro neterminál **F** a když obě uspějí, vrací také úspěch.



- **Prediktivní syntaktická analýza:** Využívá syntaktický analyzátor se **zásobníkem**, který je řízený **LL tabulkou**. **Pravá strana** pravidel se ukládá na zásobník **obráceně - reversal** a provádí se vždy syntaktická analýza

neterminálu na **vrcholu** zásobníku. Např. pro každý neterminál může být definováno pole procedur/funkcí, které provádí pravidlo při načtení symbolu. Pokud pro daný symbol není, dochází k chybě.

Syntaktická analýza zdola nahorů

Provádí Pravý rozbor = reverzovaná posloupnost pravidel, která je použita v **nejpravější derivaci** pro **vstupní řetězec**. Analyzátory pracující zdola nahoru dělíme na **precedenční syntaktické analyzátory** (nejslabší, ale jednoduché na implementaci) a **LR syntaktické analyzátory** (nejsilnější, ale složité pro implementaci, jsou silnější než LL analyzátory, protože jdou zdola - mohou existovat "nedeterministická pravidla").

Precedenční syntaktický analyzátor

- Nesmí existovat více pravidel se **stejnou pravou stranou**.
- Gramatika **nesmí** obsahovat **ϵ -pravidla**.

Pro syntaktickou analýzu využívá **precedenční tabulku**, která je dána **asociativitou** a **precedencí operátorů**. Používá se zejména k SA matematických výrazů (přiřazení do proměnných).

	+	*	()	i	\$	
+	>	<	<	>	<	>	
*	>	>	<	>	<	>	
(<	<	<	=	<		
)	>	>	>	>	>		
i	>	>	>	>	>		
\$	<	<	<	<			

Vstupní řetězec: $i + i * i \$$

Pushdown	Op	Vstup	Rule
\$	<	$i+i*i\$$	
$\$ < i$	>	$+i*i\$$	4: $E \rightarrow i$
$\$ E$	<	$+i*i\$$	
$\$ < E +$	<	$i*i\$$	
$\$ < E + < i$	>	$*i\$$	4: $E \rightarrow i$
$\$ < E + E$	<	$*i\$$	
$\$ < E + < E^*$	<	$i\$$	
$\$ < E + < E^* < i$	>	$\$$	4: $E \rightarrow i$
$\$ < E + < E^* E$	>	$\$$	2: $E \rightarrow E^* E$
$\$ < E + E$	>	$\$$	1: $E \rightarrow E + E$
$\$ E$	>	$\$$	

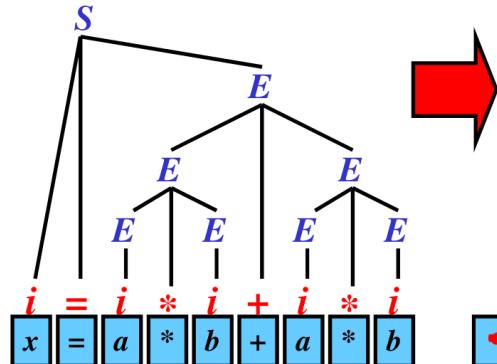
Úspěch
 Pravý rozbor: 44421

Sémantická analýza

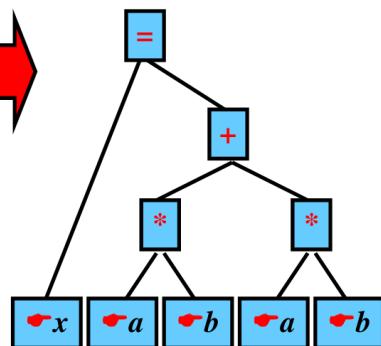
- **Vstup** - Simulace konstrukce derivačního stromu.
- **Výstup** - Abstraktní syntaktický strom.

Kontroluje sémantické aspekty programu, tj. provádí **kontrolu typů** a případně **implicitní konverze** (int na double), kontroluje **deklarace funkcí a proměnných**, kontrola dělení 0, kontrola **nepoužitých proměnných**, kontrola **pravidlosti logických výrazů** atd.

- derivační strom pro
 $x = a^*b + a^*b:$



- ASS pro
 $x = a^*b + a^*b:$



Syntaxí řízený překlad

Syntaktický analyzátor (parser) řídí:

- Provádění sémantických akcí
- Generování abstraktního syntaktického stromu

Generátor vnitřního kódu

- **Vstup** - Abstraktní syntaktický strom.
- **Výstup** - Vnitřní kód.

Generuje **vnitřní kód** - vnitřní reprezentaci programu (většinou **tříadresný**), ten je **jednotný**, lehce se **překládá** do cílového kódu a lehce se **optimalizuje**. Generování vnitřního kódu může být prováděno rekurzivně na základě abstraktního syntaktického stromu. Syntaktický analyzátor, který pracuje metodou **zdola nahorů**, může generovat tříadresný kód **přímo** bez tvorby ASS. Přímé generování 3AK je založeno na **postfixové notaci**.

Tříadresný kód

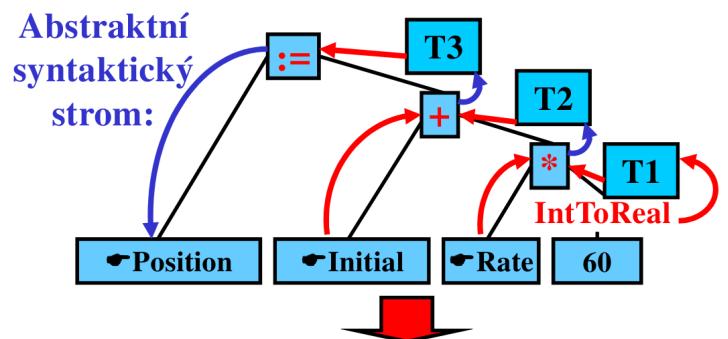
- Instrukce v tříadresném kódu (3AK) má tvar:

(**o**, $\neg a$, $\neg b$, $\neg r$)

- o** – operátor ($+$, $-$, $*$, ...)
- a** – operand 1 ($\neg a$ = adresa **a**)
- b** – operand 2 ($\neg b$ = adresa **b**)
- r** – výsledek ($\neg r$ = adresa **r**)

Příklady:

$(:=, a, , c) \dots c := a$
 $(+, a, b, c) \dots c := a + b$
 $(not, a, , b) \dots b := not(a)$
 $(goto, , , L1) \dots goto L1$
 $(goto, a, , L1) \dots \text{if } a = \text{true} \text{ then goto } L1$
 $(lab, L1, ,) \dots \underline{\text{label }} L1:$



Vnitřní kód:

```

T1 := IntToReal(60)
T2 := -Rate * T1
T3 := -Initial + T2
-Position := T3

```

Optimalizátor

- Vstup** - Vnitřní kód.
- Výstup** - Optimalizovaný vnitřní kód.

Snaží se o optimalizace vnitřního kódu. V rámci **globální** optimalizace odstraňuje mrtvý (nedosažitelný kód) a v rámci **lokální** optimalizace optimalizuje kód v bloku. Lokální optimalizace zahrnují např. **šíření konstanty**, **redukci logických výrazů**, **šíření kopírováním**, **rozbalení cyklu**, **výrazové invarianty v cyklu** (výpočet, který se provádí stejně při každém průchodu cyklem) atd. Je možné optimalizovat na **rychlosť** nebo na **velikost** výsledného programu. Překladač ale optimalizátor mít vůbec **NEMUSÍ** a výsledek programu bude funkčně **stejný**.

Generátor cílového kódu

- Vstup** - Optimalizovaný vnitřní kód (pouze vnitřní kód).
- Výstup** - Cílový program.

Převádí vnitřní kód na cílový program, který je zapsán v cílovém jazyce. Obvykle je to **bytecode**, **assembler** nebo **strojový kód**.

Slepé generování

Pro každou **3AK instrukci** existuje procedura, která generuje příslušný cílový kód.

- výhody**: jednoduché pro implementaci,
- nevýhody**: 3AK instrukce je **mimo kontext** ostatních a může tak docházet k

přebytečným načítáním a ukládáním proměnných do/z registrů.

Kontextové generování

Udržuje si přehled mezi jednotlivými 3AK instrukcemi. Pracuje na principu, jestliže je **hodnota** proměnné **v registru** a bude „**brzy**“ použita, **ponech** ji v registru.

Proměnné se dělí na **živé** (live - budou ještě v bloku použity) a **mrtvé** (dead - již jejich hodnoty nebudou použity, mohou být ale přepsány a poté **dále používány**). U živých proměnných se ještě uchovává **řádek následujícího použití**. Pro označení stavů proměnných (živá/mrtvá) se používá **zpětný algoritmus** 3AK instrukce se čtou od **konce** bloku směrem **k jeho začátku**.

24. Numerické metody (přímé a iterační metody pro řešení soustav lineárních rovnic, numerické řešení obyčejných diferenciálních rovnic).

Matice

Slouží pro zjednodušený zápis soustavy rovnic. Využívají se pro řešení lineárních rovnic.

- **Čtvercová matice** - Stejný počet řádků a sloupců.

$$A = \begin{pmatrix} 0 & 1 & 5 \\ 8 & 5 & 23 \\ 47 & 154 & 2 \end{pmatrix}$$

- **Nulová matice** - Všechny prvky jsou 0

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

- **Jednotková matice** - Čtvercová matice, která má na hlavní diagonále

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

(úhlopříčka z levého horního rohu) jedničky a všude jinde 0.

- **Schodová matice** - Každý následující řádek má na začátku více nul než předchozí řádek.
- **Transponovaná matice** - Zamění se řádky a sloupce (prvek co byl na 1;2 bude na 2;1)
- **Symetrická matice** - Taková, která je stejná i po transponování.

$$A_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & \pi \\ 0 & 0 & 1 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 8 \\ 0 & 0 & 0 & 5 & 1 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 5 \\ 8 & 5 & 23 \\ 47 & 154 & 2 \end{pmatrix}^T = \begin{pmatrix} 0 & 8 & 47 \\ 1 & 5 & 154 \\ 5 & 23 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}^T = \begin{pmatrix} 3 & 6 \\ 4 & 7 \\ 5 & 8 \end{pmatrix}$$

$$A = \begin{pmatrix} 9 & 3 & 4 \\ 3 & 7 & 0 \\ 4 & 0 & 2 \end{pmatrix}$$

- **Antisymetrická matice** - Podobné jako symetrická ale prvky na druhé straně

$$A = \begin{pmatrix} 0 & -3 & -4 \\ 3 & 0 & 5 \\ 4 & -5 & 0 \end{pmatrix}$$

jsou obrácené ($A = -A^T$). Hlavní diagonála tedy musí být 0.

- **Diagonální matice** - Všude jinde než na hlavní diagonále jsou 0 (na hlavní diagonále být můžou ale nemusí).
- **REGULÁRNÍ matice** - Její determinant je nenulový.
- **Diagonálně dominantní matice** - Pokud je absolutní hodnota každého prvku na diagonále větší nebo rovna součtu absolutních hodnot zbylých prvků ve

$$A_1 = \begin{pmatrix} 9 & 0 & 0 \\ 0 & 7 & 0 \\ 0 & 0 & 2 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Je diagonálně dominantní, protože

$$\begin{aligned} |a_{11}| &\geq |a_{12}| + |a_{13}| \quad \text{od té doby } |+3| \geq |-2| + |+1| \\ |a_{22}| &\geq |a_{21}| + |a_{23}| \quad \text{od té doby } |-3| \geq |+1| + |+2| \\ |a_{33}| &\geq |a_{31}| + |a_{32}| \quad \text{od } .|+4| \geq |-1| + |+2| \end{aligned}$$

sloupce nebo řádku (řádkově/sloupcově).

- **Pozitivně definitní matice** - Čtvercová matice, u které platí

$$\mathbf{x} \neq 0 \quad \implies \quad \mathbf{x}^T \mathbf{M} \mathbf{x} > 0$$

Determinant

Udává orientovaný obsah, respektive objem u 3. řádkové matice.

- **Křížové pravidlo** - Slouží pro výpočet determinantu **2x2 matice**.

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \det \mathbf{A} = ad - bc$$

- **Sarrusovo pravidlo** - Slouží pro výpočet determinantu **3x3 matice**.

$$\left| \begin{array}{ccc|cc} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{array} \right|$$

$$\det \mathbf{A} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

- **Determinant NxN matice** - Rozloží se na menší matice. Obsahuje-li matice **nulový řádek**, je její **determinant nulový**. Obsahuje-li matice **dva stejné řádky**, je její **determinant nulový**. Vznikla-li matice B z matice A **výměnou řádků**, pak $|B| = -|A|$. Vznikla-li matice B z matice A **vynásobením** jednoho jejího **řádku** konstantou $c \in \mathbb{R}$, platí $|B| = c|A|$.

Pozor na střídání znamének u kofaktorů:

$$A_{ij} = (-1)^{i+j} M_{ij}$$

Znaménka jsou rozmístěna jako na šachovnici.

a_{11}	a_{12}	a_{13}	\dots
a_{21}	a_{22}	a_{23}	\dots
a_{31}	a_{32}	a_{33}	\dots
\vdots	\vdots	\vdots	\ddots

Vypočtěte determinant matice A pomocí Laplaceova rozvoje.

$$A = \begin{pmatrix} 8 & 3 & 0 & 4 \\ 6 & 7 & 0 & 5 \\ -1 & 0 & 3 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix}$$

Nejvhodnější je použít rozvoj podle třetího sloupce:

$$\begin{aligned} |A| &= 0 \cdot (-1)^{1+3} \begin{vmatrix} 6 & 7 & 5 \\ -1 & 0 & 0 \\ 0 & 2 & 1 \end{vmatrix} + 0 \cdot (-1)^{2+3} \begin{vmatrix} 8 & 3 & 4 \\ -1 & 0 & 0 \\ 0 & 2 & 1 \end{vmatrix} + \\ &+ 3 \cdot (-1)^{3+3} \begin{vmatrix} 8 & 3 & 4 \\ 6 & 7 & 5 \\ 0 & 2 & 1 \end{vmatrix} + 0 \cdot (-1)^{4+3} \begin{vmatrix} 8 & 3 & 4 \\ 6 & 7 & 5 \\ -1 & 0 & 0 \end{vmatrix} = 3 \cdot 6 = 18 \end{aligned}$$

Přímé metody pro řešení algebraických rovnic

Vedou k řešení soustavy po **konečném počtu kroků**. Toto řešení by bylo **přesné** kdybychom se nedopouštěli zaokrouhlovacích chyb.

Cramerovo pravidlo

$$x_1 = \frac{D_1}{D}, \quad x_2 = \frac{D_2}{D}, \quad x_n = \frac{D_n}{D},$$

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 1 \\ -2x_1 + x_2 - 3x_3 &= 2 \\ 2x_2 - x_3 &= -2 \end{aligned}$$

$$D = \begin{vmatrix} 1 & 2 & -1 \\ -2 & 1 & -3 \\ 0 & 2 & -1 \end{vmatrix} = -1 + 4 + 6 - 4 = 5$$

Úkolem je řešit soustavu rovnic

$$x + y = 3$$

$$x - 2y = 1$$

Determinant matice soustavy je

$$\det \mathbf{A} = \begin{vmatrix} 1 & 1 \\ 1 & -2 \end{vmatrix} = -3$$

Poněvadž je $\det \mathbf{A} \neq 0$, lze použít Cramerovo pravidlo.

Dále určíme

$$\det \mathbf{A}_1 = \begin{vmatrix} 3 & 1 \\ 1 & -2 \end{vmatrix} = -7$$

$$\det \mathbf{A}_2 = \begin{vmatrix} 1 & 3 \\ 1 & 1 \end{vmatrix} = -2$$

$$D_1 = \begin{vmatrix} 1 & 2 & -1 \\ 2 & 1 & -3 \\ -2 & 2 & -1 \end{vmatrix} = -1 - 4 + 12 - 2 + 6 + 4 = 15$$

$$D_2 = \begin{vmatrix} 1 & 1 & -1 \\ -2 & 2 & -3 \\ 0 & -2 & -1 \end{vmatrix} = -2 - 4 - 6 - 2 = -14$$

$$D_3 = \begin{vmatrix} 1 & 2 & 1 \\ -2 & 1 & 2 \\ 0 & 2 & -2 \end{vmatrix} = -2 - 4 - 4 - 8 = -18$$

Řešení má tedy tvar

$$x = \frac{\det \mathbf{A}_1}{\det \mathbf{A}} = \frac{-7}{-3} = \frac{7}{3}$$

$$y = \frac{\det \mathbf{A}_2}{\det \mathbf{A}} = \frac{-2}{-3} = \frac{2}{3}$$

Vhodné pro velmi malé soustavy rovnic. Je-li matice soustavy **regulární ($D \neq 0$)**. Při výpočtu i-tého determinantu vždy nahrazujeme i-tý sloupec sloupcem s výsledky.

Gaussova eliminační metoda

Základem je úprava matice soustavy na **schodovitý tvar** - prohazování řádků, násobením a dělením nenulovým číslem a **přičítáním/odečítáním násobků jednotlivých řádků** k jiným. Pomineme-li zaokrouhlovací chyby, metoda poskytuje

přesný výsledek, ale je poměrně náročná pro výpočet, konkrétně je třeba provést **n³/3** (složitost v počtu provedených aritmetických operací je tedy **kubická**) aritmetických operací.

$$\left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 3 & 8 & 1 & 50 \\ 0 & 3 & 3 & 27 \end{array} \right) \xrightarrow{\text{x3}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 5 & 1 & 50 \\ 0 & 3 & 3 & 27 \end{array} \right) \xrightarrow{\text{x2}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 5 & 1 & 50 \\ 0 & 0 & 3 & 27 \end{array} \right) \xrightarrow{\text{-2}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 5 & 1 & 50 \\ 0 & 0 & 3 & 27 \end{array} \right) \xrightarrow{\text{x3}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 5 & 1 & 50 \\ 0 & 0 & 1 & 9 \end{array} \right) \xrightarrow{\text{x5}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 1 & 1 & 10 \\ 0 & 0 & 1 & 9 \end{array} \right) \xrightarrow{\text{-3}} \left(\begin{array}{ccc|c} 2 & 3 & 7 & 47 \\ 0 & 1 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{array} \right)$$

||

$$\begin{aligned} 6x_1 + 9x_2 + 21x_3 &= 141 \\ 21x_2 - 57x_3 &= -123 \\ 78x_3 &= 312 \end{aligned} \rightarrow$$

$$\begin{aligned} x_3 &= 4 \\ x_2 &= 5 \\ x_1 &= 2 \end{aligned}$$

LU rozklad

[LU decomposition - An Example](#)

Pomocí nalezeného LU rozkladu matice A najděte řešení soustavy

$$\begin{aligned} 2x_1 - 3x_2 + x_3 &= 5 \\ -3x_1 + 5x_2 + 2x_3 &= -4 \\ x_1 + 2x_2 - x_3 &= 1 \end{aligned}$$

Při Lower-Upper rozkladu se začne s jednotkovou maticí, která násobí původní

$$L = \begin{pmatrix} 1 & 0 & 0 \\ -1,5 & 1 & 0 \\ 0,5 & 7 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & -3 & 1 \\ 0 & 0,5 & 3,5 \\ 0 & 0 & -26 \end{pmatrix}$$

matici, provádí se úpravy jako při GEM a zapisují se do jednotkové matice viz příklad (pozor na **znaménka**)

$$\begin{pmatrix} 2 & -3 & 1 \\ -3 & 5 & 2 \\ 1 & 2 & -1 \end{pmatrix} \sim \begin{pmatrix} 2 & -3 & 1 \\ 0 & 0,5 & 3,5 \\ 0 & 3,5 & -1,5 \end{pmatrix} \text{ //} +1,5I \quad L = \begin{pmatrix} 1 & ? & ? \\ -1,5 & ? & ? \\ +0,5 & ? & ? \end{pmatrix}$$

$$\sim \begin{pmatrix} 2 & -3 & 1 \\ 0 & 0,5 & 3,5 \\ 0 & 0 & -26 \end{pmatrix} \text{ //} -7II \quad L = \begin{pmatrix} 1 & 0 & ? \\ -1,5 & 1 & ? \\ +0,5 & +7 & ? \end{pmatrix}$$

Následně je nutné pomocí matice **L** upravit vektor pravé strany rovnice a na základě tohoto vektoru vypočítat řešení rovnice pomocí matice **U**.

$$\begin{array}{lcl} y_1 & = & 5 \\ -1,5y_1 + y_2 & = & -4 \\ 0,5y_1 + 7y_2 + y_3 & = & 1 \end{array} \Rightarrow \begin{array}{lcl} y_1 & = & 5 \\ y_2 & = & 3,5 \\ y_3 & = & -26 \end{array}$$

$$\begin{array}{lcl} 2x_1 - 3x_2 + x_3 & = & 5 \\ 0,5x_2 + 3,5x_3 & = & 3,5 \\ -26x_3 & = & -26 \end{array} \Rightarrow \begin{array}{lcl} x_1 & = & 2 \\ x_2 & = & 0 \\ x_3 & = & 1 \end{array}$$

Iterační metody

Narozdíl od přímých **nevedou k přesnému výsledku** po konečném předem daném počtu kroků. Zvolíme počáteční approximaci řešení a tu v každém kroku **zlepšujeme**. K řešení se **přibližujeme postupně** a až je dostatečně přesný výpočet ukončíme (**výsledek** je tedy **přibližný**, k přesnému bychom se dostali s nekonečným počtem operací).

Jacobiho metoda

[The Jacobi Method](#)

Konverguje pokud je matice soustavy rovnic **ostře řádkově diagonálně dominantní** (součet absolutních hodnot řádku je **větší**, než hodnota na diagonále) nebo **ostře sloupcově diagonálně dominantní** (součet absolutních hodnot sloupce je **větší**, než hodnota na diagonále). Pokud podmínky nejsou splněny, může konvergovat, ale nemusí.

$$\begin{aligned}
 15x_1 - x_2 + 2x_3 &= 30 \\
 2x_1 - 10x_2 + x_3 &= 23 \\
 x_1 + 3x_2 + 18x_3 &= -22
 \end{aligned}$$

- Matice soustavy je diagonálně dominantní, protože platí:
 $|15| > |1| + |2|, |10| > |2| + |1|, |18| > |1| + |3|$

- Proto je konvergence metody vždy zaručena. Vypíšeme iterační vztahy:

$$\begin{aligned}
 x_1^{(r+1)} &= \frac{1}{15}(30 + x_2^{(r)} - 2x_3^{(r)}) \\
 x_2^{(r+1)} &= -\frac{1}{10}(23 - 2x_1^{(r)} - x_3^{(r)}) \\
 x_3^{(r+1)} &= \frac{1}{18}(-22 - x_1^{(r)} - 3x_2^{(r)})
 \end{aligned}$$

- Jako počáteční approximaci zvolíme $x = (0,0,0)^T$. Postupné získávané approximace řešení budeme zapisovat do tabulky:

r	$x_1^{(r)}$	$x_2^{(r)}$	$x_3^{(r)}$
0	0	0	0
1	2	-2,3	-1,2222
2	2,0096	-2,0222	-0,9500
3	1,9918	-1,9930	-0,9968
4	2,0000	-2,0013	-1,0007

Gauss-Seidelova metoda

Metoda je podobná Jacobiho metodě. Liší se v tom, že v každém kroku používá již **novou hodnotu proměnné** (pokud je známa) a ne tu z minulé iterace. Konverguje pro **ostře řádkově nebo sloupcově dominantní matice** a navíc nebo pokud je matice **pozitivně definitní** (lze zjistit např., že všechny její subdeterminanty z levého horního rohu musí být **větší než 0** [Checking if a Matrix is Positive Definite](#) nebo lze zjistit **pivot testem** - převodem matice na horní trojúhelníkovou matici a porovnáním hodnot na diagonále s 0, pokud jsou **větší než 0**, je matice pozitivně definitní [Positive Definite Matrices and Minima](#) (pro větší nebo rovno 0 jsou matice semidefinitní)).

$$\begin{aligned}
 x_1^{(r+1)} &= \frac{a_{14} - a_{12}x_2^{(r)} - a_{13}x_3^{(r)}}{a_{11}} \\
 x_2^{(r+1)} &= \frac{a_{24} - a_{21}x_1^{(r+1)} - a_{23}x_3^{(r)}}{a_{22}} \\
 x_3^{(r+1)} &= \frac{a_{34} - a_{31}x_1^{(r+1)} - a_{32}x_2^{(r+1)}}{a_{33}}
 \end{aligned}$$

$$\begin{aligned}x_1^{k+1} &= \frac{1}{2} \left(x_2^k + \frac{1}{3} \right) \\x_2^{k+1} &= \frac{1}{2} \left(x_1^{k+1} + x_3^k + 1 \right) \\x_3^{k+1} &= \frac{1}{2} \left(x_2^{k+1} - \frac{1}{3} \right)\end{aligned}$$

Průběh výpočtu je zaznamenán v následující tabulce.

k	x_1^k	x_2^k	x_3^k
0	0	0	0
1	0.1667	0.5833	0.1250
2	0.4583	0.7917	0.2290
:	:	:	:
14	0.6666	1.0000	0.3333
15	0.6666	1.0000	0.3333

Numerické řešení obyčejných diferenciálních rovnic

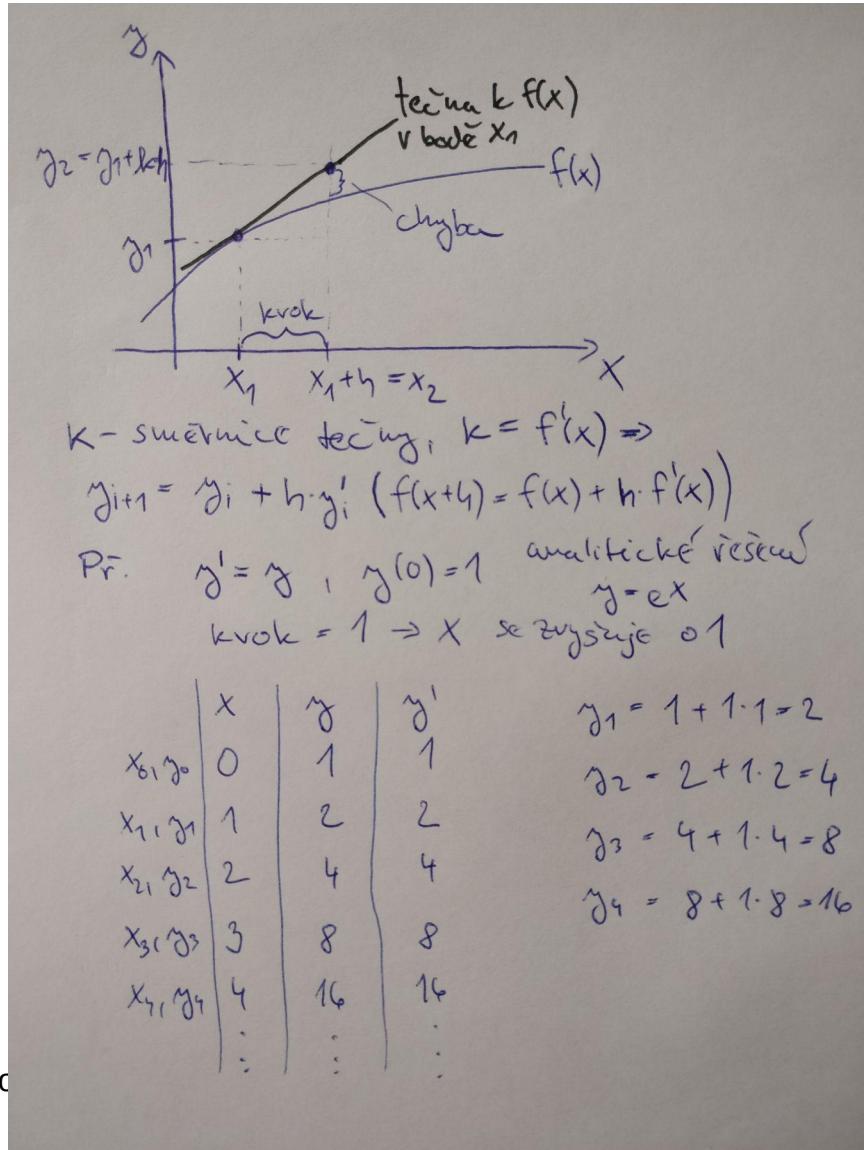
Rovnice, kde jako proměnné vystupují **derivace funkcí**. U některých lze nalézt přesné analytické řešení, ale většinou **nelze nebo je to velmi obtížně**. Naštěstí lze řešení diferenciální rovnice velmi **dobře approximovat** za použitím některé **numerické metody**, které jsou založené na **iteračním řešení** těchto rovnic.

Numerické metody dělíme na:

- **jednokrokové**: vychází pouze z **aktuálního stavu** (aktuální ohodnocení proměnných), např. **Eulerova metoda**, metody **Runge-Kutta (RK2, RK4, RK8)**.
- **vícekrokové**: využívají historii stavů (ohodnocení proměnných), používají **hodnoty zapamatované z předchozích kroků**. Mohou být rychlejší než jednokrokové, ale obvykle mají **problém se startem** (pro prvních n iterací se použije **jednokroková** metoda → nevhodné pro nespojitě funkce). Jedná se např. o **metodu Adams-Bashforth**. (existují i samostartující metody)
- **prediktor-korektor**: Nejprve se vypočítá odhad nového **y_n+1**. V tomto bodě je vypočtena derivace **f_n+1**, která je následně použita pro výpočet přesnější approximace **y_n+1**.
- **explicitní**: výsledek v každé iteraci získáme **dosazením do vzorce**.
- **implicitní**: vyžaduje řešení **algebraických rovnic** v každé iteraci.

Eulerova metoda

$$y' = f(x, y), \quad y(x_0) = y_0 \quad y_{i+1} = y_i + h \cdot f(x_i, y_i), \quad i = 0, 1, 2, \dots$$



směrnice tečny v bodě, která je daná **1. derivací** funkce v tomto bodě. Směrnice udává, o kolik se **zvětší/zmenší** hodnota y při změně x (definuje tak přímku).

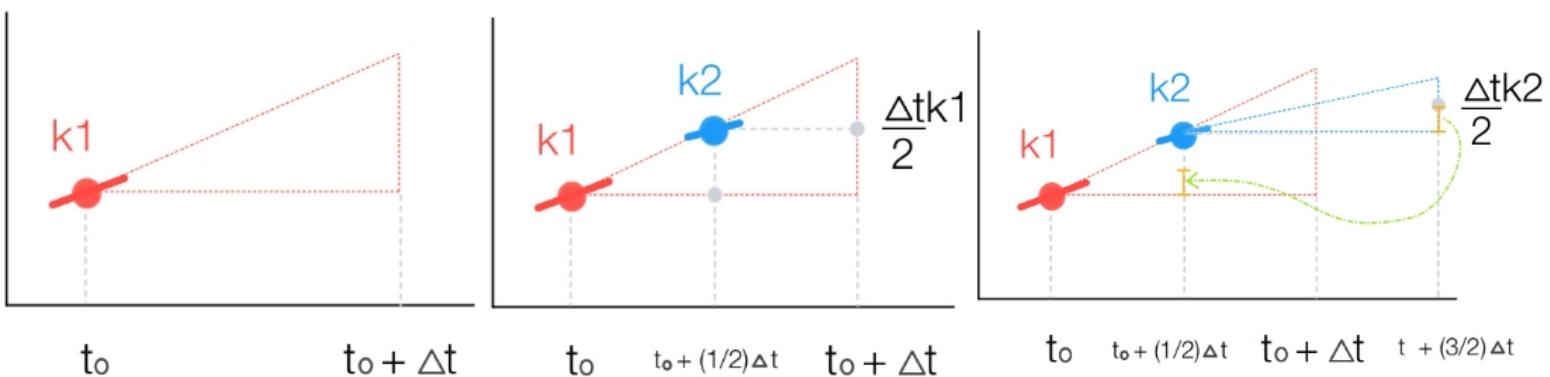
Př. 2: $y' = x - y; \quad y(0) = 1; \quad h=0,2$ na $<0; 0,6>$

$$y_1 = y_0 + h \cdot f(x_0, y_0) = y_0 + h \cdot (x_0 - y_0) = 1 + 0,2 \cdot (0 - 1) = \underline{\underline{0,8}}$$

$$y_2 = y_1 + h \cdot f(x_1, y_1) = y_1 + h \cdot (x_1 - y_1) = 0,8 + 0,2 \cdot (0,2 - 0,8) = \underline{\underline{0,68}}$$

$$y_3 = y_2 + h \cdot f(x_2, y_2) = y_2 + h \cdot (x_2 - y_2) = 0,68 + 0,2 \cdot (0,4 - 0,68) = \underline{\underline{0,624}}$$

Runge-Kutta



Jednokroková metoda. Vylepšuje eulerovu metodu. Existují RK metody různých řádů (RK1 - eulerova metoda, RK2, RK4, RK8). Koeficienty u téchto metod jsou vypočteny tak, aby metoda řádu **b** odpovídala **Taylorovu polynomu** funkce $y(t)$ stejného řádu. Pro výpočty se nejčastěji používá metoda **Runge-Kutta 4. řádu**. Pro RK metodu **k-tého** řádu se počítá **k** částečných odhadů (druhý odhad je závislý na prvním, třetí na druhém, ...) a výsledný odhad se poté vypočte jako krok vynásobený jejich váženým průměrem.

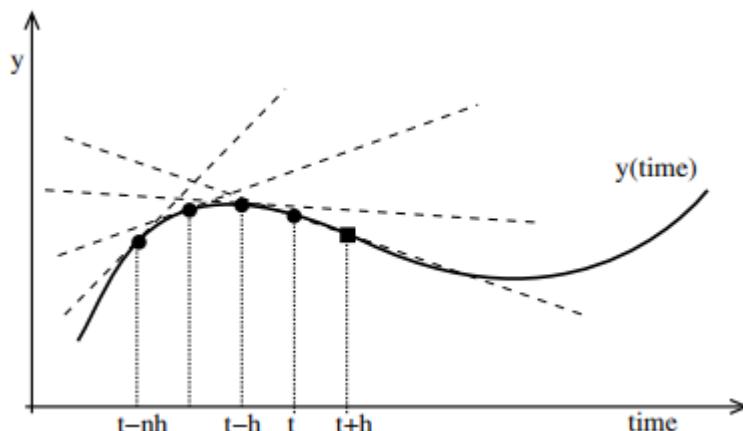
$$y_{n+1} = y_n + h \left(\frac{1}{2} k_1 + \frac{1}{2} k_2 \right) \quad \begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + h, y_n + hk_1). \end{aligned}$$

Adams-Bashforth

$$y_{n+2} = y_{n+1} + \frac{3}{2}hf(t_{n+1}, y_{n+1}) - \frac{1}{2}hf(t_n, y_n).$$

$$y_{n+1} = y_n + \frac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})$$

Vícekroková metoda, pamatuje si výsledky předchozích kroků. Např. dvojkrokový a čtyřkrokový:



Adams-Bashforth-Moulton

Metoda typu **prediktor-korektor**, zpřesňují výsledek použitím prvotního odhadu pro výpočet výsledného odhadu.

$$y_{n+1} = y_n + \frac{h}{24}(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2})$$

Tuhé systémy

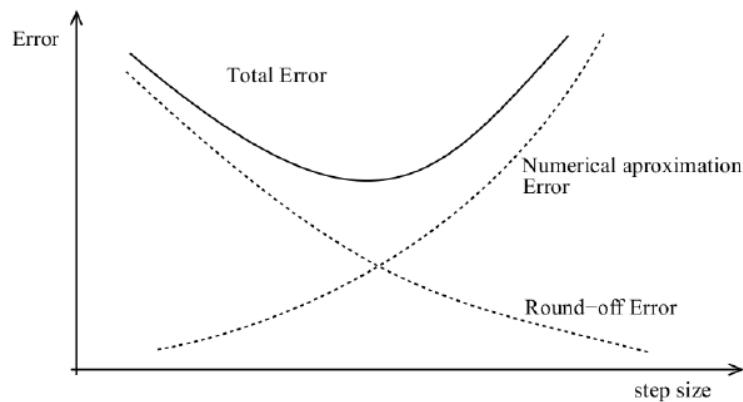
Problematické pro řešení pomocí běžných numerických metod (RK). Vyskytuje se zde velmi rozdílné časové konstanty. Zkrácení kroku často nelze (zaokrouhlovací chyby, malá efektivita). Je nutné použít speciální metody.

$$y'' + 101y' + 100y = 0$$

Chyby numerických metod

Při každé aproximaci se musí počítat s chybou výsledku.

- **Lokální chyba** - Vzniká v každém kroku - **zaokrouhlovací nebo approximační**.
- **Akumulované chyby** - Sesbírané chyby po **celou dobu výpočtu**.



Hledání ideální délky kroku.

25. Teorie grafů. Pojem grafu, základní pojmy, isomorfismus grafů, souvislost. Grafové algoritmy pro hledání nejkratší cesty a minimální kostry.

Neformálně se graf skládá z vrcholů a hran, které tyto vrcholy spojují. Jedná se o speciální případy binárních relací. **Formálně** je graf (jednoduchy neorient.) uspořádaná dvojice $\mathbf{G} = (\mathbf{V}, \mathbf{E})$

- \mathbf{V} je množina vrcholů,
- \mathbf{E} je množina hran – množina vybraných **dvouprvkových podmnožin množiny vrcholů** (znázorňují spojení mezi vrcholy).

Hranu mezi vrcholy u a v píšeme jako $\{u, v\}$, nebo zkráceně uv .

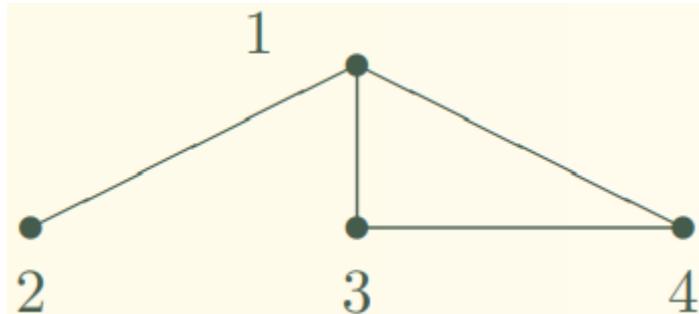
- **sousední vrcholy:** vrcholy spojené hranou.

Hrana uv vychází z vrcholů u a v .

Množiny grafu G odkazujeme:

- $\mathbf{V}(G)$: množina **vrcholů**,
- $\mathbf{E}(G)$: množina **hran**.

Grafy zadáváme **neformálně** graficky



$$V = \{1, 2, 3, 4\}, \quad E = \left\{ \{1, 2\}, \{1, 3\}, \{1, 4\}, \{3, 4\} \right\}$$

nebo formálně výčtem vrcholů a hran

Stupeň vrcholu

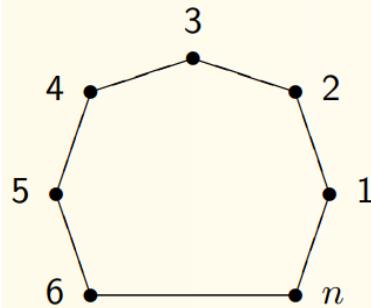
Stupeň vrcholu x v grafu G rozumíme počet hran vycházejících (hrna vychází z obou konců současně) z vrcholu x . Stupeň vrcholu x v grafu G značíme $d_G(x)$. Graf je:

- **d -regulární:** všechny jeho vrcholy mají stejný stupeň d .

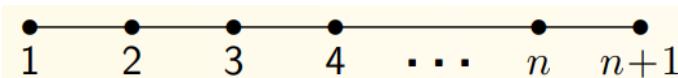
Nejvyšší stupeň grafu G značíme $\Delta(G)$. **Nejnižší stupeň** grafu G značíme $\delta(G)$.

Typy grafů

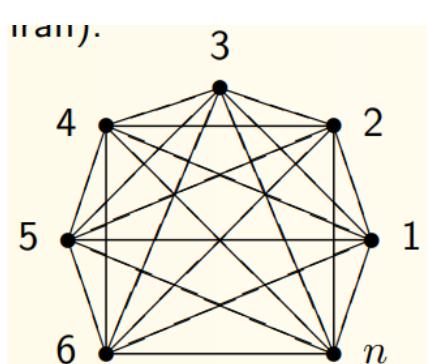
- **Kružnice:** Kružnice délky n má $n \geq 3$ různých vrcholů spojených do jednoho cyklu n hranami. Značí se jako C_n .



- **Cesta:** Cesta délky $n \geq 0$ má $n+1$ různých vrcholů spojených za sebou n hranami. Značí se jako P_n . (žádné vrcholy ani hrany se neopakují)

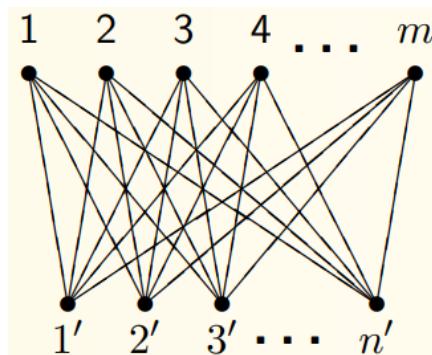


- **Úplný graf:** Úplný graf na $n \geq 1$ vrcholech má n různých vrcholů spojených po všech dvojicích - celkem n nad 2 hran. Značí se jako K_n .

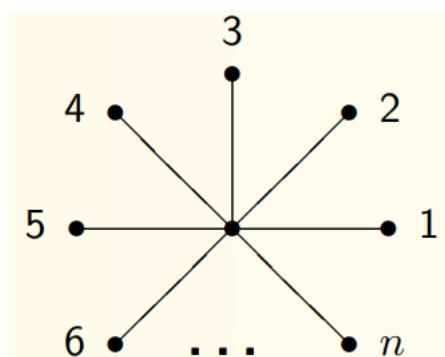


$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

- **Úplný bipartitní graf:** Úplný bipartitní graf na $m \geq 1$ a $n \geq 1$ vrcholech má $m+n$ vrcholů ve dvou skupinách (partitách), přičemž hranami jsou spojeny všechny $m \cdot n$ dvojice z různých skupin. Značí se jako $K_{m,n}$.



- **Hvězda:** Hvězda s $n \geq 1$ rameny je zvláštní název pro **úplný bipartitní graf**.

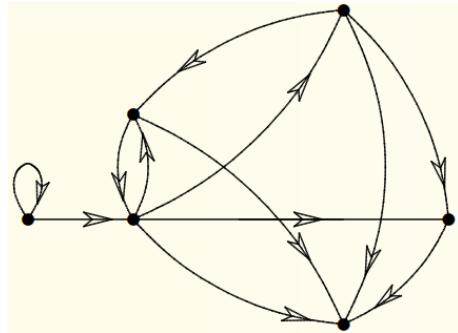


Značí se jako $K_{1,n}$.

- **Sled:** je v grafu **posloupnost vrcholů** taková, že mezi každými dvěma po sobě jdoucími vrcholy je hrana. **Hrany i vrcholy** se mohou na rozdíl od kružnice **opakovat**. Sled je procházka po hranách grafu z u do v , která může obsahovat cykly.
- **Tah:** je sled v grafu, ve kterém se **neopakují hrany** (vrcholy se opakovat mohou). **Uzavřený tah** je tah, který končí ve vrcholu, ve kterém **začal** (jinak je neuzavřený). Graf G lze nakreslit **jedním uzavřeným tahem právě**, když G je **souvislý** a všechny jeho vrcholy jsou **sudého stupně**.

Orientované grafy

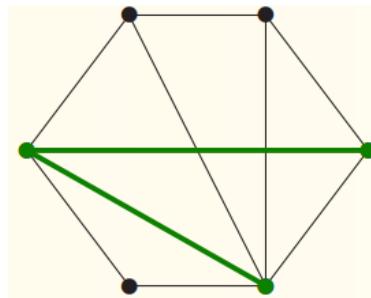
V orientovaných grafech má každá hrana jistý směr. Formálně mají orientované grafy množinu orientovaných hran A , která je dána $A \subseteq V(G) \times V(G)$.



Podgraf

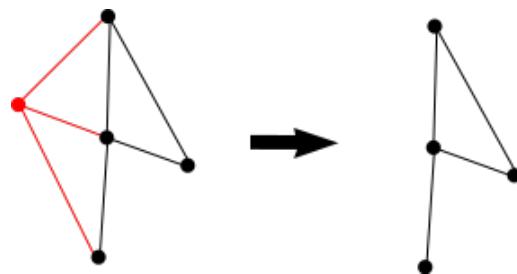
Podgrafram grafu **G** rozumíme libovolný graf **H** pro který platí:

- množina vrcholů grafu **H** je podmnožinou vrcholů grafu **G**: $V(H) \subseteq V(G)$.
- za hrany má **libovolnou podmnožinu** hran grafu **G**, které ale mají oba vrcholy ve $V(H)$.



Indukovaný podgraf

Indukovaným podgrafram je podgraf $H \subseteq G$ takový, který obsahuje **všechny hrany grafu G** mezi dvojicemi vrcholů z $V(H)$. Jinak řečeno graf **H** vznikne smazáním části vrcholů grafu **G** a **pouze** hran, které vycházely z těchto vrcholů.



Izomorfismus

Izomorfismus grafů **G** a **H** je **bijektivní** zobrazení $f: V(G) \rightarrow V(H)$, pro které každá dvojice $u, v \in V(G)$ je spojená hranou v grafu **G** právě, když je dvojice $f(u), f(v) \in V(H)$ spojená hranou v grafu **H**. Příklad pro grafy **G** a **G'**:

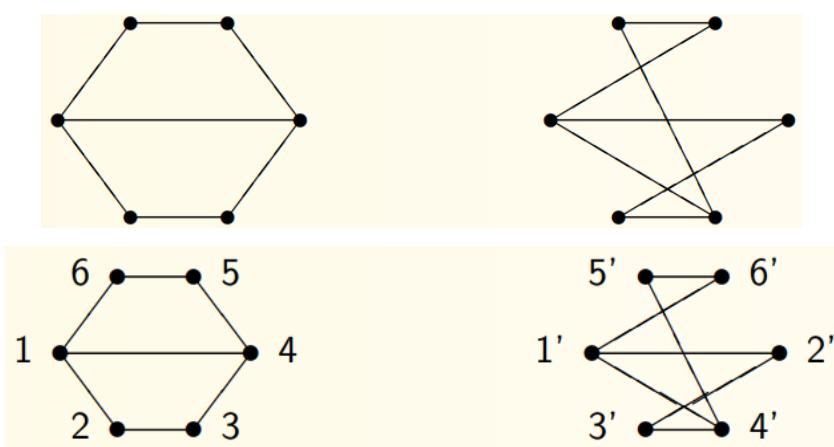
$$\exists F: V(G) \rightarrow V(G'): \{x, y\} \in E(G) \Leftrightarrow \{f(x), f(y)\} \in E(G')$$

Pro izomorfní grafy **G** a **H** platí (může to platit ale i pro neizomorfní grafy):

- **G** a **H** mají stejný počet vrcholů,
- **G** a **H** mají stejný počet hran,
- **zobrazení f** zobrazuje na sebe vrcholy **stejných stupňů**, tzn. mají stejné počty vrcholů o stejných stupních.

Postup hledání izomorfismu (pokud nějaký bod neplatí, grafy nejsou izomorfní):

1. ověříme **stejný počet vrcholů** u obou grafů,
2. ověříme **stejný počet hran** u obou grafů,
3. vytvoříme posloupnosti stupňů vrcholů pro každý graf (seřazeny od nejmenšího po největší) a ověříme, že jsou stejné.



4. zkoušíme všechny **přípustné možnosti** zobrazení izomorfismu.

Typy podgrafů grafu G

- **Kružnice v G**, je podgraf $H \subseteq G$, který je **izomorfní** nějaké **kružnici**. (kružnici délky 3 říkáme trojúhelník).
- **Indukovaná kružnice v G**, je **indukovaný** podgraf $H \subseteq G$, který je **izomorfní** nějaké **kružnici**.
- **Cesta v G**, je podgraf $H \subseteq G$, který je **izomorfní** nějaké **cestě**.
- **Klika v G**, je podgraf $H \subseteq G$, který je **izomorfní** nějakému **úplnému grafu** (graf jehož všechny vrcholy jsou spojeny hranou se všemi zbylými).
- **Nezávislá množina X v G**, je podmnožina vrcholů $X \subseteq V(G)$, mezi kterými nevedou v **G** žádné **hrany** (přímo tyto vrcholy spojuje hrana, spojení přes

více hran existovat může).

Souvislost

Možnost se v grafu pohybovat z jakéhokoliv vrcholu do jakéhokoliv jiného vrcholu podél jeho hran. To znamená, že pro každé dva vrcholy $u, v \in V(G)$ existuje sled z vrcholu u do vrcholu v . U orientovaných grafů rozlišujeme:

- **slabá souvislost** — graf je slabě souvislý, pokud jeho symetrizace (odstranění směru hran) je souvislý graf.
- **silná souvislost** — graf je silně souvislý, pokud pro každé dva vrcholy u, v existuje cesta z u do v i z v do u .

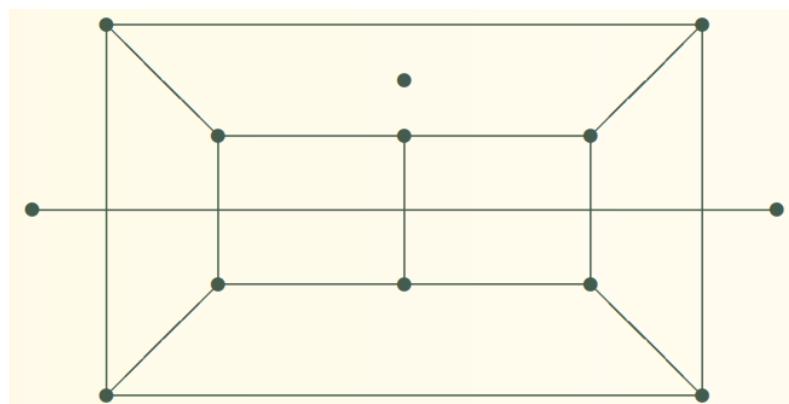
Relace \sim na množině vrcholů $V(G)$ libovolného grafu G , je definována tak, že $u, v \in V(G)$ jsou v relaci $u \sim v$, právě když v grafu G existuje **sled** začínající v u a končící ve v . Tato relace je:

- **reflexivní**: každý vrchol je spojen sám se sebou sledem délky 0.
- **symetrická**: sled z u do v lze obrátit na sled z v do u vždy u neorientovaného grafu.
- **tranzitivní**: dva sledy na sebe můžeme vždy **navázat v jeden**.

Relace \sim je tedy relací **ekvivalence**.

Komponenty souvislosti

Jsou jednotlivé **třídy ekvivalence** grafu (graf je **souvislý**, pokud má **pouze jednu** komponentu souvislosti). Graf o střech komponentách:



Stromy

Strom je jednoduchý **souvislý graf T bez kružnic**. Grafy bez kružnic lze také nazývat **acyklické**. **Les** je nesouvislý graf tvořený více stromy. Stromové grafy jsou zároveň kostrou grafu. Pro stromy platí:

- pokud mají více než jeden vrchol, existuje vrchol se **stupněm 1**,
- mezi každými dvěma vrcholy vede **právě** jedna cesta.

Grafové algoritmy

Využívají nějakou paměť - zásobník, frontu, seřazené pole.

Prohledávání do šířky - BFS

Algoritmus postupně prochází celé úrovně od počátečního vrcholu. Jako paměť využívá **frontu**. Pracuje následovně:

1. První vrchol se vloží do fronty,
2. Pokud není fronta prázdná, zpracuj vrchol na **začátku fronty**. Zpracování znamená umístění vrcholů, do kterých vede hrana ze zpracovávaného vrcholu, **do fronty**.

Algoritmus BFS lze použít pro zjištění nejkratší **vzdálenosti** mezi dvěma vrcholy spojeného **neváženého** grafu.

Prohledávání do hloubky - DFS

Algoritmus prochází nejprve do co nejvzdálenější úrovně a poté se postupně vynořuje a zase co nejvíce zanořuje. Jako paměť využívá zásobník a pracuje následovně:

1. První vrchol se vloží na zásobník,
2. Pokud není zásobník prázdný, zpracuje se vrchol na **vrcholu zásobníku**. Zpracování znamená umístění vrcholů, do kterých vede hrana ze zpracovávaného vrcholu, na **zásobník**.

Dijkstrův algoritmus

Algoritmus pro hledání **nejkratší cesty** (vzdálenosti) mezi dvěma **vrcholy u a v v kladně váženém grafu**. Jako úložiště používá **prioritní frontu**. Může pracovat s časovou komplexitou $O((hrany+vrcholy)*\log(vrcholy))$ nebo $O(vrcholy^2)$ při použití pole. Princip algoritmu:

1. Všechny vrcholy až na počáteční jsou ohodnoceny **nekonečnem** (neznáme do nich cestu), počáteční vrchol je ohodnocen **0** (cesta do tohoto vrcholu je nulová). Všechny vrcholy jsou označeny za nezpracované.
2. Z nezpracovaných vrcholů vybereme ten s **nejmenší hodnotou** (na začátku prioritní fronty - při 1. iteraci to bude počáteční vrchol - vrchol **u**). Pokud je tento vrchol hledaným vrcholem (vrchol **v**), ukončíme algoritmus, jinak **přepíšeme** vzdálenosti všech vrcholů (již zpracované ignorujeme), do kterých se můžeme **z vybraného vrcholu dostat hranou, přičtením ohodnocení této hrany k hodnotě zpracovávaného vrcholu**, pokud je tato hodnota **MENŠÍ** než aktuální ohodnocení. V tomto případě si také **zaznamenáme**, přes **jaký vrchol** vede tato nejkratší cesta.
3. Aktuálně zpracovávaný uzel označíme za zpracovaný a pokud jsou ještě nezpracované vrcholy, pokračujeme 2. bodem.
4. Zpětně **rekonstruujeme cestu** z cílového vrcholu k startovacímu vrcholu na základě **zapamatovaných** údajů v bodě 2.

[Dijkstra's Algorithm - Computerphile](#)

Jarníkův (Primův) algoritmus

Algoritmus, který hledá **minimální kostru ve váženém grafu** (stromový podgraf, který propojuje všechny vrcholy) - **minimum spanning tree**. Jako úložiště může použít např. prioritní frontu pro nenavštívené. Princip algoritmu:

1. Do úložiště ulož všechny vrcholy s ohodnocením **nekonečno** (ještě nejsou v kostře)
2. Vyber jakýkoliv uzel grafu, změň jeho ohodnocení na **0** a odstraň jej z úložiště.
3. U vrcholů (musí být v úložišti, tj. **ještě nenavštívené**), do kterých se lze z tohoto vrcholu dostat **aktualizuj** jejich **ohodnocení** hodnotou cesty k tomuto vrcholu, pokud je tato hodnota **menší než aktuální**, a zapiš vrchol, ze kterého **tato cesta vede**.
4. Vyber vrchol z úložiště, který má **nejmenší ohodnocení**, a odstraň jej z úložiště.
5. Pokud úložiště **není prázdné**, pokračuj s bodem 3.
6. Zpětně rekonstruuji použité hrany.

[Prim's algorithm in 2 minutes — Review and example](#)

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(V ^2)$
binary heap and adjacency list	$O((V + E) \log V) = O(E \log V)$
Fibonacci heap and adjacency list	$O(E + V \log V)$

Kruskalův algoritmus

Algoritmus pro hledání minimální kostry ve váženém grafu. Pracuje na principu, že z grafu vybíráme vždy **hranu s nejmenším ohodnocením**, aby se **nevytvořila kružnice**, dokud nespojíme všechny vrcholy. Komplexita je **$O(|E| * log(|V|))$** .

[Kruskal's Algorithm: Minimum Spanning Tree \(MST\)](#)

26. Řešení úloh (prohledávání stavového prostoru, rozklad na podúlohy, metody hraní her).

Stavový prostor

Graf, kde uzly představují jednotlivé stavy úlohy a hrany představují použité operátory. Je to dvojice (**S, O**), kde:

- S - množina stavů úlohy - **uzly** (vrcholy) grafu.
- O - množina operátorů - **hrany** grafu.

Úloha

Dvojice (**S₀, G**), $S_0 \in S$ a $G \subset S$ kde:

- S_0 - **počáteční stav**.
- G - **množina** všech konečných/**cílových** stavů.

Hodnotící kritéria prohledávacích metod

- **Úplnost** - Algoritmus je úplný, když vždy najde řešení, pokud daná úloha řešení má. Každá úplná metoda pro **CSP** (Constraint Satisfaction Problem - záleží pouze na nalezení cílového stavu, posloupnost operátorů je irrelevantní) je **optimální**.
- **Optimálnost** - Pokud existuje více možných řešení, metoda najde to **nejlepší**. (optimální úloha je tedy **vždy úplná**).
- **Paměťová/Prostorová a časová složitost** - Např. lineární - **O(n)**, kvadratická **O(n^2)**, linearitmická **O(n*logn)**, exponenciální **2^O(n)**, ...

Metody řešení úloh prohledáváním stavového prostoru

- **Neinformované/Slepé metody**: Nevyužívají **žádné** informace, které by mohly **usnadnit** nalezení řešení. Používají se pouze pokud o řešené úloze skutečně žádné informace **nemáme**.
- **Informované metody** - Využívají nějaké informace (**heuristické funkce**) o řešené úloze, které mohou **usnadňovat** nebo **umožňovat** jejich řešení.
- **Metody lokálního prohledávání** - Místo **systematického** prohledávání stavového prostoru prohledávají pouze **okolí aktuálního stavu**. Vhodné pro řešení optimalizačních problémů. **Nemusí být úplné ani optimální**.

Pojmy

- **Expanze uzlu** - Určení všech jeho bezprostředních následovníků - uzlů spojených hranou s expandovaným uzlem (postupná aplikace všech operátorů na uzel).

- **Ohodnocení uzlu** - Je dáno součtem **cen přechodů** (součet ohodnocení hran po cestě) z kořenového uzlu do tohoto uzlu.

Slepé metody

Prohledávání do šířky - Breadth First Search (BFS)

Úplná a optimální. Používá **frontu** pro ještě nezpracované uzly - **OPEN** a **seznam** pro již zpracované uzly - **CLOSED**.

1. Sestroj frontu OPEN a seznam CLOSED (pro variantu s CLOSED). Do OPEN vlož počáteční uzel.
2. Je-li fronta OPEN prázdná, pak úloha **nemá řešení**, jinak pokračuj.
3. Vyber z **čela** fronty OPEN uzel.
4. Je-li **cílovým** ukonči prohledávání jako **úspěšné** a **vrat' cestu**, jinak pokračuj.
5. Expanduj uzel (zde lze také testovat na to, jestli je uzel cílový), jeho následovníky co **nejsou** v OPEN ani v CLOSED, umísti do fronty OPEN, expandovaný uzel do CLOSED. Vrat' se na bod 2.

Složitost je stejná časově i prostorově a závisí na **faktoru větvení - b** a hloubce **cílového uzlu - d**:

- **bez modifikace** v bodě 5: $O(b^{d+1})$,
- **s modifikací** v bodě 5: $O(b^d)$.

Prohledávání do hloubky - Depth First Search (DFS)

1. Sestroj **zá sobník** OPEN a umísti do něj počáteční uzel.
2. Je-li OPEN **prázdný**, pak úloha **nemá řešení**, jinak pokračuj.
3. Vyber z vrcholu OPEN první uzel.
4. Je-li uzel uzlem **cílovým**, ukonči prohledávání jako **úspěšné** a vrat' cestu, jinak pokračuj.
5. Expanduj vybraný uzel a do OPEN vlož všechny jeho **bezprostřední** následníky (modifikace: bezprostředního následníka vlož na zásobník jen pokud tam **ještě není** a pokud **není předkem** právě expandovaného uzlu). Vrat' se na bod 2.

Metoda bez modifikace **není úplná ani optimální** (do nekonečna se můžeme pohybovat v cyklu). Po modifikaci **je** metoda **úplná** ale stále **není optimální**.

- **bez modifikace**: časově i prostorově má nekonečnou časovou složitost,
- **s modifikací**: časová složitost je $O(b^m)$ a prostorová složitost je $O(m)$, kde **b** je faktor větvení a **m** je počet **různých stavů**. U stromové struktury je prostorová složitost $O(b^d)$, kde **d** je hloubka.

Prohledávání do omezené hloubky - Depth Limited Search (DLS)

Jedná se o metodu DFS, která prohledává pouze do omezené hloubky od počátečního uzlu. **Není úplná ani optimální**.

1. Sestroj **zá sobník** OPEN a umísti do něj počáteční uzel.
2. Je-li OPEN **prázdné**, pak úloha **nemá řešení**. Jinak pokračuj.

3. Vyber z vrcholu OPEN první uzel.
4. Je-li vybraný uzel uzlem **cílovým**, ukonči prohledávání jako **úspěšné** a vrat' cestu. Jinak pokračuj.
5. Je-li **hloubka** vybraného uzlu **menší** než zadaná **maximální hloubka**, tak tento uzel expanduj a do OPEN vlož všechny následovníky, kteří tam ještě nejsou. Vrat' se na bod 2.

Složitost závisí na prohledávané hloubce, **b** - faktor větvení, **I** - prohledávaná hloubka:

- **časová:** $O(b^I)$,
- **prostorová:** $O(b^I)$.

Prohledávání do omezené hloubky s postupným zanořováním - Iterative Deepening Search (IDS)

Metoda je **úplná i optimální**.

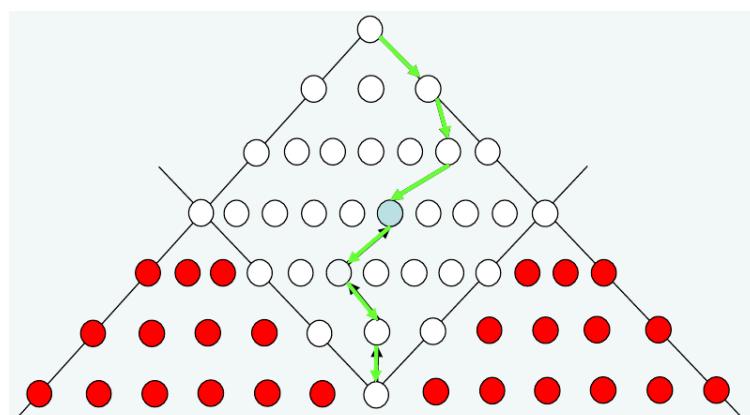
1. Nastav aktuální maximální hloubku na **1**.
2. Zavolej proceduru **DLS** s omezením na nynější hloubku.
3. Skončí-li **DLS** s úspěchem, ukonči prohledávání s úspěchem a vrat' cestu.
4. Skončí-li **DLS** s neúspěchem, tak pokud v DLS **nebyl alespoň jeden uzel expandován z důvodu dosažení maximální hloubky**, inkrementuj maximální hloubku a vrat' se na bod 2. Jinak ukonči prohledávání jako **neúspěšné**.

Složitost je dána maximální hloubkou, **b** - faktor větvení, **d** - maximální prohledávaná hloubka

- **časová:** $O(b^d)$,
- **prostorová:** $O(b^d)$.

Obousměrné prohledávání - Bidirectional Search (BS)

Metoda je **úplná i optimální**. Metodu lze použít pouze na řešení úloh s **reverzibilními** operátory (např. **Lloydova osmička**). Současně prohledává prostor **od počátečního stavu i od cílového stavu** a hledá uzel, ve kterém se obě prohledávání setkají.



Časová i prostorová složitost metody je stejná, **b** - faktor větvení, **d** - hloubka řešení:

- $O(2^*b^{(d/2)})$, což odpovídá $O(b^{(d/2)})$.

Prohledávání do šířky s respektováním cen přechodů - Uniform Cost Search (UCS)

Metoda je **úplná i optimální**. Není vhodná pro úlohy, kde **optimální řešení** leží na **málo cestách s vysokou cenou** (prohledává se zbytečně mnoho cest s malými cenami). Jedná se o variantu Dijkstrova algoritmu, kde do **prioritní fronty** se místo všech uzlů vkládají pouze ty, ke kterým jsme již došli.

1. Sestroj dva **seznamy** OPEN (prioritní fronta - uzly určené k expanzi) a CLOSED (již expandované uzly - existuje verze i bez tohoto seznamu). Do open vlož počáteční uzel včetně jeho ohodnocení.
2. Je-li seznam OPEN **prázdný**, pak úloha **nemá řešení**. Jinak pokračuj.
3. Vyber ze seznamu OPEN uzel s **nejnižším ohodnocením**.
4. Je-li vybraný uzel uzlem **cílovým**, ukonči prohledávání jako **úspěšné** a vrat' cestu. Jinak pokračuj.
5. Vybraný uzel expanduj a jeho následníky, kteří nejsou v CLOSED umísti do OPEN (**včetně jejich ohodnocení**). Expandovaný uzel vlož do CLOSED. Z uzlů, které se v OPEN vyskytují **vícekrát** ponech jenom ten s **nejnižším ohodnocením** (jinak řečeno aktualizuj ohodnocení, pokud je lepší). Vrat' se na bod 2.

Časová i prostorová složitost je dána **cenou optimálního řešení C^*** a **minimálním přírůstkem ceny ΔC_{\min}** mezi dvěma uzly:

- $O(b^{(C^*/\Delta C_{\min})})$

Prohledávání se zpětným navracením - Backtracking Search

Metoda je vhodná i pro **CSP** (pokud aplikace operátoru vede na stav porušující omezující podmínky, pak je tento operátor považován za **neaplikovatelný** - bod 3 algoritmu), **je úplná, ale není optimální**. Backtracking je podobný DFS, ale místo expanze uzlu generuje pouze jediného následníka - nejprve jednoho a při návratech pak další.

- 1. Sestroj zásobník OPEN a umísti do něj počáteční uzel.
- 2. Je-li OPEN prázdný, pak úloha nemá řešení - ukonči prohledávání jako neúspěšné. Jinak pokračuj.
- 3. Jde-li na uzel na vrchu zásobníku aplikovat první/další operátor, tak tento operátor aplikuj a pokračuj bodem 4. Jinak odstraň testovaný uzel z vrcholu a vrat' se na 2.
- 4. Je-li vygenerovaný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrat' cestu. Jinak ulož uzel na vrchol zásobníku (modifikace: uzel na zásobník ulož, pokud se tam ještě nenachází) a vrat' se na 2.

Časová a prostorová složitost je stejná jako pro DFS:

- **bez modifikace**: časově i prostorově má nekonečnou časovou složitost,
- **s modifikací**: časová složitost je $O(b^m)$ a prostorová složitost je $O(m)$, kde **b** je faktor větvení a **m** je počet **různých stavů**.

Prohledávání s dopřednou kontrolou - Forward Checking Search

Vhodné i pro CSP. Metoda je **úplná** i **optimální**. [forward checking, CSP heuristics](#).

Funguje na principu, že každé proměnné (uzlu) přiřadí množinu všech přípustných hodnot pro danou proměnnou (u barvení mapy to budou **všechny dostupné barvy**). U první proměnné vybereme některou z přípustných hodnot (např. modrou barvu) a aktualizujeme přípustné hodnoty u ostatních proměnných (proměnné, které na mapě sousedí s právě obarvenou proměnnou nemohou být modré). Vybereme další proměnnou a jí přiřadíme hodnotu (např. zelenou) a takto postup opakujeme dokud nejsou pro všechny proměnné vybrány hodnoty. Může se ale stát, že u některé proměnné dojde k odstranění všech možných hodnot, které ji lze přiřadit (už nelze obarvit, aby nebylo porušeno pravidlo). Pak musíme u předešlých proměnných změnit jejich přiřazení (postupně se vynořovat a měnit). Pokud již nejde přiřazení nijak změnit, úloha nemá řešení.

Heuristiky pro výběr proměnných:

- Proměnná s **nejmenším počtem přípustných hodnot**.
- Proměnná, která má **největší vliv na omezení zbyvajících volných proměnných**.

Heuristiky pro přiřazení hodnoty proměnné:

- Vyber hodnotu, která **vylučuje nejméně hodnot**, které mají společná omezení s vybranou proměnnou.

Složitost **n** - počet proměnných, **m** - počet přípustných hodnot:

- časová: $O(m^n)$,
- prostorová: $O(n)$.

Prohledávání s minimalizací konfliktů - Min-Conflict Search

Vhodné i pro CSP. **Neexistuje důkaz** o její **úplnosti** ani **optimálnosti** a tedy ani odhad časové složitosti.

1. Přiřaď každé proměnné x_i ($i = 1, \dots, n$) **libovolnou** hodnotu z množiny **přípustných** hodnot. Nastav pomocné proměnné i (počítadlo proměnných) a j (počítadlo **správně** přiřazených proměnných) na hodnoty **1**.
2. Spočítej počet **možných konfliktů** pro každou hodnotu proměnné x_i .
3. Pokud existuje pro jinou možnou hodnotu x_i počet konfliktů **menší nebo stejný** než pro její aktuální hodnotu, tak ji změň na tuto hodnotu a nastav hodnotu **j na 1**, jinak **inkrementuj j**.
4. Pokud **j == n** (všechny proměnné jsou přiřazeny nejlépe), pak bylo nalezeno optimální řešení. Jinak inkrementuj **i a pokračuj**.
5. Je-li hodnota **i > n**, pak **i = 1** a pokračuj bodem 2.

V paměti si udržuje pouze nynější stav a počet proměnných, tedy prostorová náročnost je: **O(1)**

Informované metody

Používají heuristické funkce pro **odhad ceny** z aktuálního stavu do cíle a přičítají jej

k aktuální ceně cesty. $f(s_k)$ je ohodnocení k-tého stavu, které je dáno součtem:

- $g(s_k)$: cena cesty od **počátečního stavu** ke **k-tému stavu**.
- $h(s_k)$: je odhadovaná (pomocí **heuristické funkce**) cena cesty od **k-tého stavu** ke stavu **cílovému**.

Prohledávání od nejlepšího - Best First Search (BestFS)

- Sestroj **seznam OPEN** (popř. CLOSED) a vlož do něj počáteční uzel včetně jeho ohodnocení.
- Je-li seznam OPEN prázdný, pak úloha nemá řešení - skončí s neúspěchem. Jinak pokračuj.
- Vyber z OPEN uzel s nejlepším ohodnocením.
- Je-li vybraný uzel uzlem cílovým - skončí s úspěchem a vrát' cestu. Jinak pokračuj.
- Expanduj vybraný uzel. Všechny jeho následovníky co nejsou předky (využití CLOSED), umísti do OPEN včetně jejich ohodnocení. Z těch co jsou v OPEN **vícekrát** ponech pouze uzel s **nejlepším ohodnocením**. Vrať se na 2.

Neinformovaná metoda UCS je extrémním případem BestFS, u které je hodnota $h(s_k)$ **vždy 0**.

Chamtivé prohledávání - Greedy Search (GS)

Jedná se také o extrémní případ BestFS, kdy $g(s_k) = 0$. **Úplná** (pokud se používá seznam CLOSED), ale **není optimální**.

Časová a prostorová složitost je **$O(b^d)$** , kde **b** je faktor větvení a **d** je hloubka.

A* prohledávání/Search

Spadá pod BestFS. Je **úplná** a **optimální**. Heuristická funkce je **spodním odhadem skutečné ceny (nejlepší/nejlevnější odhad ceny)** cesty od ohodnoceného uzlu k cíli. Tato funkce nikdy **nesmí** odhadovanou cenu cesty **přecenit**, jinak algoritmus nebude fungovat správně, $h = 0$ je přípustná vždy (ale pak je to metoda UCS).

Časová i prostorová náročnost závisí na heuristice, pohybuje se od:

- pro **h** blízké nule: $O(b^d)$,
- pro **h** rovné skutečné ceně: $O(d)$,

kde **b** je faktor větvení a **d** je hloubka cíle.

Metody lokálního prohledávání

většinou nejsou úplné ani optimální, ale mohou být například rychlé.

Hill Climbing

Není úplná ani optimální. Umí jít pouze jedním směrem a nedokáže se vracet. Pokud by měla hledat např. nejvyšší horu z údolí, je velmi pravděpodobné, že se zastaví na prvním kopci, protože neumožňuje klesání.

1. Vytvoř uzel **Current** a ulož do nej počáteční stav s_0 spolu s jeho

ohodnocením.

2. Expanduj **Current**, ohodnoť jeho bezprostřední následníky a vyber z nich **nejlépe ohodnoceného (Next)**.
3. Je-li ohodnocení **Current** lepší než ohodnocení **Next**, ukonči řešení a vrat' **Current**. Jinak pokračuj.
4. Nahraď **Current** uzlem **Next**. Vrat' se na 2.

Prostorová složitost je **O(1)** - používáme pouze 2 proměnné Current a Next. Časová složitost je **O(d)** - jdeme pouze jednou cestou.

Simulated Annealing

Není úplný ani optimální. Pracuje jak s ohodnocením, tak s **náhodností**. Narození od Hill Climbing **dokáže opustit lokální extrémy**. Je inspirována tuhnutím kovů. S postupně **klesající pravděpodobností** (tuhnutím kovu s klesající teplotou) umožňuje vybrat **hůře** ohodnocený uzel - na začátku algoritmu je poměrně pravděpodobné, že opustíme lokální extrém.

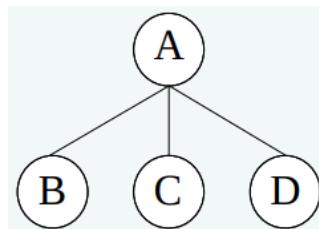
1. Vytvoř tabulku pro klesání "teploty" v závislosti na kroku výpočtu.
2. Vytvoř pracovní uzel Current a ulož do něj počáteční stav a jeho ohodnocení. Nastav krok výpočtu na 0 ($k=0$).
3. Z tabulky zjistí aktuální teplotu T . Je-li $T=0$, ukonči řešení a vrát jako výsledek Current. Jinak pokračuj.
4. Expanduj Current a z jeho následovníků vyber náhodně jednoho z nich (Next).
5. Vypočítej rozdíl ohodnocení uzel Current a Next
 $\Delta E = \text{value}(Current) - \text{value}(Next)$.
6. Pokud $\Delta E > 0$, pak nahraď uzel Current uzlem Next. Jinak je zaměň s pravděpodobností $p = e^{\Delta E/T}$.
7. Inkrementuj k a vrát se na 3.

Časová složitost závisí na rychlosti klesání teploty (rychlosti klesání pravděpodobnosti, že bude vybrán hůře ohodnocený uzel). Prostorová složitost je **O(1)** - konstantní.

Metody řešení úloh rozkladem na podproblémy (AND/OR grafy)

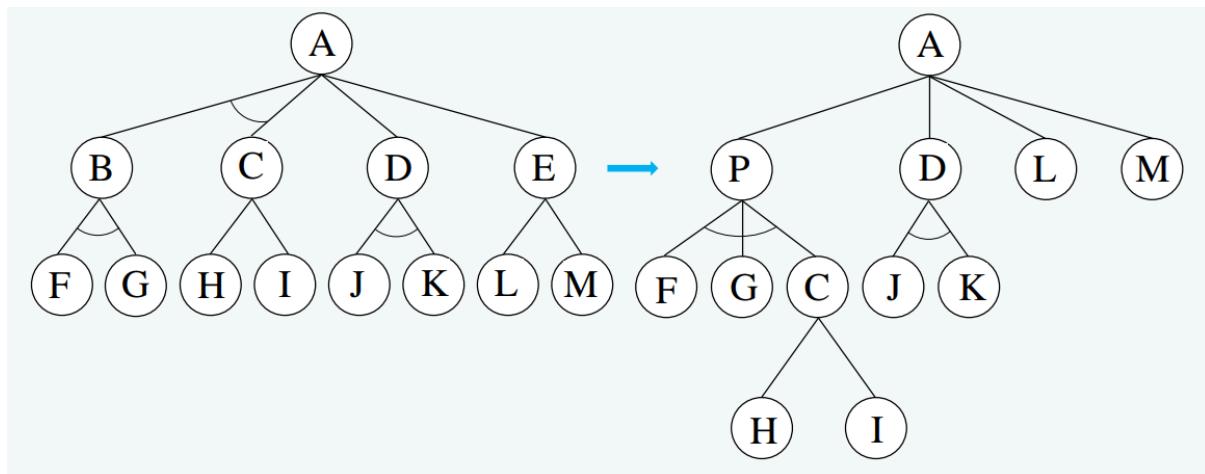
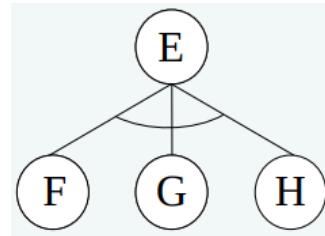
U těchto úloh uzly znamenají **problémy/podproblémy NIKOLIV** stavy. Problém lze rozložit dvěma způsoby:

- **OR uzel:** Problém je řešitelný, pokud je **alespoň jeden** z podproblémů řešitelný.



- **AND uzel:** Problém je řešitelný, pokud jsou **řešitelné všechny** jeho

podproblémy.



Obecný **AND/OR** graf (graf s uzly AND a OR) lze převést na graf, ve kterém jsou v každé **vrstvě** buď pouze **AND**, nebo **OR** uzly. Každou musí být možné převést ze stavového prostoru na **rozklad podproblémů**.

Slepé AND/OR graf pro BFS, DFS

BFS a DFS prohledávání s tím, že řešitelnost uzlu je dána jeho typem (AND, OR, případně kombinací AND a OR) a řešení spočívá v dokázání, že je **řešitelný i počáteční uzel** (kořen). Toho se docílí **propagováním informace o**

řešitelnosti/neřešitelnosti z nižších vrstev do vyšších a na tom je algoritmus založený.

1. Sestroj OPEN (**fronta** pro BFS, **zásobník** pro DFS) a prázdný **graf/strom G** a do obou ulož počáteční uzel (problém), který **nesmí** být elementárně řešitelný nebo neřešitelný.
2. Výjmi z OPEN uzel a označ jej X.
3. Expanduj X a všechny jeho následovníky připoj ke grafu G.
 - a. Pro všechny **řešitelné** následníky X přenes **informaci o jejich řešitelnosti jejich předchůdcům**. Je-li **řešitelný počáteční** problém, ukonči řešení jako **úspěšné**.
 - b. Pro všechny **neřešitelné** následníky X přenes **informaci o jejich neřešitelnosti jejich předchůdcům**. Není-li **řešitelný počáteční** problém, ukonči řešení jako **neúspěšné**.
 - c. Všechny ostatní následníky X ulož do OPEN.
4. Odstraňte z OPEN všechny uzly, které mají **vyřešení předchůdce** (nedává smysl je už řešit).
5. Je-li OPEN **prázdný** - skonči s **neúspěchem**. Jinak se vrat na 2.

Slepý AO pro Backtracking

Podobný jako DFS, test uzlu na **řešitelnost/neřešitelnost** probíhá **před** generováním jeho následníka - vždy se generuje pouze **jeden následník**, který se vyhodnotí a až poté se **případně generuje další**.

1. Sestroj **graf G** a **zásobník OPEN** a do obou ulož počáteční uzel.
2. Je-li uzel na vršku OPEN řešitelný, pak:
 - a. Je-li tento uzel počáteční - skonči **úspěšné**.
 - b. Jinak přenes informaci o řešitelnosti na předchůdce a uzel z OPEN odstraň.
3. Je-li uzel na vršku neřešitelný, pak:
 - a. Je-li tento uzel uzlem počátečním, ukonči řešení jako **neúspěšné**.
 - b. Jinak přenes informaci o neřešitelnosti na předchůdce, a uzel odstraň z OPEN.
4. Není-li uzel na vršku OPEN řešitelný/neřešitelný, pak **generuj** jeho následníka, ulož ho do OPEN a připoj k G.
5. Vrat se na bod 2.

Informovaný AO* algoritmus

Algoritmus řeší, jestli je uzel řešitelný a také jaká je cena vyřešení uzlu. Může skončit neúspěšně, i když existuje řešení, pokud je převýšena jeho **maximální cena**.

Výpočet ceny pro uzly je (může být doplněno o heuristickou funkcí):

- Ohodnocení **AND** je rovno **součtu** ohodnocení všech jeho následníků.
- Ohodnocení **OR** je rovno **nejmenšímu** z ohodnocení jeho následníků.

Algoritmus pracuje následovně:

1. Sestroj strukturu G (AO strom) a umísti do něj počáteční uzel (INIT) s

ohodnocením. Stanov hodnotu FUTILITY (maximální povolenou cenu).

2. Je-li INIT označen jako řešitelný (SOLVED), ukonči řešení jako úspěšné. Je-li INIT \geq FUTILITY, skonči s neúspěchem. Jinak pokračuj.
3. Procházej nejnadějnější podstrom až narazíš na neexpandovaný uzel (NODE).
4. Expanduj NODE, pokud nemá následníka, přiřaď mu hodnotu FUTILITY a přejdi na 7. Jinak pokračuj.
5. Představují-li některí bezprostřední následníci elementární úlohy, označ je SOLVED (0 ohodnocení) u zbývajících jej vypočti.
6. Připoj následníky NODE ke stromu G a přenes informaci o jejich ohodnocení směrem k INIT.
7. V uzlech OR označ nejnadějnější podstomy.
8. Vrat' se na bod 2.

graficky viz: <https://www.goeduhub.com/7063/implement-ao-search-algorithm>

Metody hraní her

Cílem je určit tah hráče na tahu, který povede k vítězství, nebo pro který je pravděpodobnost jeho vítězství největší. Nejprve budeme uvažovat pouze hry, které hrají dva pravidelně se střídající hráči, které označíme symboly **A** a **B**. Každý z nich chce vyhrát a oba mají úplný přehled o aktuálním stavu hry. Pro tyto hry obecně platí:

- Hráč na tahu (označuje se vždy jako hráč **A**) zvítězí, vede-li k jeho vítězství **alespoň jeden** tah – problém **OR**.
- Hráč na tahu zvítězí, vedou-li po jeho tahu k jeho vítězství **všechny možné tahy protihráče** (hráče **B**) – problém **AND**.

Rozdělení her dvou protihráčů

- **Jednoduché hry** - Lze prozkoumat všechny možné tahy (NIM). K řešení lze použít **AND/OR** algoritmy.
- **Složité hry** - Zkoumá se pouze několik následujících tahů (šachy).
- **Hry s neurčitostí** - (hod kostkou např.) Zkoumá se pouze několik následujících tahů s respektováním **neurčitosti** (hod kostkou - člověče). Pracuje se s očekávanými hodnotami.

MinMax (Složité hry)

Každý stav je ohodnocen hodnotící funkcí (kladná příznivé pro hráče na tahu - **A**, vítězství/prohra má hodnotu +/nekonečno). Hráč **A** chce maximální hodnoty - v OR uzlu se bere **max** jeho potomků, **B** chce minimální - v AND uzlu se bere min jeho potomků. Řešením problému v daném stavu je určení **nejvhodnějšího tahu hráče A** (pro každý tah se musí řešit znovu a znovu). Metoda je **rekurzivní** a zahajuje se, když je **na tahu hráč A**. Vstupem jsou

- stavy hry X,

- maximální hloubka prohledávání.

Výstupem je

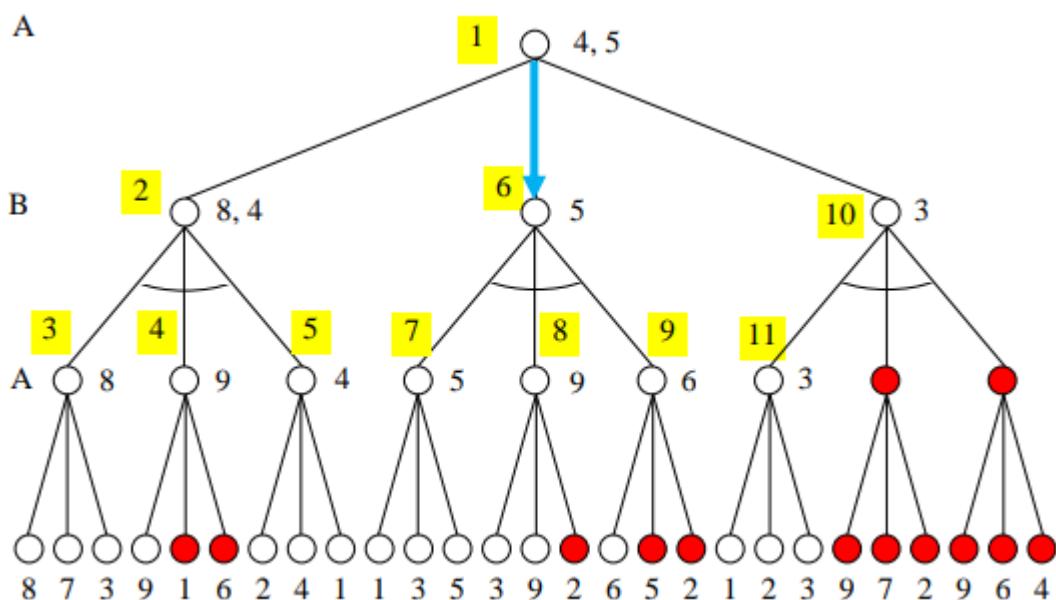
- **aktuální stav**,
- **tah**, který k tomuto stavu vede.

Algoritmus může vypadat následovně

1. Je-li uzel **X** listem vrací ohodnocení tohoto uzlu.
2. Je-li na tahu hráč **A**, tak postupně pro všechny jeho možné tahy volá **MinMax** pro hráče **B** a vrací **maximální** z navrácených hodnot a tah, který k tomuto vede.
3. Je-li na tahu hráč **B**, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu X, tj. stavy před tahem hráče A) volá proceduru **MiniMax** pro hráče **A** a vrací **minimální** z navrácených hodnot.

Popis příkladu na obrázku:

- žlutá čísla určují **pořadí** při vyhodnocování,
- nepodbarvená čísla určují **ohodnocení uzelů** a změny jejich ohodnocení,
- **červené** uzly jsou vyhodnocovány **zbytečně** (už je jasné, že si hráč tuto větev stejně nezvolí. V kroku 4 je jasné, že hráč **B** si **radši** vybere větev kroku 3 (ohodnocení 8 < 9), než větev 4, která má již po prvním potomkovi ohodnocení 9 a už může mít pouze vyšší).



AlfaBeta (Složité hry)

Step by Step: Alpha Beta Pruning

Zbytečnému vyšetřování uzelů lze zabránit pomocí alfa-beta řezů. **Alfa** řezy zabraňují zbytečnému vyšetřování tahů hráče **A**, **beta** řezy pak zbytečnému vyšetřování tahů hráče **B**. Procedura AlfaBeta vychází z procedury MiniMax, je rekurzivní, vstupními parametry jsou stejné a navíc přidává parametry α a β , které jsou na začátku nastaveny na $\alpha = -\infty$ a $\beta = +\infty$ (tedy opačné hodnoty než hráči A a B požadují).

1. Je-li uzel X listem, procedura vrací ohodnocení tohoto uzlu.

2. Je-li na tahu A:

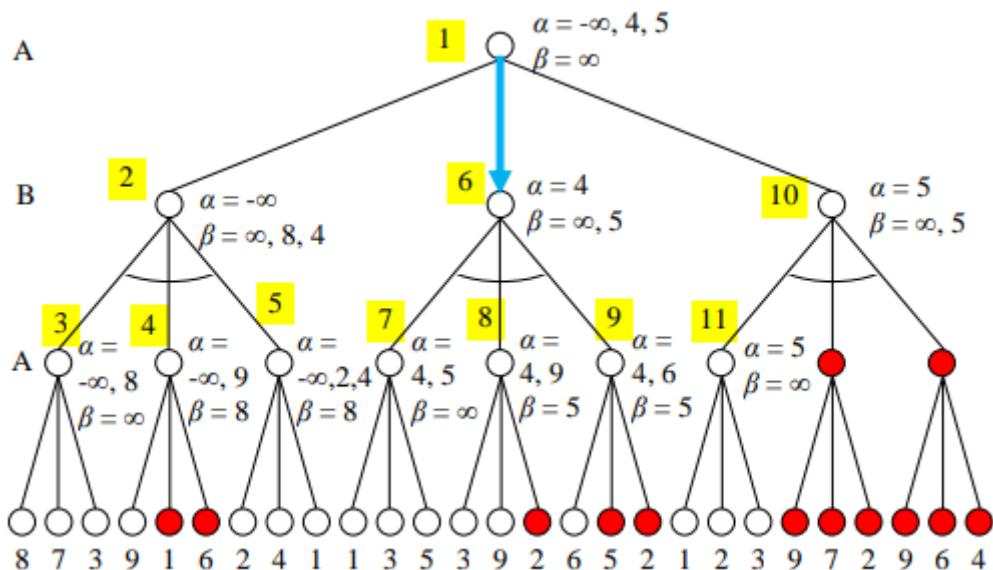
- Dokud platí $\alpha < \beta$, tak postupně pro všechny tahy volá **AlfaBeta** s aktuálními hodnotami α a β (po každém volání se α mění). Po každém vyšetření tahu nastaví α na **maximum** (postupně se zvětšuje z $-\infty$) z aktuální a vracené hodnoty.
- Je-li $\alpha \geq \beta$ (před každým dalším následník se **musí toto kontrolovat**), nebo nemá-li X žádného dalšího bezprostředního následníka, vrací aktuální hodnotu α a tah, který vede k nejlepšímu stavu (v případě více stejných ten první nalezený).

3. Je-li na tahu B:

- Pokud platí $\alpha < \beta$, tak postupně pro tahy volá AlfaBetu s aktuálními α a β (po každém volání se β mění). Po každém vyšetření nastaví β na **minimum** (postupně se zmenšuje z ∞) z aktuální a vrácené hodnoty.
- Je-li $\alpha \geq \beta$ nebo nemá-li X žádné následníky, vrací **aktuální β** .

Popis příkladu na obrázku:

- žlutá čísla určují **pořadí** při vyhodnocování,
- nepodbarvená čísla určují **ohodnocení α a β** a změny jejich ohodnocení,



ExpectiMinimax (Hry s neurčitostí)

[Games: Q2. Expectiminimax AI U3 D5 Prababilistic Cut ExpectiMax AlphaBeta](#)

Podobné MinMax, počítá se ale z pravděpodobnosti, že k nějakému stavu dojde. Metoda je rekurzivní a pracuje s **expectimin** pro hráče B (očekávané **minimum**) a **expectimax** pro hráče A (očekávané **maximum**). **Expectimin** a **expectimax** jsou spočteny jako **součet ohodnocení po všech možných výsledcích hodu kostky** (případně jiné pravděpodobnostní veličiny). Např. pokud má uzel 2 následovníky, první má ohodnocení 10, druhý má ohodnocení 5. Pravděpodobnost vybrání 1. následníka je ale pouze 0.25, druhého 0.75. celkové ohodnocení uzlu tak bude

$$0.25*10 + 0.75*5 = 6,25.$$

$$\text{expectimin} (C_i) = \sum_k P(h_k) * \min_j (D_{ikj})$$

$$\text{expectimax} (D_j) = \sum_k P(h_k) * \max_i (C_{jki})$$

V těchto vzorcích značí:

C_i	i -tý možný tah hráče A po jeho hodou kostkou
D_{ikj}	ohodnocení stavu po i -tém tahu hráče A, hodu h_k a j -tém tahu hráče B
D_j	j -tý možný tah hráče B po jeho hodou kostkou
C_{jki}	ohodnocení stavu po j -tém tahu hráče B, hodu h_k a i -tém tahu hráče A
$P(h_k)$	pravděpodobnost hodu h_k

1. Je-li uzel **X** listem (konečným stavem hry, nebo uzlem v **maximální hloubce**) procedura vrací **ohodnocení tohoto uzlu**.
2. Je-li na tahu hráč **A**, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu X) volá proceduru **Expectiminimax** pro hráče B, vrací **maximální** hodnotu z navrácených hodnot **expectimin** a tah, který k nejlépe ohodnocenému bezprostřednímu následníkovi **vede** (tentohle **tah** má opět význam pouze u kořenového uzlu, kdy představuje nejvýhodnější reálný tah hráče **A**).
3. Je-li na tahu hráč **B**, tak postupně pro všechny jeho možné tahy (bezprostřední následníky uzlu **X**) volá proceduru **Expectiminimax** pro hráče A a vrací **minimální** hodnotu z navrácených hodnot **expectimax**.

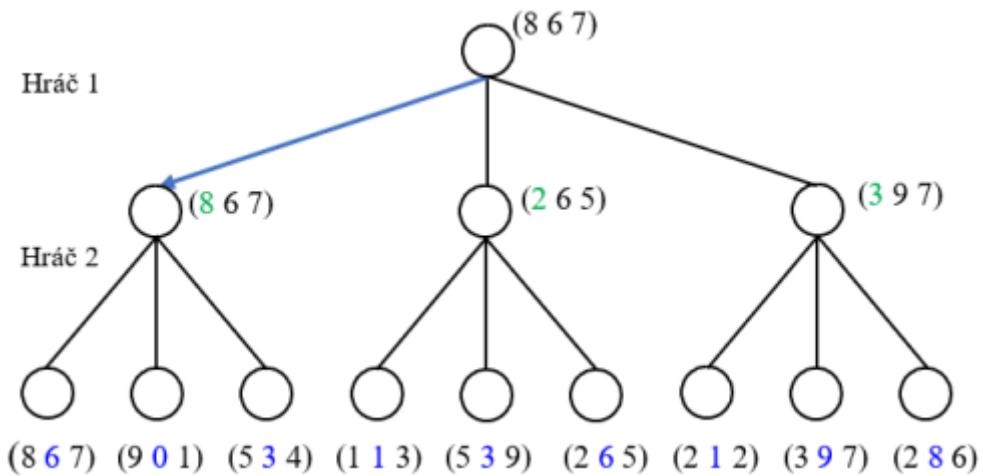
Metody hraní her n hráčů

Hráči se pravidelně střídají, každý hráč chce vyhrát a všichni mají úplný přehled o aktuálním stavu hru.

Jednotlivé stavy hry jsou ohodnocovány **jedinou hodnotící funkcí** aplikovanou na **každého hráče zvlášt'**, přičemž hodnota této funkce musí být **tím vyšší**, čím je daná pozice pro hodnoceného hráče **výhodnější**. U každého stavu je pak jeho hodnocení dánou **seznamem hodnot hodnotící funkce** pro jednotlivé hráče.

Například pro hru se třemi hráči ohodnocení stavu **(8 3 5)** znamená, že daný stav má ohodnocení **8 pro prvního** hráče, **3 pro druhého** hráče a **5 pro třetího** hráče.

Každý hráč si vybírá pro něj nejvýhodnější ohodnocení. Na obrázku jsou zeleně vyznačena ohodnocení pro hráče 1 a modře ohodnocení pro hráče 2.



Procedura Maxⁿ (hry více hráčů)

Jedná se o zobecnění procedury **Minimax** pro **n** hráčů. Proceduru volá vždy hráč na tahu

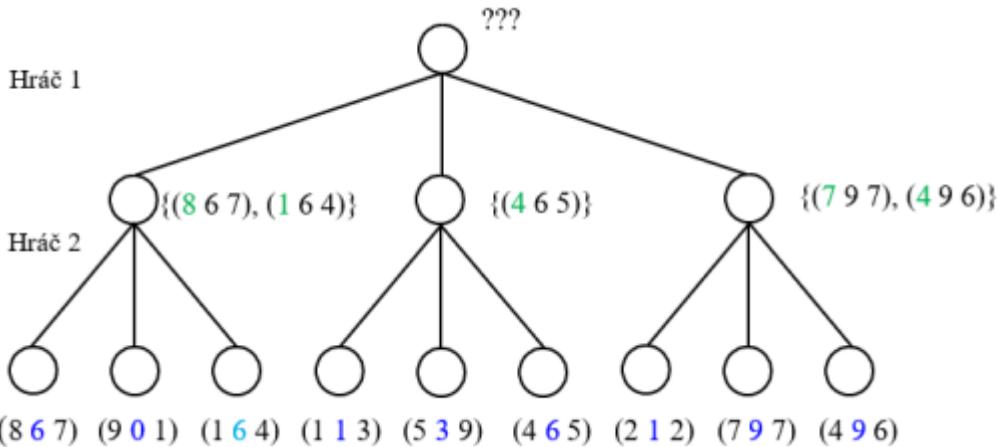
- vstup: aktuální stav hry (uzel X)
- výstup: ohodnocení tohoto stavu

Princip algoritmu:

1. Je-li uzel X listem (konečným stavem hry nebo uzlem v maximální hloubce), tak procedura vrací ohodnocení tohoto uzlu (v_1, v_2, \dots, v_n).
2. Jinak tato procedura pro aktuálního hráče i volá rekursivně sama sebe na všechny své bezprostřední následníky (tj. stavy před tahem následujícího hráče j ($j = i + 1$ a **pokud $j > n$, pak $j = 1$**) a vrací ohodnocení, které je pro daného hráče, tj hráče i , nejvýhodnější.

Soft-Maxⁿ

Problémem procedury **Maxⁿ** je nejednoznačnost hodnocení uzlu (stavu hry) v případech, kdy stejně **maximální ohodnocení má několik** jeho bezprostředních následníků. Tento problém procedura Soft-Maxⁿ řeší, vracením množiny se stejnými ohodnoceními pro příslušného hráče, a tím umožňuje hráči na tahu informovanější rozhodování.



Pokud **hráči 1** stačí k vítězství 4 body, pak si tento hráč zřejmě vybere tah rovně dolů k uzlu **(4 6 5)**. V opačném případě může bud' **riskovat**, tj. zvolit tah vlevo (hráč 2 si však může vybrat tah **(1 6 4)**, který je pro hráče 1 velmi nevýhodný - hodnota 1), nebo sázet raději na jistotu, tj. zvolit tah vpravo.

Sprehladnenie algoritmov v tabuľke:

	Algorithms	Typ	Úplnosť	Optimálnosť	Využitie	Priestorová	Časová
1	BFS	Slepá	Áno	Áno	Fronta	$O(b^{d+1})$ (*)	$O(b^{d+1})$ (*)
2	BS	Slepá	Áno	Áno	Fronta	$O(2 \cdot b^{d/2}) \sim O(b^{d/2})$	$O(2^{d/2}) \sim O(b^{d/2})$
3	DFS	Slepá	Nie (*)	Nie	Zásobník	$O(\infty)$ (*)	$O(\infty)$ (*)
4	DLS	Slepá	Nie	Nie	Zásobník	$O(b \cdot l)$	$O(b^l)$
5	IDS	Slepá	Áno	Áno	Zásobník	$O(b \cdot d)$	$O(b^d)$
6	Backtracking	Slepá	Áno	Nie	Zásobník	$O(b \cdot m/b)$	$O(b^{m/b})$
7	Backtracking CSP	Slepá	Áno	Áno	Zásobník	$O(n)$	$O(m^n)$
8	UCS	Slepá	Áno	Áno	Seznam	$O(b^{C^* / \Delta c_{min}})$	$O(b^{C^* / \Delta c_{min}})$
9	A*	Informovaná	Áno	Áno	Seznam	$O(b^d) \sim O(d)$	$O(b^d) \sim O(d)$
10	GS	Informovaná	Áno (*)	Nie	Seznam	$O(b^d)$	$O(b^d)$
11	BestFS	Informovaná	Áno	Áno	Seznam		
12	Hill-climbing	Lokálne prehľadávanie					
13	Simulované žihání	Lokálne prehľadávanie					
14	AO – AND/OR	Lokálne prehľadávanie					

GS (*) - Pokud se do seznamu OPEN ukládají všechni bezprostřední následníci expandovaného uzlu \Rightarrow i jeho předci(v bodu 5 se pak vypustí kontrola "kteří nejsou jeho předky") pak **GS není uplný**!

DFS (*) - Modifikovaná metoda (s eliminací stejných stavů a předků v Open) je **úplná**, ale **není optimální**. Potom

Časová: $O(b^{m/b})$
Priestorová: $O(b \cdot m/b)$

BFS (*) - Metodu BFS lze modifikovat testováním cílového stavu již při generování uzlu. Potom

Časová: $O(b^d)$
Priestorová: $O(b^d)$

Odkazy:

- Tabulka metod ve větším:
<https://cdn.discordapp.com/attachments/539908031157370900/577122492934520833/unknown.png>
- [Metody prohledávání](#)

27. Strojové učení (učení s učitelem, učení bez učitele, posilované učení).

Strojové učení (ML, Machine Learning) je schopnost inteligentního systému **měnit své znalosti** tak, aby příště vykonával stejnou nebo podobnou činnost **účinněji a efektivněji**.

Učení s učitelem

Spočívá v tom, že pro každý krok učení je **známá požadovaná odezva** a systém je tak okamžitě informován o **aktuálním hodnocení jeho poslední akce**. Učení se provádí na tzv. **trénovací množině** příkladů **T**, kdy každý příklad je představován **množinou vstupních hodnot** (vstupním vektorem) a **množinou správných/požadovaných výstupních hodnot** (výstupním vektorem):

$$T = \{(\vec{i}_1, \vec{d}_1), (\vec{i}_2, \vec{d}_2), \dots, (\vec{i}_P, \vec{d}_P)\},$$

$\vec{i}_i \dots i\text{-tý vstupní vektor}$

$\vec{d}_i \dots i\text{-tý výstupní vektor}$

Často se **množina** dostupných dat **nepoužívá** k trénování **celá**, ale rozdělí se na dvě nebo tři podmnožiny. První (**trénovací, cca 80 % dat**) se použije k trénování, druhá (**testovací, cca 10 % až 20 % dat**) se použije k testování a třetí (**cca 10 % dat**) se někdy použije k doladění parametrů.

Metody učení s učitelem pracují s vektory, které nabývají symbolických hodnot (případné **číselné hodnoty** jsou rovněž chápány jako symbolické) a jsou založené na předpokladu, že každá hypotéza, která **vyhovuje dostatečně velké množině trénovacích příkladů**, bude vyhovovat i dalším, dosud **neznámým příkladům**.

Tvorba rozhodovacích stromů (Decision Tree Building)

Rozhodovací stromy slouží ke klasifikaci objektů na základě hodnot jejich atributů/vlastností. Jsou vytvářeny ze známé množiny příkladů. Používají se při

Klient	Příjem	Dluh	Historie úvěrů	Ručení	Risk
1	< 15	vysoký	špatná	žádné	vysoký
2	15 – 35	vysoký	neznámá	žádné	vysoký
3	15 – 35	nízký	neznámá	žádné	přiměřený
4	< 15	nízký	neznámá	žádné	vysoký
5	> 35	nízký	neznámá	žádné	nízký
6	> 35	nízký	neznámá	adekvátní	nízký
7	< 15	nízký	špatná	žádné	vysoký
8	> 35	nízký	špatná	adekvátní	přiměřený
9	> 35	nízký	dobrá	žádné	nízký
10	> 35	vysoký	dobrá	adekvátní	nízký
11	< 15	vysoký	dobrá	žádné	vysoký
12	15 – 35	vysoký	dobrá	žádné	přiměřený
13	> 35	vysoký	dobrá	žádné	nízký
14	15 – 35	vysoký	špatná	žádné	vysoký

dolování dat z databází.

Příklad viz str 105: <https://www.fit.vutbr.cz/study/courses/IZU/private/2022-opora-IZU.pdf>

Algoritmus Decision Tree

Jedná se o základní algoritmus pro budování **rozhodovacích stromů**. Má dva vstupní parametry:

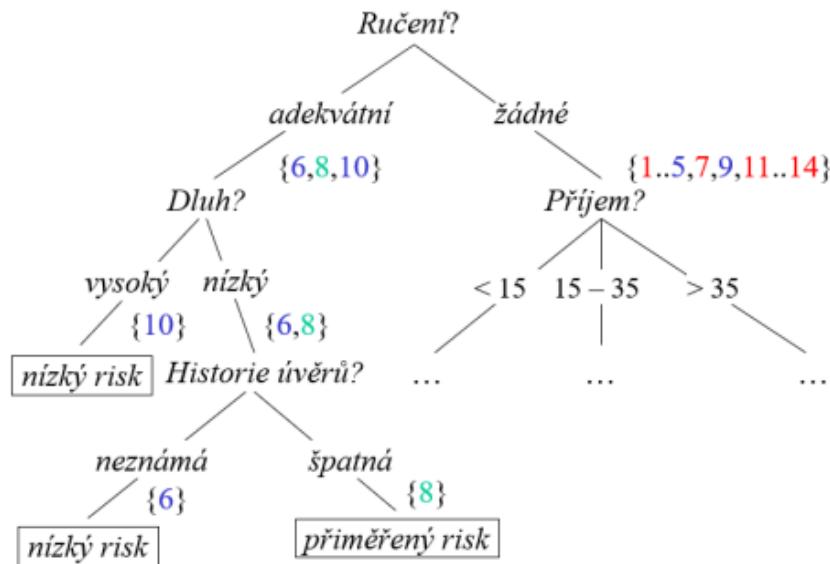
- **množinu příkladů MP**: jednotlivé řádky tabulky viz výše,
- **množinu podmínkových atributů MA**.

Algoritmus funguje následovně:

1. Patří-li **všechny prvky** množiny příkladů **MP** do **stejné třídy**, vratěte **listový uzel** označený touto třídou, jinak pokračujte.
2. Je-li množina atributů **MA** prázdná, vratěte **listový uzel** označený **disjunkcí** všech tříd, do kterých patří prvky v množině příkladů **MP**, jinak pokračujte.
3. Vyberte atribut **Ai**, odstraňte jej z množiny atributů **MA** a učiňte jej kořenem **aktuálního stromu**. Nechť **MA-i** je množina atributů **MA** bez atributu **Ai**.
4. Pro každou hodnotu **Hji** vybraného atributu **Ai**:
 - a. Vytvořte novou větev stromu označenou hodnotou **Hji**.
 - b. Volejte rekurzivně algoritmus s parametry **MPji** a **MA-i**, kde množina **MPji** je podmnožinou všech prvků množiny příkladů **MP**, které **mají** hodnotu **Hji** atributu **Ai**.
 - c. Připojte vrácený podstrom/uzel k této větvi.

Jinak řečeno, vyváří **strom na základě rozhodovacích parametrů**. Hloubka stromu závisí na tom, jestli je možné nějaký rozhodovací atribut ignorovat, protože nehledě na jeho hodnotu jsou výsledky z trénovací množiny stejné.

Na **základě trénovací množiny**, viz tabulka výše, lze říct, že pokud má uživatel



adekvátní ručení a vysoký dluh je **risk** poskytnutí úvěru **nízký** bez zohlednění jeho **příjmu a historie úvěru** (příjem a historie úvěru budou mít v realitě vliv při rozhodnutí o riziku, ale trénovací množina je neobsahuje, takže se podle nich nemůže naučit a musí je ignorovat).

Algoritmus je jednoduchý, ale obsahuje zásadní problém, kterým je výběr atributu **A_i** v bodě 3. **Nevhodné** výběry vedou k **hlubokým** a **neefektivním** vyhledávacím stromům, přestože **optimální strom** může být poměrně **jednoduchý**.

Algoritmus ID3 (Induction of Decision Tree)

Jedná se o modifikaci algoritmu Decision Tree, který **řeší** problém při **výběru rozhodovacího atributu A_i** v bodě 3. Výběr atributu není náhodný, ale je prováděn tak, aby byl **maximalizován informační zisk**, tj. aby byl **nejdříve** vybrán takový **atribut**, který co **nejvíce ovlivní výsledné rozhodnutí** (atribut, který ovlivňuje rozhodnutí úplně nejvíce je pak kořenem rozhodovacího stromu). Lze na to nahlížet také tak, že **množiny**, které vzniknou **rozdelením dle nějakého atributu** mají co **nejmenší míru entropie** (míru neuspořádanosti) tj. je v nich co nejvíce prvků spadajících pod stejný výsledek. Pokud jsou v množině 3 prvky, které spadají **všechny do jedné kategorie** výsledků (např. nízký risk), je míra **entropie** této množiny **nulová**. Pokud zde budou naopak 3 prvky spadající do **3 různých kategorií** (nízký, přiměřený a vysoký risk), je míra entropie této množiny **nejvyšší**. Vybíráme tedy takový **atribut**, který rozdělí množinu na podmnožiny s **nejmenší entropií** a přinese tak největší informační zisk.

Informační zisk s respektováním hodnot **atributu Příjem (<15, 15-35, >35)** se

vypočítá takto: **MP1** (< 15) = $\{1, 4, 7, 11\}$, **MP2** ($15 - 35$) = $\{2, 3, 12, 14\}$, **MP3** (> 35) = $\{5, 6, 8, 9, 10, 13\}$ (už jen podle rozdělení prvků do množin je zřejmé, že MP1 bude mít nulovou entropii, MP2 bude mít z této trojice největší entropii a entropie MP3 bude mezi).

Celková entropie rozdělení podle atributu Příjem je pak spočítána jako **vážený**

$$E(MP_1) = -\frac{4}{4} \log_2 \left(\frac{4}{4}\right) = 0$$

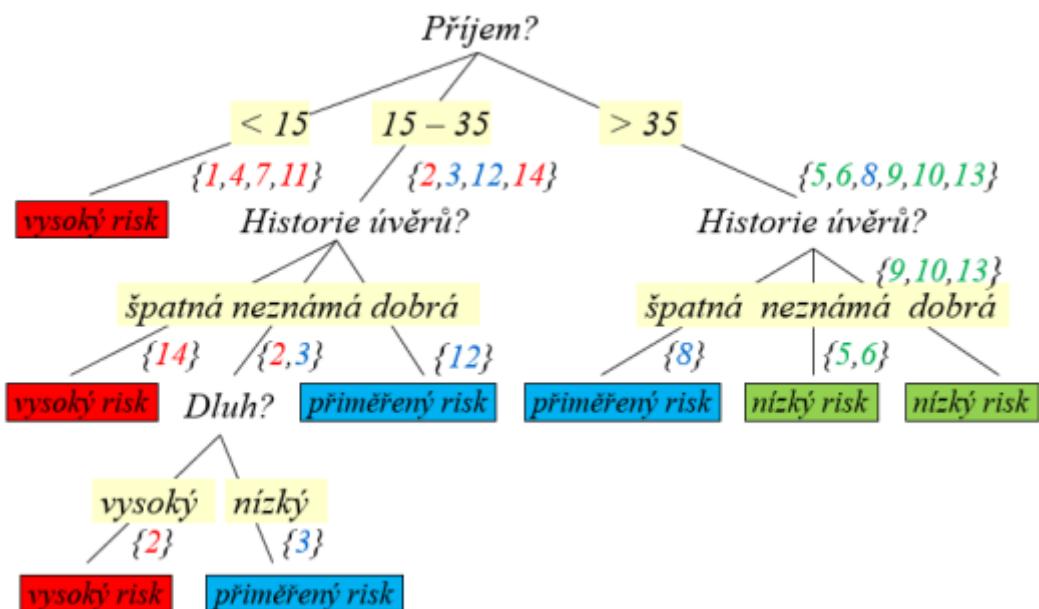
$$E(MP_2) = -\frac{2}{4} \log_2 \left(\frac{2}{4}\right) - \frac{2}{4} \log_2 \left(\frac{2}{4}\right) = 1$$

$$E(MP_3) = -\frac{5}{6} \log_2 \left(\frac{5}{6}\right) - \frac{1}{6} \log_2 \left(\frac{1}{6}\right) = 0.650$$

průměr (váha je dána počtem prvků v množině) entropií podmnožin.

$$\begin{aligned} E(MP, \text{Příjem}) &= \sum_{i=1}^3 \frac{|MP_i|}{|MP|} E(MP_i) = \frac{4}{14} E(MP_1) + \frac{4}{14} E(MP_2) + \frac{6}{14} E(MP_3) = \\ &= \frac{4}{14} \left(-\frac{4}{4} \log_2 \left(\frac{4}{4}\right) \right) + \frac{4}{14} \left(-\frac{2}{4} \log_2 \left(\frac{2}{4}\right) - \frac{2}{4} \log_2 \left(\frac{2}{4}\right) \right) + \\ &\quad + \frac{6}{14} \left(-\frac{5}{6} \log_2 \left(\frac{5}{6}\right) - \frac{1}{6} \log_2 \left(\frac{1}{6}\right) \right) = 0.564 \end{aligned}$$

Výsledný prohledávací strom vytvořený pomocí ID3 bude vypadat následovně:



Prohledávání prostoru verzí (Version Space Search)

Prohledávání prostoru verzí představuje soubor metod učení významných pojmu/hypotéz na základě **pozitivních** a **negativních** příkladů. Při učení se hledá takový **popis daného pojmu/objektu/hypotézy**, který **zahrnuje všechny pozitivní příklady a vylučuje všechny negativní příklady** z trénovací množiny příkladů. Trénovací množina **vždy** musí obsahovat **pozitivní i negativní** příklady. Příklad

trénovací množiny pro identifikaci pojmu míč může být následující:

1. objekt(<i>malá,červená,koule</i>)	kladný
2. objekt(<i>malá,modrá,kvádr</i>)	záporný
3. objekt(<i>velká,červená,koule</i>)	kladný
4. objekt(<i>velká,červená,krychle</i>)	záporný
5. objekt(<i>malá,modrá,koule</i>)	kladný
6. objekt(<i>malá,bilá,kvádr</i>)	záporný

Algoritmus Specific to general search

Metoda učení prohledáváním prostoru, která pracuje **od specifického k obecnějšímu**.

1. Vytvořte dvě prázdné množiny **S** (Specific) a **N** (Negative).
2. Uložte do množiny **S** první **kladný příklad**. Pro každý další příklad **p** z trénovací množiny:
 - a. je-li **p kladným příkladem**, pak pro každý pojem **s ∈ S**
 - i. jestliže pojem **s** nelze unifikovat (**s** je obecnější než **p** a **p** spadá do množiny prvků, které lze pomocí **s** sestavit) s příkladem **p**, pak jej nahraďte jeho **nejvíce specifickým zobecněním** (tak aby už vyjádřit pomocí **s** šel), které lze unifikovat s příkladem **p**,
 - ii. odstraňte z **S** všechny pojmy **s**, které jsou **více obecné** než jiné pojmy v **S**,
 - iii. odstraňte z **S** všechny pojmy **s**, které lze **unifikovat** s některým pojemem v **N**.
 - b. je-li **p záporným příkladem**, pak
 - i. odstraňte z **S** všechny pojmy **s**, které lze unifikovat (lze pomocí nich vyjádřit záporný prvek **p**) s příkladem **p**,
 - ii. přidejte příklad **p** do **N**.

Jinak řečeno s **pozitivními příklady** se výsledné řešení stává **obecnější** - přidávají se **möznosti správných řešení**. Negativní příklady naopak množinu možných řešení redukují (odstraňují ta řešení, která by správně identifikovala i špatný objekt). Průběh algoritmu na příkladu s míčem vypadá následovně, výsledkem je, že míč je **objekt(X,Y,koule)**:

	$S = \{ \}, N = \{ \}$
1. $p = \text{objekt}(\text{malá,červená,koule})$	$S = \{\text{objekt}(\text{malá,červená,koule})\}$
2. $p = \text{objekt}(\text{malá,modrá,kvádr})$	$N = \{\text{objekt}(\text{malá,modrá,kvádr})\}$
3. $p = \text{objekt}(\text{velká,červená,koule})$	$S = \{\text{objekt}(X,\text{červená,koule})\}$
4. $p = \text{objekt}(\text{velká,červená,krychle})$	$N = \{\text{objekt}(\text{malá,modrá,kvádr}), \text{objekt}(\text{velká,červená,krychle})\}$
5. $p = \text{objekt}(\text{malá,modrá,koule})$	$S = \{\text{objekt}(X,Y,\text{koule})\}$
6. $p = \text{objekt}(\text{malá,bilá,kvádr})$	$N = \{\text{objekt}(\text{malá,modrá,kvádr}), \text{objekt}(\text{velká,červená,krychle}), \text{objekt}(\text{malá,bilá,kvádr})\}$

Algoritmus General to Specific Search

Metoda učení prohledáváním prostoru, která pracuje **od obecného ke specifickému**.

1. Vytvořte dvě prázdné množiny **G** (General) a **P** (Positive) a uložte do **G nejobecnější** pojem (všechny parametry jsou vyjádřené proměnnou).
2. Pro každý další příklad **p** z trénovací množiny:
 - a. je-li **p záporným příkladem**, pak pro každý pojem $g \in G$:
 - i. jestliže pojem **g** lze unifikovat s příkladem **p**, pak jej nahraďte jeho **nejobecnější specializací**, kterou **nelze unifikovat** s příkladem **p**,
 - ii. odstraňte z **G** všechny pojmy **g**, které jsou **více specializované** než jiné pojmy v **G**,
 - iii. odstraňte z **G** všechny pojmy **g**, které nelze unifikovat s některým pojmem v **P**.
 - b. je-li **p kladným příkladem**, pak:
 - i. odstraňte z **G** všechny pojmy **g**, které nelze unifikovat s příkladem **p**,
 - ii. přidejte příklad **p** do **P**.

Jinak řečeno se **zápornými příklady** se z obecného řešení **stává konkrétnější**, protože je řešení **konkretizováno**, aby nevyhovovalo **záporným** řešením. **Kladné příklady** pak **odstraňují** ta řešení, pomocí kterých je **není možné vyjádřit** (unifikovat). Průběh algoritmu je na obrázku, výsledek je **objekt(X,Y,koule)**:

1. $p = \text{objekt(malá,červená,koule)}$	$G = \{\text{objekt}(X,Y,Z)\}, P = \{\}$
2. $p = \text{objekt(modrá,červená,kvádr)}$	$P = \{\text{objekt(modrá,červená,kvádr)}\}$
	$G = \{\text{objekt(velká,Y,Z)},$ $\quad \text{objekt}(X,\text{červená},Z), \text{objekt}(X,\text{bilá},Z),$ $\quad \text{objekt}(X,Y,\text{koule}), \text{objekt}(X,Y,\text{krychle})\}$
3. $p = \text{objekt(velká,červená,koule)}$	$\Rightarrow G = \{\text{objekt}(X,\text{červená},Z), \text{objekt}(X,Y,\text{koule})\}$
4. $p = \text{objekt(velká,červená,krychle)}$	$P = \{\text{objekt(modrá,červená,kvádr)},$ $\quad \text{objekt(velká,červená,krychle)}\}$
5. $p = \text{objekt(modrá,červená,kvádr)}$	$G = \{\text{objekt(modrá,červená,Z)},$ $\quad \text{objekt}(X,\text{červená},kvádr),$ $\quad \text{objekt}(X,\text{červená},\text{koule}),$ $\quad \text{objekt}(X,Y,\text{koule})\}$
6. $p = \text{objekt(modrá,červená,krychle)}$	$\Rightarrow G = \{\text{objekt}(X,Y,\text{koule})\}$
	$P = \{\text{objekt(modrá,červená,kvádr)},$ $\quad \text{objekt(velká,červená,krychle)},$ $\quad \text{objekt(modrá,červená,krychle)}\}$
6. $p = \text{objekt(modrá,červená,krychle)}$	$G = \{\text{objekt}(X,Y,\text{krychle})\}$
6. $p = \text{objekt(modrá,červená,koule)}$	$P = \{\text{objekt(modrá,červená,koule)},$ $\quad \text{objekt(velká,červená,koule)},$ $\quad \text{objekt(modrá,červená,koule)}\}$
6. $p = \text{objekt(modrá,červená,kvádr)}$	$G = \{\text{objekt}(X,Y,\text{kvádr})\}$

Candidate eliminations

Spojuje postupy předchozích algoritmů.

Vytvořte dvě prázdné množiny **G** (General) a **S** (Specific) a uložte do **G** nejobecnější pojem.

Uložte do množiny **S** první **kladný příklad**.

Pro každý další příklad **p** z trénovací množiny:

je-li **p kladným příkladem** pak:

odstraňte z **G** všechny pojmy **g ∈ G**, které nelze unifikovat s příkladem **p**,

pro každý pojem **s ∈ S**:

jestliže pojem **s nelze unifikovat** s příkladem **p**, pak jej nahraďte jeho **nejvíce specifickým zobecněním**, které lze unifikovat s příkladem **p**,

odstraňte z **S** všechny pojmy **s**, které jsou **více obecné** než jiné pojmy v **S**,

odstraňte z **S** všechny pojmy **s**, které nejsou **více specifické**, než některé pojmy v **G**.

je-li **p záporným příkladem**:

odstraňte z **S** všechny pojmy **s**, které lze unifikovat s příkladem **p**,

pro každý pojem **g ∈ G**:

jestliže pojem **g lze unifikovat** s příkladem **p**, pak jej nahraďte jeho **nejvíce zobecněnou specializací**, kterou **nelze unifikovat** s příkladem **p**,

odstraňte z **G** všechny pojmy **g**, které jsou **více specifické** než jiné pojmy v **G**,

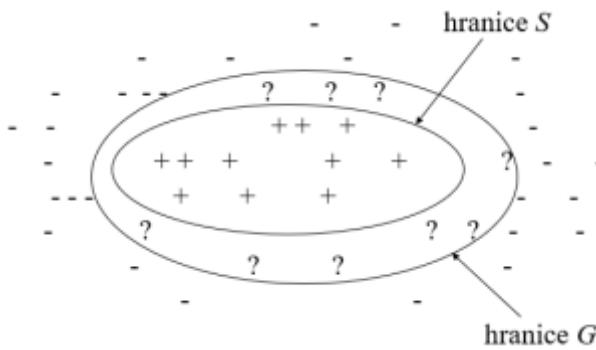
odstraňte z **G** všechny pojmy **g**, které nejsou více obecné než některé pojmy v **S**.

Jestliže **G = S** a obě množiny přitom **obsahují jediný pojem**, pak je výsledkem učení právě tento pojem. Příklad tohoto algoritmu, výsledek je **objekt(X,Y,koule)**:

		$G = \{\text{objekt}(X,Y,Z)\}, S = \{\}$
1.	$\text{p} = \text{objekt(malá,červená,koule)}$	$S = \{\text{objekt}(malá,červená,koule)\}$
2.	$\text{p} = \text{objekt(malá,modrá,kvádr)}$	$G = \{\text{objekt(velká,Y,Z)}, \text{objekt}(X,\text{červená},Z), \text{objekt}(X,\text{bilá},Z), \text{objekt}(X,Y,\text{koule}), \text{objekt}(X,Y,\text{krychle})\}$
		$\Rightarrow G = \{\text{objekt}(X,\text{červená},Z), \text{objekt}(X,Y,\text{koule})\}$
3.	$\text{p} = \text{objekt(velká,červená,koule)}$	$S = \{\text{objekt}(X,\text{červená},\text{koule})\}$
4.	$\text{p} = \text{objekt(velká,červená,krychle)}$	$G = \{\text{objekt}(malá,\text{červená},Z), \text{objekt}(X,\text{červená},\text{kvádr}), \text{objekt}(X,\text{červená},\text{koule}), \text{objekt}(X,Y,\text{koule})\}$
		$\Rightarrow G = \{\text{objekt}(X,Y,\text{koule})\}$
5.	$\text{p} = \text{objekt(malá,modrá,koule)}$	$S = \{\text{objekt}(X,Y,\text{koule})\}$
6.	$\text{p} = \text{objekt(malá,bilá,kvádr)}$	$G = \{\text{objekt}(X,Y,\text{koule})\}$

Význam a vztah množin **S** a **G** je na obrázku níže. Každý pojem, který **by byl obecnější** než nějaký pojem v **G**, **by zahrnoval některé negativní příklady**, každý pojem, který **by byl specifický** než nějaký pojem v **S**, **by vylučoval některé**

pozitivní příklady. Výsek s otazníky značí objekty, které nejsou v trénovací množině, ale na základě výsledků algoritmů **prohledávání prostoru verzí** je poté lze identifikovat, např.: **míč je kulatý objekt, na jehož barvě, ani velikosti nezáleží.**



Rozpoznávání a klasifikace obrazů (Pattern Recognition and Classification):

Pro rozpoznávání obrazů existují různé algoritmy, které pracují s jedním z následujících popisů:

- **Příznakový** (popis vektory číselných příznaků): využívá **statických informací o příznacích** obrazů obsažených v množině trénovacích dat. Jde o tzv. **statické příznakové rozpoznávání**.
- **Strukturální/syntaktický** (popis strukturálními prvky, tzv. primitivy): využívá **vztahy mezi příznaky** obrazů rozpoznávaných objektů.

<u>příznakový popis</u>	objekt	<u>strukturální popis</u>
počet segmentů	4	
počet horizontálních segmentů	2	
počet vertikálních segmentů	2	
počet diagonálních segmentů	0	
vektor příznaků	(4,2,2,0)	
počet segmentů	3	
počet horizontálních segmentů	1	
počet vertikálních segmentů	0	
počet diagonálních segmentů	2	
vektor příznaků	(3,1,0,2)	

Zásadním předpokladem úspěšného rozpoznávání je **výběr relevantních příznaků**, resp. **primitiv**.

Příznakové rozpoznávání

U příznakového rozpoznávání pracujeme s:

- n-rozměrnými **číselnými vektory** příznaků, které popisují rozpoznávané objekty,
- **množinou tříd**, do kterých rozpoznávané objekty chceme zařadit,
- **trénovací množinu**, která je tvořena **dvojicemi** n-rozměrný číselný **vektor** a k němu **třídou**, do které spadá.

Cílem je poté zařadit **libovolný** vektor příznaků do **jedné** z tříd - **klasifikovat** jej.

Metody **příznakového rozpoznávání/klasifikace** vycházejí z předpokladu, že obrazy objektů nebo jevů **stejných tříd** tvoří v **n rozměrném** obrazovém prostoru **shluky**. Tyto shluky mohou být od sebe **zřetelně oddělitelné (separable)**, nebo se mohou **prolínat** a pak jsou **neoddělitelné (inseparable)**. Systémy, které se na trénovací množině naučí obrazy rozpoznávat a poté **klasifikují nové** obrazy, se nazývají **klasifikátory**.

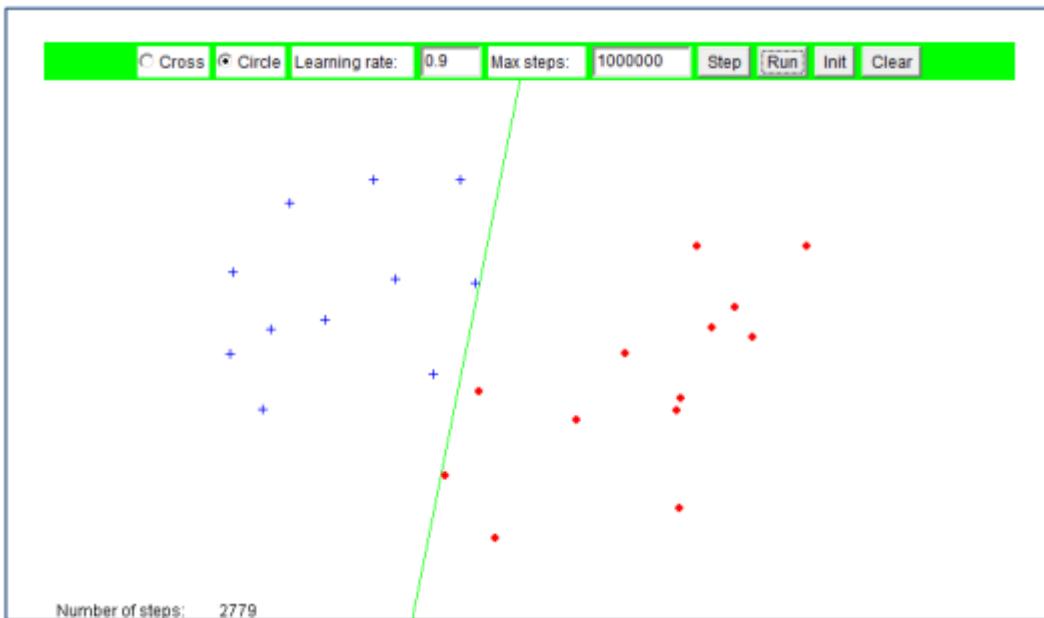
Dichotomie

Jedná se o **klasifikaci** do **dvou tříd**. V praxi je **dichotomie** poměrně **častá**, obecně jde klasifikovat do **n** tříd, což spočívá v nalezení **křivek/ploch/k-rozměrných útvarů**, které obrazy jednotlivých tříd od sebe **oddělují**. Toho se zajišťuje pomocí **diskriminačních** funkcí (pro každou třídu jedna), které obraz ohodnocují. **Obraz** je pak **umístěn/klasifikován** do třídy, jejíž **diskriminační funkce** má **největší** hodnotu (je nutné pro každý obraz vypočítat hodnoty všech diskriminačních funkcí). U dichotomie stačí jedna diskriminační funkce (respektive **2**, které od sebe **odečítáme**) a obrazy objektů klasifikujeme na základě **znaménka** do dvou tříd (kladná a záporná).

Algoritmus určení **lineárního klasifikátoru** pro dichotomii (diskriminační funkce):

1. Vynulujte vektor **vah**.
2. Nastavte indikátor změny **modif = false**.
3. Pro každý **vektor** z trénovací množiny, který **není správně klasifikován** (diskriminační funkce vrací jiné znaménko, než je v trénovacím příkladě) **upravte vektor vah** (kterým je vektor klasifikován) a nastavte **modif = true**.
4. Pokud došlo k úpravě vektoru **vah** (**modif = true**), tak se vraťte na bod 2.

Výsledkem bude rozdělení prostoru dle následujícího obrázku pro 2D.



Jeden klasifikátor klasifikující do **R tříd** může být nahrazen **R klasifikátory** pro dichotomii **naučených** na klasifikaci **patří/nepatří** do třídy r , $r \in \langle 1, R \rangle$.

Etalony

Jedná se o **těžiště shluků obrazů** jednotlivých tříd. Obraz je klasifikován do třídy, jejímuž **etalonu má nejblíže** (nejmenší vzdálenost v prostoru). Tato klasifikace však opět pracuje na principu klasifikace podle diskriminačních funkcí.

Rozpoznávání obrazů reprezentovaných neoddělitelnými třídami obrazů

V případě **neoddělitelných tříd** obrazů již nelze rozhodnout, že **obraz x** patří do třídy r , ale lze konstatovat, že obraz x patří do třídy r s pravděpodobností $P(r|x)$. Úkolem trénovacích algoritmů je nalézt takový klasifikátor, který bude klasifikovat obrazy s co největší pravděpodobností, respektive minimalizovat ztrátu, která vzniká klasifikací do nesprávné třídy. Využívá se k tomu ztrátová matice, ztráta pro **správnou klasifikaci obrazu je 0** a pro **chybnou 1**. Na základě ztrát se matice zjednoduší na obrázek vpravo.

$$\lambda = \begin{bmatrix} \lambda(1,1) & \lambda(1,2) & \dots & \lambda(1,R) \\ \lambda(2,1) & \lambda(2,2) & \dots & \lambda(2,R) \\ \vdots & \vdots & \vdots & \vdots \\ \lambda(R,1) & \lambda(R,2) & \dots & \lambda(R,R) \end{bmatrix} \quad \lambda = \begin{bmatrix} 0 & 1 & \dots & 1 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & \dots & 0 \end{bmatrix}$$

Strukturální rozpoznávání

Jedná se o rozpoznávání obrazů na základě jeho popisu pomocí **primitiv**. Existují dva základní přístupy ke strukturálnímu rozpoznávání obrazů:

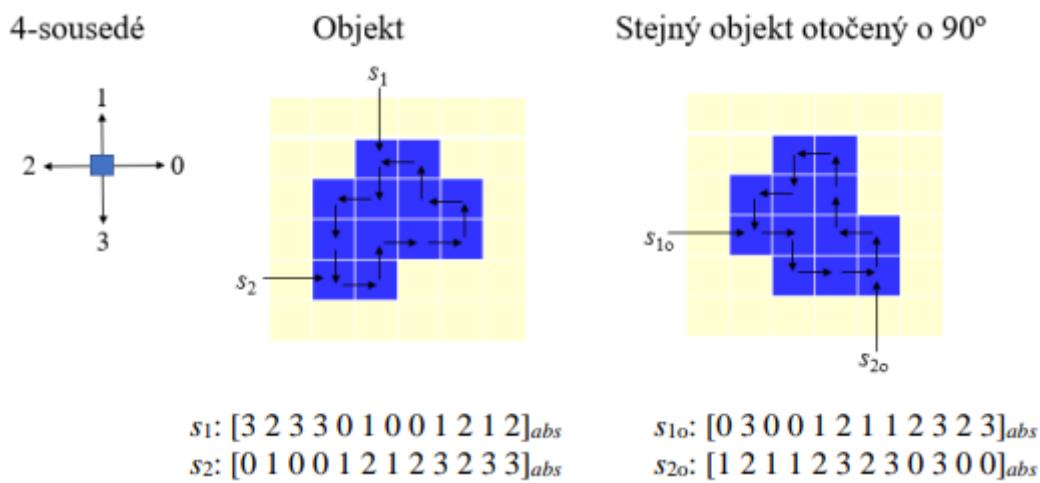
- rozpoznávání obrazů pomocí **gramatik - syntaktické rozpoznávání** (syntactic recognition)
- rozpoznávání obrazů **porovnáním se vzory** uloženými v databázi (template matching) - **neučili jsme se**.

Syntaktické rozpoznávání klasifikuje obrazy do **R** tříd pomocí **syntaktické analýzy**. Obrazy, tj. **řetězcové popisy objektů** nebo jevů, jsou přitom chápány jako **slova** a množina **primitiv** jako množina **terminálních symbolů**. Pokud pak gramatika generuje **všechna slova/obrazy**, která reprezentují obrazy třídy **r**, a negeneruje **zádné slovo**, která reprezentuje obraz **jiné třídy** (jinak řečeno všechny gramatiky generují vzájemně **různá slova**), lze klasifikaci převést na problém určení gramatiky, která jako jediná ze všech **R** gramatik generuje rozpoznávané slovo/obraz.

Na **úspěšnost** strukturálního/syntaktického rozpoznávání má velký vliv **výběr primitiv** (malá množina primitiv má malou vyjadřovací sílu, velká množina může být nezvládnutelná při učení).

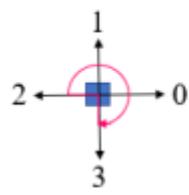
Freemanův řetězcový kód

Používá se ke strukturálnímu popisu obrysu objektu (obrazu). Za primitiva popisující obrys objektu považujeme **směry sousedů**. **Problémy** při popisu objektů tímto způsobem je jejich **natočení** a **startovací pozice** popisu (popis je invariantní vůči posuvu), což jej činí prakticky nepoužitelným.



Problémy lze eliminovat následovně:

- **natočení**: řešením je popis pomocí relativního (diferenciálního) kódu, který místo primitiv používá **rozdílů/diferencí natočení** mezi dvěma sousedními primitivy (**následující – aktuální**). Pro určení těchto rozdílů se postupuje v **opačném směru** (tj. po směru hodinových ručiček, overflow 2-3 = 3, 1-3 = 2, ... pokud je **první číslo menší**, je nutné k výsledku **přičíst 4**). Na obrázku lze po provedení postupu vidět, že **diferenční kódy** pro s1 a s10 jsou stejně jako jsou stejné diferenční kódy s2 a s20.
- **startovací pozice**: řešení spočívá v tom, že řetězec **relativního** (diferenciálního) **kódu** rotujeme tak, aby řetězec číslic po převodu na číslo bylo **číslo největší** (tj. zleva obsahovalo největší číslice). Tomuto číslu se říká



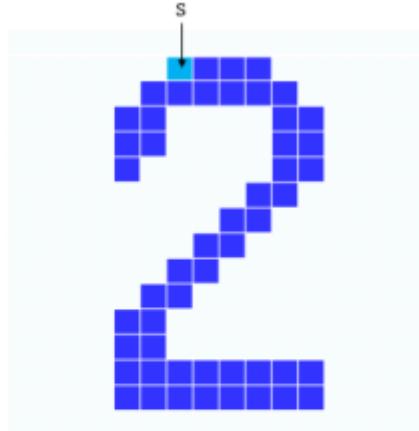
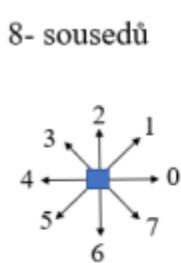
$2 - 3 \rightarrow 3$

$s_1: [3 \ 2 \ 3 \ 3 \ 0 \ 1 \ 0 \ 0 \ 1 \ 2 \ 1 \ 2]_{abs}$
 $s_1: [3 \ 1 \ 0 \ 1 \ 1 \ 3 \ 0 \ 1 \ 1 \ 3 \ 1 \ 1]_{dif}$
 $s_2: [0 \ 1 \ 0 \ 0 \ 1 \ 2 \ 1 \ 2 \ 3 \ 2 \ 3 \ 3]_{abs}$
 $s_2: [1 \ 3 \ 0 \ 1 \ 1 \ 3 \ 1 \ 1 \ 3 \ 1 \ 0 \ 1]_{dif}$
 $s_{1o}: [0 \ 3 \ 0 \ 0 \ 1 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ 3]_{abs}$
 $s_{1o}: [3 \ 1 \ 0 \ 1 \ 1 \ 3 \ 0 \ 1 \ 1 \ 3 \ 1 \ 1]_{dif}$
 $s_{2o}: [1 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ 3 \ 0 \ 3 \ 0 \ 0]_{abs}$
 $s_{2o}: [1 \ 3 \ 0 \ 1 \ 1 \ 3 \ 1 \ 1 \ 3 \ 1 \ 0 \ 1]_{dif}$

číslo tvaru - shape number.

Číslo tvaru (shape number) je **invariantní** vůči **posunutí, natočení** objektu (pro 4 sousedy pouze **po 90°**) a **startovacímu bodu** popisu, je však **závislé na velikosti objektu**.

Freemanův řetězcový kód existuje i pro 8 sousedů, viz obrázek:



Řetězcový kód pro 8 sousedů:

$[5566121000066655555666000000024444432111112233444]_{abs} =$
 $[0103177000600700001002000002200007770000010101001]_{dif}$

Číslo tvaru:

$[77700000101010010103177000600700000100200000220000]$

Číslo tvaru $[3 \ 1 \ 1 \ 3 \ 1 \ 0 \ 1 \ 1 \ 3 \ 0 \ 1 \ 1]$ popisující objekt výše by generovala například **gramatika** s těmito přepisovacími pravidly:

- $S \rightarrow 3N$,
- $N \rightarrow 0N$,
- $N \rightarrow 1N$,
- $N \rightarrow 3N$,
- $N \rightarrow 1$

Tato gramatika by však **generovala i další** řetězce, které by daný objekt **nepopisovaly**, a proto k rozpoznávání by byla zřejmě **nepoužitelná**. Nalezení

relevantních přepisovacích pravidel (tj. určení správné gramatiky) je mnohem **náročnější** než učení příznakových klasifikátorů nebo trénování neuronových sítí a dnes se tak moc nepoužívá.

Učení bez učitele

Učení bez učitele spočívá v **hledání podobností** mezi příklady **trénovací množiny** a v zařazování příkladů s **podobnými charakteristikami do skupin**. Umělý systém přitom **nedostává žádnou informaci** o správnosti klasifikace a jedinou informací, kterou má, resp. může mít je **počet skupin**, do kterých má příklady z trénovací množiny zařazovat/klasifikovat.

Příklady trénovací množiny jsou téměř vždy představované **číselnými vektory** příznaků klasifikovaných objektů či dějů. Metody **učení bez učitele** jsou pak založeny na předpokladu, že tyto vektory, resp. body, které specifikují v příslušném **n rozměrném** obrazovém prostoru **shluky**. Tyto **shluky** mohou představovat velmi **rozmanité n rozměrné útvary**, například v několika souřadnicích může jít o zcela kompaktní shluky, zatímco v jiných souřadnicích mohou být značně rozptýlené.

On-Policy

TODO

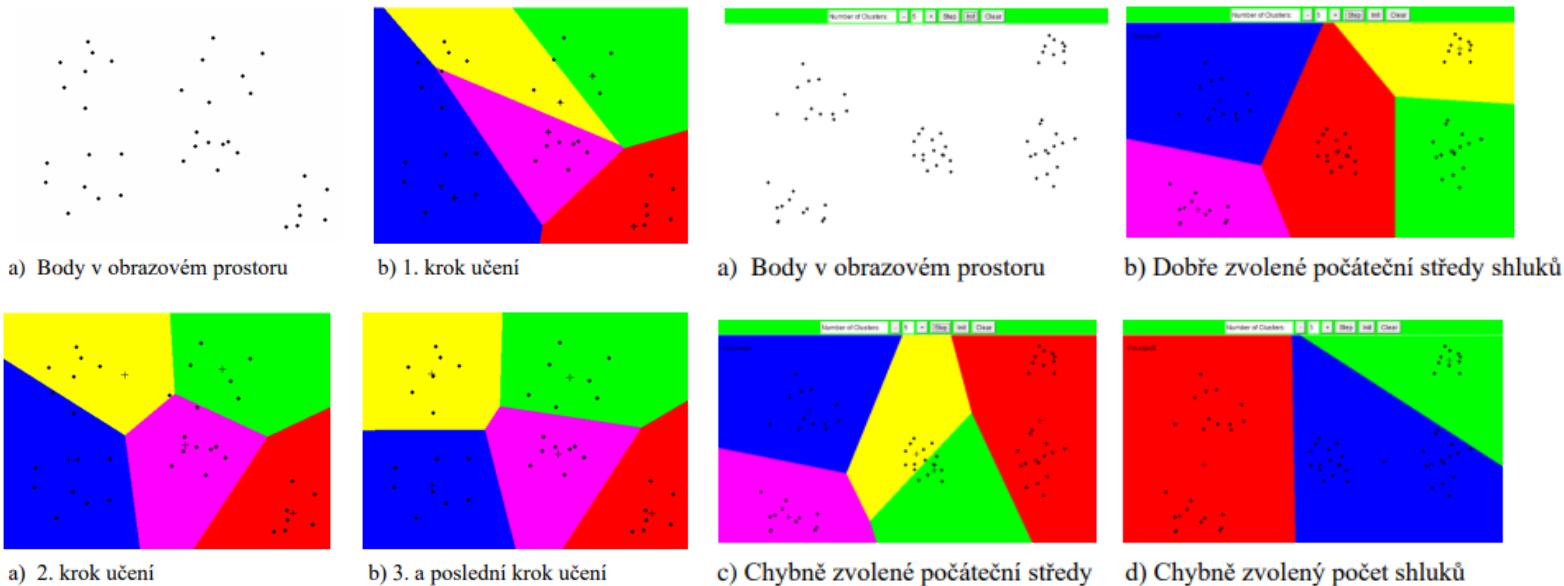
Off-Policy

TODO

k-means clustering

Algoritmus **klasifikuje** příklady (číselné vektory) z trénovací množiny do předem daného počtu **k** shluků. Algoritmus **zařazuje** vstupní vektor do toho **shluku**, k jehož **středu/těžišti má nejkratší vzdálenost**. Vstupem algoritmu je počet shluků **k** (musí být menší než počet vektorů) a průběh učení je následující:

1. Náhodně určí **k** rozdílných vektorů (často je vybere z trénovací množiny), které považuje za **středy (těžiště)** shluků.
2. Zařadí všechny vektory trénovací množiny do **příslušných shluků**, dle použité **metriky** (nejčastěji se používá asi **Euklidovská** vzdálenost - délka úsečku mezi body, ale lze použít např. **Hammingovu** vzdálenost - je nejmenší počet pozic, na kterých se řetězce stejně délky daného kódu liší)
3. **Přepočítá středy** všech shluků.
4. Pokud se pozice ani jednoho středu nezměnila (tj. vektory byly opakovaně zařazeny do stejných shluků), algoritmus končí. Jinak pokračuje bodem 2.



Posilované učení

Posilované učení se od učení s učitelem odlišuje tím, že systém **ohodnocuje** své akce na základě **penalizací** či **odměn** získaných v **koncových stavech** a na základě **svých hodnocení** stavů/akcí získaných **vlastními** předchozími **zkušenostmi**.

Policy-only learning

Princip tohoto algoritmu je založený na tom, že z **každého uzlu** grafu vedou **maximálně 2** (koncové 1) **cesty** (hrany). každý uzel obsahuje **schránku s kameny** (analogie pravděpodobnosti) dvojí barvy - **bílé** a **černé** (pokud je počet bílých kamenů větší, je pravděpodobnější výběr cesty vlevo, naopak je to u většího počtu

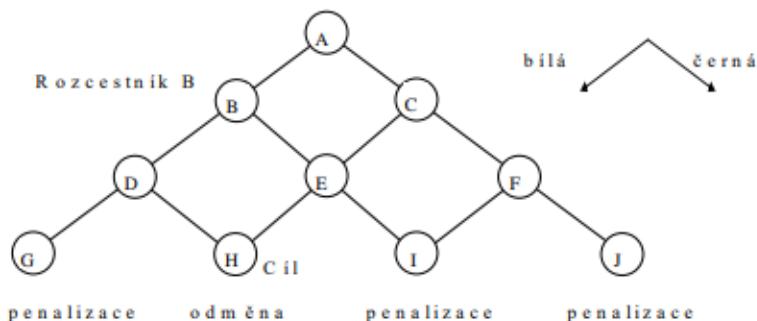
$$P_{vlevo} = \frac{N_{bílé}}{N_{bílé} + N_{černé}}, \quad P_{vpravo} = \frac{N_{černé}}{N_{bílé} + N_{černé}}$$

(černých kamenů).

Na začátku učení obsahuje schránka každého rozcestníku **stejný a dostatečný počet** černých a bílých kamenů. Učení pak probíhá na tomto principu, že se provádí náhodné procházky (náhodnost je dána pravděpodobností výběru cesty v uzlech).

Pokud výsledná cesta:

- **skončí v cílovém stavu** (na obrázku uzel H), je do schránek na této cestě **přidán** kámen odpovídající barvy podle výběru směru - **odměna**.
Pravděpodobnost výběru této cesty **se zvyšuje**.
- **skončí mimo cílový stav** (na obrázku uzly G, I, J), je ze schránek **odebrán** kámen odpovídající barvy podle výběru směru - **penalizace**.
Pravděpodobnost výběru této cesty **se snižuje**.



Po ukončení učení vybírá umělý systém cestu **pouze porovnáním počtu kamenů**, tj. je-li ve schránce rozcestníku **více bílých kamenů**, pokračuje **levou** cestou **jinak** pokračuje cestou **pravou**.

Počet cest z uzlu lze jednoduše **rozšířit** přidáním více kamenů **různých barev** (rozdělení počáteční pravděpodobnosti mezi více cest).

Metoda může mít 2 zásadní **problémy**:

- V případě obecného grafu se může **vracet do již dříve vyšetřovaných uzelů** (například v bludišti), může se teoreticky při učení zacyklit.
- Všechny akce provedené na jedné **cestě se hodnotí stejnými vahami**, přestože správná ohodnocení mohou být značně rozdílná (Např. u řešení, které končí v cílovém stavu, je vložen do každé schránky kámen podporující tuto cestu, i když tato cesta nemusí být výhodná - zbytečně dlouhá atd., lepší by bylo, kdyby odměna/penalizace měla vždy stejně velkou hodnotu, která by byla rozdělena mezi uzly na cestě).

Metoda TD learning (On-Policy)

Metoda je založená na **náhodných procházkách**, během kterých ohodnocuje jednotlivé stavy **s**, **stav s je vždy bezprostředním předchůdcem stavu s'**. Dochází tedy k **šíření hodnot** ze stavů s **odměnou/penalizací**. Šíření ohodnocení je dáno následujícím vzorcem:

$$\text{ohodnocení}(s) = \text{ohodnocení}(s) + \alpha * (r(s') + \gamma * \text{ohodnocení}(s') - \text{ohodnocení}(s)),$$

kde:

- α : koeficient **učení**,
- γ : koeficient určující **vliv** ohodnocení stavu **s'** na ohodnocení předcházejícího stavu **s**,
- $r(s')$: odměnu (reward) za dosažení stavu **s'**,
- **ohodnocení(s)**: ohodnocení (utility) stavu **s** při použití dané **strategie pohybu**.

Pro přecházení mezi stavy lze použít různé strategie:

- **Pravděpodobnosti** přechodů mezi sousedními stavami jsou **stejné** a do sousedních stavů přechází **zcela náhodně (random policy)**.
- Pravděpodobnosti jsou **různé** (mohou být například dány **aktuálními hodnotami sousedních stavů**, tj. čím je **vyšší ohodnocení** sousedního stavu, tím je **vyšší pravděpodobnost** přechodu do tohoto stavu).
- Pravděpodobnost jednoho přechodu je **jedničková** a pravděpodobnosti ostatních **nulové**, což je **extrémní případ** předchozího stavu (**greedy policy**).
- **Kombinace předchozích** případů, kdy s pravděpodobností danou parametrem ϵ se použije random policy a s pravděpodobností ($1-\epsilon$) se použije **greedy policy** (ϵ -greedy policy).

Princip učení:

1. Zvolte hodnoty koeficientů α a γ ($0 < \alpha \leq 1$; $0 < \gamma \leq 1$) a **vynulujte** ohodnocení všech stavů. Dále **vynulujte počítadlo** procházkę p a nastavte

jejich **maximální** počet na **pmax**. Nastavte **start** → **s**.

2. Generujte nový stav **s'** s použitím některé strategie.
3. Přeypočítejte novou hodnotu stavu s pomocí vztahu pomocí vzorce výše.
4. Je-li stav **s'** cílovým stavem, pak inkrementuj počet procházk a **start** → **s**, jinak **s'** → **s**.
5. Je-li $p < p_{max}$, pak se vraťte na bod 2.

1	0	0	0
2	0	0	0.1
3	0	-1	1
1	2	3	

1	-0.135254	-0.185655	-0.086196
2	-0.205577	-0.419297	0.231171
3	-0.220843	-1	1

Příklad, vlevo po 1. kroku učení, vpravo po naučení:

Po naučení se již přechází z libovolného necílového stavu do jeho sousedního stavu, který má **nejvyšší hodnotu** (ale musí být **stejnou nebo lepší** než hodnota aktuálního stavu - to může způsobit problém uváznutí, z nějaké stavu nemusí být dosažitelný žádný jiný stav, lze řešit snížením koeficientu učení α)

Q learning (Off policy)

Metoda je podobná metodě **TD learning**. Tato metoda **místo hodnocení stavů hodnotí akce v těchto stavech** a k tomuto hodnocení používá vztah:

$$Q(s, a) = Q(s, a) + \alpha * (r(s') + \gamma * \max Q(s', a') - Q(s, a)),$$

kde:

- α : koeficient učení,
- γ : koeficient určující vliv ohodnocení stavu **s'** na ohodnocení předcházejícího stavu **s**,

- $r(s')$: odměnu (reward) za dosažení stavu s ,
- $Q(s,a)$: označuje ohodnocení akce a provedené ve stavu s .
- $\max Q(s',a')$: označuje maximální hodnotu z ohodnocení všech akcí a' , které je možné provést ve stavu s' .

Metoda je aktivní metodou, tj. bez předem dané strategie výběru stavu s' . Princip učení:

1. Zvolte hodnoty koeficientů α a γ ($0 < \alpha \leq 1$; $0 < \gamma \leq 1$) a vynulujte ohodnocení $Q(s,a)$ všech akcí a ve všech stavech s . Dále vynulujte počítadlo procházek $p = 0$ a nastavte jejich maximální počet p_{max} . Nastavte start $\rightarrow s$.
2. Vyberte akci a , která povede k přechodu ze stavu s do stavu s' .
3. Vypočítejte novou hodnotu vybrané akce a ve stavu s pomocí vztahu výše.
4. Je-li stav s' cílovým stavem, pak inkrementuj počítadlo procházek a nastav start $\rightarrow s$, jinak nastav $s' \rightarrow s$.
5. Je-li $p < p_{max}$, pak se vrátíte na bod 2.

Ohodnocení akcí může vypadat následovně:

	1	0.12	0.22	0.41	0.18	
	0.12		0.36		0.59	
1	0.25		0.27		0.20	
2	0.31	-0.1	0.45	-0.22		
2	-0.26		-0.57		0.77	
3	0					
3	-1	-1	-1	1	1	
	1		2		3	

Po naučení se přechází již na základě získaného ohodnocení akcí (v tomto případě přechodů)

SARSA (On-Policy)

Metoda sarsa je přístup, který také ohodnocuje akce, ale k jejich výběru používá nějakou strategii π , tzn. místo hledání maxima z možných následujících akcí se používá přímo akce $Q(s', a')$ vybraná také strategií π . Jde tedy o postup s, a, r', s', a' (SARSA). Hlavní funkce pro aktualizaci Q-hodnoty závisí na aktuálním stavu s , akci a , kterou agent zvolí, odměně r , kterou agent dostane za volbu této akce a , stavu s' , do kterého agent vstoupí po provedení akce a a nakonec další akci a' , kterou agent zvolí ve svém novém stavu.

28. Principy modelování a simulace systémů (systémy, modely, simulace, algoritmy řízení simulace).

Systémy

Systém je soubor elementárních částí (prvků systémů), které mají mezi sebou určité vazby. Můžeme je dělit na:

- **Reálné systémy**
- **Nereálné systémy** - Fiktivní, ještě neexistující.
- **Spojité** - Všechny prvky mají **spojité chování**.
- **Diskrétní** - Všechny prvky mají **diskrétní chování**.
- **Kombinované** - Obsahuje **spojité i diskrétní prvky**.
- **Deterministické** - Všechny prvky jsou **deterministické**.
- **Nedeterministické** - **Alespoň jeden** prvek s **nedeterministickým chováním**.

Formálně se jedná o dvojici $S = (U, R)$, kde:

- $U = \{u_1, u_2, \dots, u_n\}$ je univerzum - konečná **množina prvků** systému. Každý prvek je navíc tvořen dvojicí (X, Y) , $u = (X, Y)$, kde:
 - X je množina všech **vstupních proměnných**.
 - Y je množina všech **výstupních proměnných**.
- R - Množina všech **propojení** (relací), která je podmnožinou **kartézských součinů vstupů a výstupů** jednotlivých prvků.

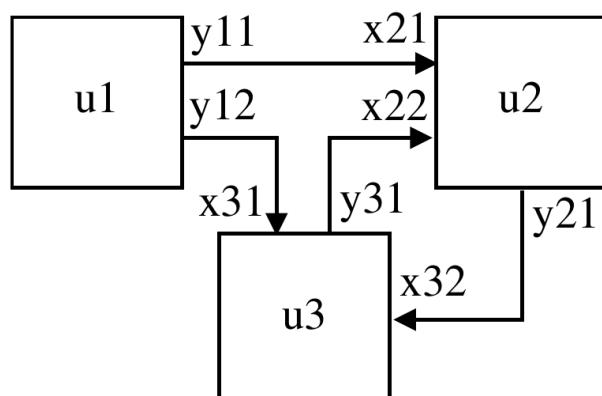
$$R = \bigcup_{i,j=1}^N R_{ij}$$

Propojení prvku u_i s u_j :

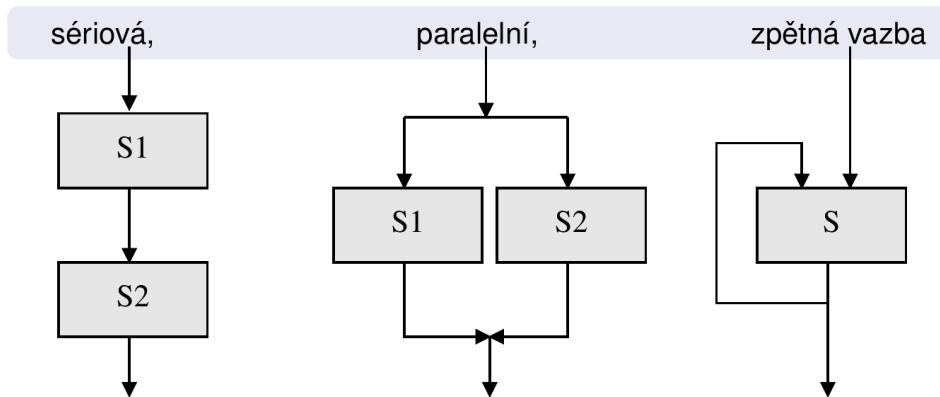
$$R_{ij} \subseteq Y_i \times X_j$$

Příklad formálně definovaného systému může vypadat následovně:

$$\begin{aligned} U &= \{u_1, u_2, u_3\} \\ R &= \{(y_{11}, x_{21}), (y_{12}, x_{31}), (y_{31}, x_{22}), (y_{21}, x_{32})\} \end{aligned}$$

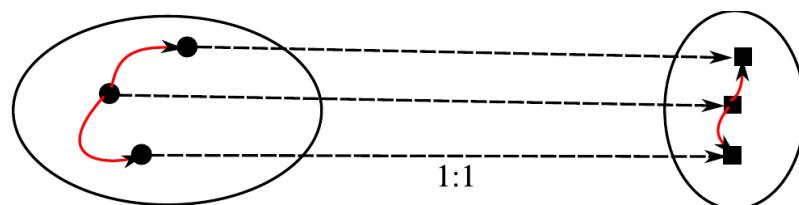


Vazba mezi prvky může být **sériová**, **paralelní** a nebo **zpětná**.

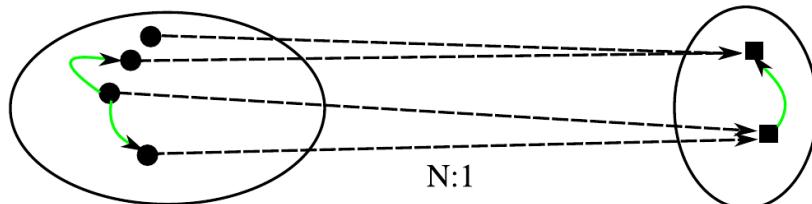


Vztahy mezi systémy

- **Izomorfní systémy** - Systémy **S1 = (U1, R1)** a **S2 = (U2, R2)** jsou izomorfní pokud mezi nimi **existuje bijekce**:
 1. Prvky **U1** lze **vzájemně jednoznačně přiřadit** **U2** (**bijektivní zobrazení - 1:1**).
 2. Prvky **R1** lze **bijektivně** zobrazit na **R2** se stejně **orientovanými vztahy** na prvky univerz.



- **Homomorfní systémy** - Je základním **principem modelování**, systémy jsou homomorfní pokud mezi nimi **existuje surjekce**:
 1. Prvkům **U1** je možné přiřadit **jednoznačně** prvky **U2** (**surjektivní zobrazení - N:1**).
 2. Prvkům **R1** je možné jednoznačně přiřadit prvky **R2** se **stejně orientovanými vztahy** s univerzity.



Chování systémů

Každému časovému průběhu **vstupních proměnných** přiřazuje časový průběh **výstupních proměnných**. Je dán vzájemnými interakcemi mezi prvky. Jinak

řečeno jedná se **způsob, jakým systém převádí vstupy na výstupy**. Jedná se o zobrazení:

Ekvivalence chování systémů - Pokud stejné podněty u obou vyvolají stejné

$$\chi : [\sigma_i(S)]^T \rightarrow [\sigma_o(S)]^T$$

kde:

- $[A]^T$ je množina všech zobrazení T do množiny A ,
- $\sigma_i(S)$ je vstupním prostorem systému S ,
- $\sigma_o(S)$ je výstupním prostorem systému S .

reakce, tak mají ekvivalentní chování.

Modely

Model je **napodobenina** systému jiným systémem. Reprezentuje znalostí, které máme o systému. Klasifikace:

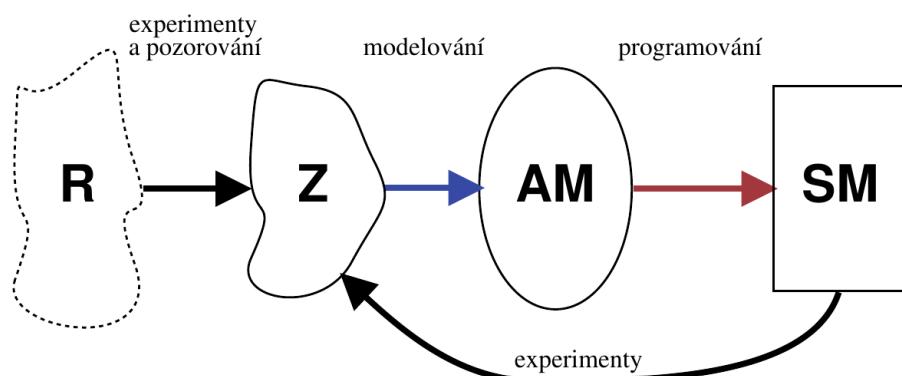
- fyzikální modely, matematické modely - přírodní zákony jsou matematické modely (Ohmův zákon: $U = R*I$).

Modelování

Vytváření modelů systému na základě znalostí, které o nich máme. Proces modelování je následující:

1. Experimenty a pozorování reality (někdy není možné, modelujeme a simulujeme neexistující věc),
2. Zisk znalostí o modelovaném systému,
3. Tvorba **abstraktního modelu** - formování zjednodušeného popisu systému,
4. Tvorba **simulačního modelu** - zápis abstraktního modelu programem,
5. **Verifikace a validace** - Ověřování správnosti modelu.
6. **Simulace** - experimentování se simulačním modelem,
7. Analýza a interpretace získaných výsledků, což vede zpět na bod 2 a celý proces lze opakovat např. dokud nejsme spokojení s výsledkem.

Realita → Znalosti → Abstraktní model → Simulační model

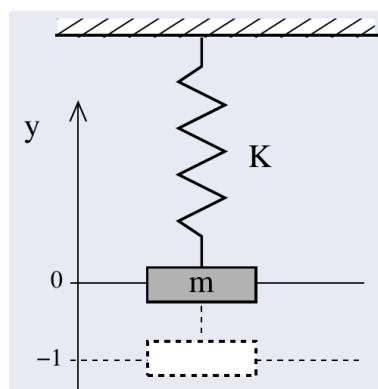


Verifikace a validace modelů

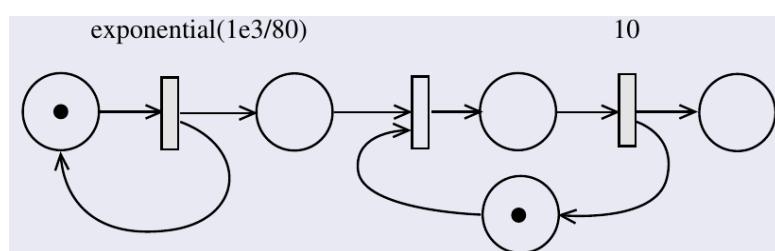
- **Verifikace modelu** - Ověřujeme **korespondenci abstraktního a simulačního modelu**, tj. izomorfní vztah mezi AM a SM. Předchází vlastní etapě simulace.
- **Validace modelu** - Snažíme se dokázat, že **skutečně pracujeme s modelem adekvátním modelovanému systému**. Velmi obtížné a nelze absolutně dokázat. Pokud chování modelu neodpovídá předpokládanému chování, musí model modifikovat, nebo nalézt příčinu odchylky. Je nutné neustále porovnávat informace, které o modelu máme a které získáváme simulací.

Klasifikace modelů

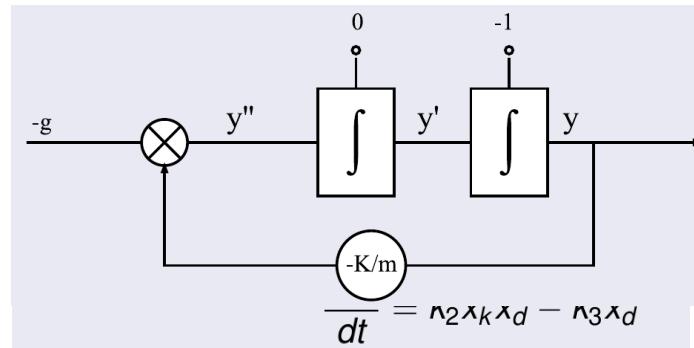
- **Konceptuální** - Jejich komponenty **nebyly** (zatím) **přesně popsány** ve smyslu teorie systémů. Obvykle se používají v **počáteční fázi** modelování pro ujasnění souvislostí a **komunikaci v týmu**. Mají formu **textu** nebo **obrázku**.



- **Deklarativní** - Popis **přechodů** mezi **stavy** systému. Model je definován **stavy a událostmi**, které **způsobí přechod** z jednoho do druhého za jistých podmínek. Vhodné především pro **diskrétní modely**. Obvykle zapouzdřeny do objektů (**konečné automaty**, **petriho sítě**, **událostmi řízené systémy s kalendářem**, **Markovovy modely**, ...).



- **Funkcionální** - Grafy zobrazující **funkce a proměnné**. Bud' je uzel grafu proměnná nebo funkce (**systémy hromadné obsluhy, bloková schémata systémová dynamika**).



- **Popsané rovnicemi (constraint)** - Rovnice (**algebraické, diferenciální, diferenční**), jedná se např. o rovnice kyvadla, RC článku,
- **Prostorové (spatial)** - Rozdělují systém na prostorově menší ohrazené podsystémy (**Parciální diferenciální rovnice, celulární automaty, L-systémy, N-body problém**).
- **Multimodely** - Je složen z modelů různého typu, které jsou obvykle heterogenní (spojité + diskrétní, spojité + fuzzy, HLA).

Simulace

Proces získávání **nových znalostí** o systému pomocí **experimentování s jeho modelem**.

Výhody

- cena,
- rychlosť,
- bezpečnosť,
- někdy jediný způsob (srážky galaxií)
- atd.

Problémy

- kontrola validity (nemusí být validní a nepoznáme to),
- náročnost na vytváření
- výpočetní náročnost,
- nepřesnost numerického měření,
- problémy stability numerických metod
- atd.

Postup

Opakované řešení modelu, experimentování s ním. Opakuje se, dokud nezískáme

dostatek informací o chování nebo dokud **nenajdeme parametry** pro které má systém **žádané chování**. Jeden simulační cyklus vypadá následovně:

- Nastavení hodnot parametrů a počátečního stavu modelu.
- Zadání vstupních podnětů z okolí při simulaci.
- Vyhodnocení výstupních dat (informace o chování systému).

Typy simulace

- **Podle popisu modelu:**
 - **Spojitá/diskrétní/kombinovaná**
 - Kvalitativní/kvantitativní
- **Podle simulátoru**
 - Na **analogovém/číslicovém** počítači, fyzikální
 - **Real-Time** simulace
 - **Paralelní a distribuovaná** simulace

Analytické řešení modelů

Popis chování modelu matematickými vztahy a jeho matematické řešení. Vhodné pro jednoduché systémy nebo zjednodušený popis složitých. Dosazením korektních hodnot získáme řešení (např. model volného pádu ve vakuu - v atmosféře už by byl ale složitější).

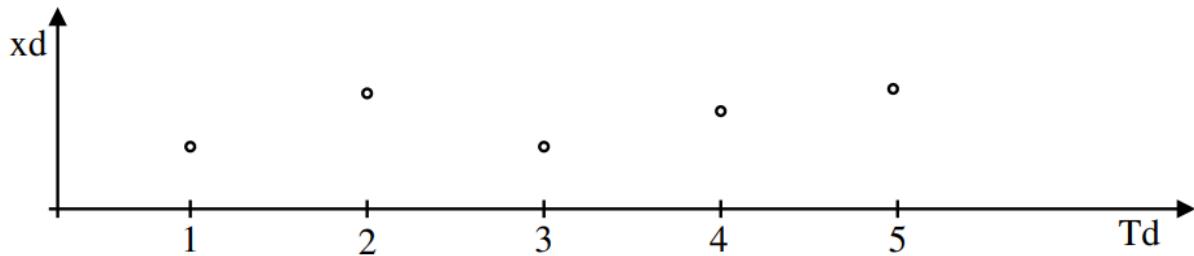
Čas

- **Reálný** - Ve kterém probíhá skutečný děj.
- **Modelový** - Časová osa modelu (nemusí být synchronní s reálným).
- **Strojový** - Čas CPU spotřebovaný na výpočtu programu.

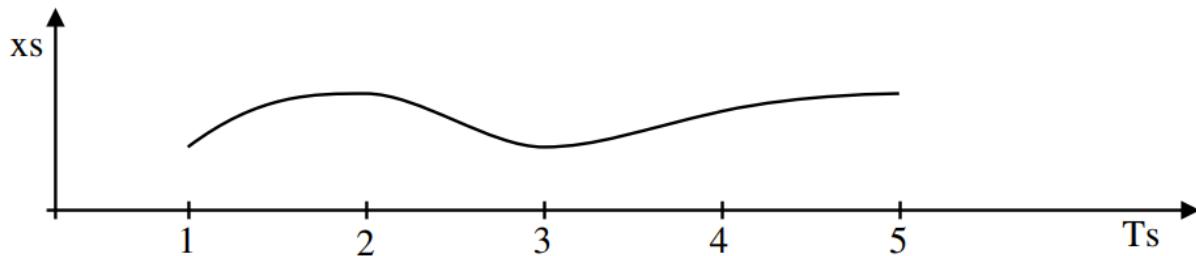
Časová množina

Množina všech časových okamžiků, ve kterých jsou definovány hodnoty vstupních, stavových a výstupních proměnných prvků systému

- **Diskrétní** - {1,2,3,4,5}.



- **Spojitá** - $<1.0, 5.0>$. Tato se na počítači diskretizuje.



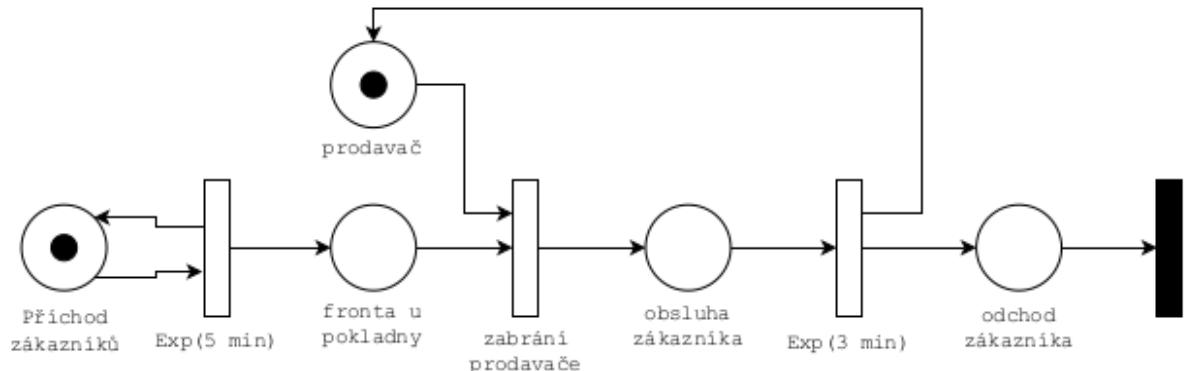
Simulační metody

Petriho sítě

Petriho síť je formálně definována jako pětice $N = (P, T, I, O, M_0)$, kde:

- $P = \{ p_1, p_2, \dots, p_m \}$ je konečná množina míst,
- $T = \{ t_1, t_2, \dots, t_n \}$ je konečná množina přechodů, $P \cup T \neq \emptyset$ a $P \cap T = \emptyset$,
- $I : T \times P \rightarrow \mathbb{N}$ je vstupní funkce, která definuje orientované křivky z míst do přechodů, kde \mathbb{N} je množina nezáporných celých čísel,
- $O : T \times P \rightarrow \mathbb{N}$ je výstupní funkce, která definuje orientované křivky z přechodů
- do míst,
- $M_0 : P \rightarrow \mathbb{N}$ je počáteční značení.

Značení znamená přiřazení žetonů do míst Petriho sítě. Orientovaná křivka směřující z místa p_i do přechodu t_j definuje p_i jako **vstupní místo přechodu t_j** . Orientovaná křivka směřující z **přechodu t_j** do **místa p_i** pak definuje p_i jako **výstupní místo** přechodu t_j . Vstupní místo je místo, ze kterého je žeton při provádění přechodu **odebrán**, a naopak výstupní místo je místo, do kterého je **žeton** po provedení přechodu **přesunut**. Graficky jsou **místa** v Petriho sítích značena elipsami, obvykle je ale znázorňujeme **kružnicemi**. **Přechody** jsou značeny **obdélníky**, orientované **křivky šipkami**. **Žetony** mohou být značeny **tečkami** nebo **číslem**, které vyjadřuje počet teček – žetonů. Značení žetonů se obvykle vpisuje do značky místa. Příklad systém M/M/1 dle Kendalovy klasifikace:



Metoda Monte Carlo

Experimentální numerická (simulační) metoda. Řeší úlohu experimentování se stochastickým modelem. Využívá vzájemného vztahu mezi hledanými veličinami a pravděpodobností, se kterými nastanou jevy. Vyžaduje generování náhodných čísel. Není příliš přesná. Vhodné, když jsou běžné numerické metody nepraktické.

Jednoduchá implementace (existuje více variant).

1. Vytvoříme stochastický model.
2. Provádíme náhodné experimenty.
3. Získanou pravděpodobnost nebo průměr použijeme pro výpočet výsledku.

Použití:

- výpočet obsahu, objemu těles (nemusíme znát obor hodnot funkce)
- výpočet integrálů, zejména vícerozměrných,
- řešení diferenciálních rovnic

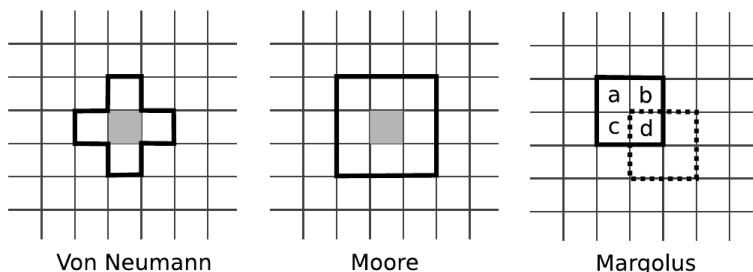
Přesnost (N je počet provedených experimentů):

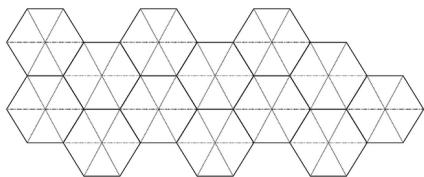
$$err = \frac{1}{\sqrt{N}}$$

Celulární automaty

Celulární automaty jsou diskrétní systémy. Je možné implementovat jako pole, vyhledávací tabulku (nenulové buňky). Existují také reverzibilní automaty, u kterých je možné se vracet nazpět v simulaci. Celulární automaty jsou tvořeny:

- **Buňka (cell)** - základní element, může být v jednom z **konečného počtu stavů**, např. $\{0, 1\}$.
- **Pole buněk (lattice)** - **N-rozměrné** (obvykle 1D nebo 2D). Rovnoměrné rozdělení prostoru, může být **konečné nebo nekonečné**.





- **Okolí (neighbourhood)** - Typy se liší počtem a pozicí **okolních buněk**.
- **Pravidla (Rules)** - Funkce stavu buňky a jejího okolí definující nový stav buňky v čase: $s(t+1) = f(s(t), N_s(t))$.

Lze je rozdělit do 4 tříd:

- **Třída 1** - Po konečném počtu kroků dosáhnou jednoho ustáleného konkrétního stavu.
- **Třída 2** - Dosáhnou periodického opakování nebo zůstanou stabilní.
- **Třída 3** - Chaotické chování (fraktální útvary).
- **Třída 4** - Kombinace běžného a chaotického chování (např. life) - nejsou reverzibilní.

Nejznámější celulární automat je hra **Life**.

Numerické metody

Markovovy modely

Metoda snižování řádu derivace

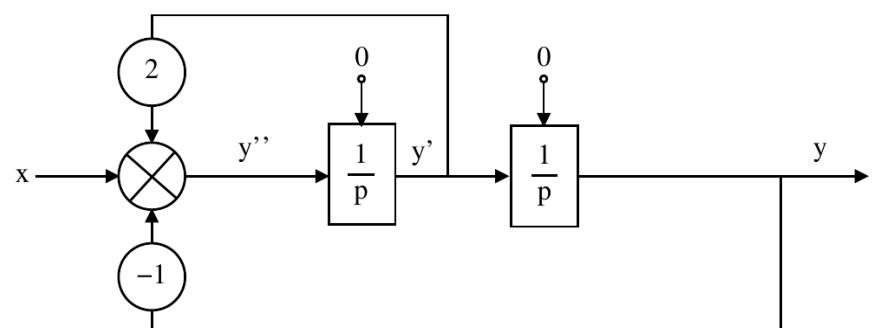
1. Osamostatnit nejvyšší řád derivace.
2. Zapojit všechny integrátory za sebe a na vstupu prvního zapojit (modifikovaný - násobení, přičítání) výsledek
3. Tato metoda funguje, pokud nejsou derivované vstupy derivace vstupů (x' , x'' , ...).

Příklad: rovnice $y'' - 2y' + y = x$

$$y'' = 2y' - y + x$$

$$y' = \int y''$$

$$y = \int y'$$



Metoda postupné integrace

1. **Osamostatnit nejvyšší řád** derivace.
2. Postupná integrace rovnice a zavádění nových stavových podmínek.

3. Výpočet nových počátečních podmínek.

Příklad: rovnice $p^2y + 2py + y = p^2x + 3px + 2x$

$$p^2y = p^2x + p(3x - 2y) + (2x - y)$$

$$py = px + (3x - 2y) + \frac{1}{p}(2x - y), \text{ proměnná } w_1 = \frac{1}{p}(2x - y)$$

$$py = px + (3x - 2y) + w_1$$

$$y = x + \frac{1}{p}(3x - 2y + w_1), \text{ proměnná } w_2 = \frac{1}{p}(3x - 2y + w_1)$$

$$y = x + w_2$$

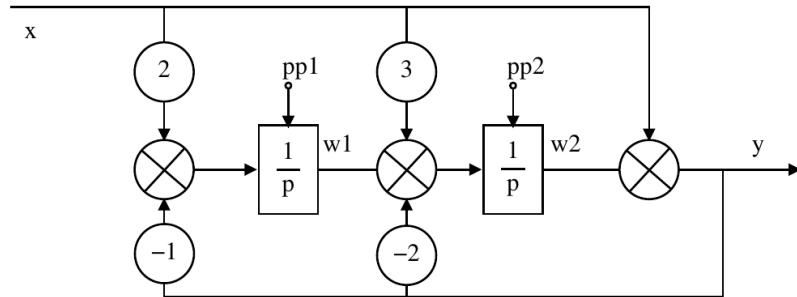
Výsledná soustava rovnic:

$$w_1 = \frac{1}{p}(2x - y), \quad w_1(0) = y'(0) - x'(0) - 3x(0) + 2y(0)$$

$$w_2 = \frac{1}{p}(3x - 2y + w_1), \quad w_2(0) = y(0) - x(0)$$

$$y = x + w_2$$

4.



Algoritmy řízení simulace

Různé algoritmy pro diskrétní, spojité a kombinované simulace.

Next event (řízení diskrétní simulace)

Jedná se algoritmus řízení diskrétní simulace. Vypadá následovně:

1. **Inicializace** času, kalendáře, modelu.
2. Dokud **není kalendář prázdný**, vyjmi první záznam z kalendáře (záznamy jsou seřazeny)
3. Pokud aktivační čas události **přesáhl** čas konce simulace, ukonči simulaci.
4. **Nastav** čas na aktivační čas události.
5. Proved' **chování**, které popisuje událost a **vrat' se** na bod 2.

Kalendář

Uspořádaná **datová struktura** uchovávající aktivační záznamy budoucích událostí.

Každá naplánovaná událost má v kalendáři záznam, který je tvořen minimálně:

- **aktivační čas**,

- priorita,
- vykonávanou událost.

Kalendář umožňuje **výběr prvního záznamu** s nejmenším aktivačním časem a **vkládání/rušení záznamu**.

Řízení spojité simulace

Spojitá simulace je složena ze tří částí:

- metoda/funkce **Dynamic**, ve které dochází k **aktualizaci vstupů integrátorů**,
- **numerická metoda** (Euler, RK) ve které se **pro každý integrátor počítají nové stavy**,
- **hlavní cyklus**, který řídí simulaci (volá předchozí metody), **inkrementuje čas dle kroku** a sleduje, jestli nebylo dosaženo konce simulace a případně provádí **dokročení**.

Dokročení může být řešeno dvěma způsoby:

- pokud do konce simulace zbývá **menší časový okamžik**, než je nastavený krok, lze poslední **krok snížit** tak, aby čas po jeho provedení přesně odpovídal konci simulace. (příliš malý krok může způsobit nepřesnost)
- pokud do konce simulace **zbývá krok a kousek**, můžeme poslední **krok prodloužit** tak, aby čas po jeho provedení přesně odpovídal konci simulace.

Příklad řízení spojité simulace bez dokročení:

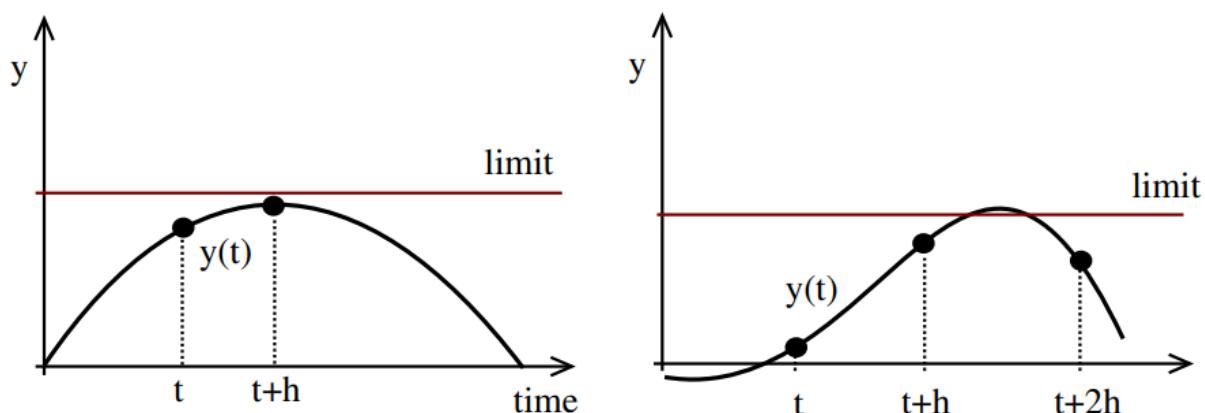
```

double yin[2], y[2] = { 0.0, 1.0 }, time = 0, h = 0.001;
void Dynamic() { // f(t,y): výpočet vstupů integrátorů
    yin[0] = y[1]; // y'
    yin[1] = -y[0]; // y'',
}
void Euler_step() { // výpočet jednoho kroku integrace
    Dynamic(); // vyhodnocení vstupů integrátorů
    for (int i = 0; i < 2; i++) // pro každý integrátor
        y[i] += h * yin[i]; // vypočteme nový stav
    time += h; // posun modelového času
}
int main() { // Experiment: kruhový test, čas 0..20
    while (time < 20) {
        printf("%10f %10f\n", time, y[0]);
        Euler_step();
    }
}

```

Řízení kombinované simulace

U kombinované simulace je nutné řešit kombinaci **stavových událostí** a **numerické integrace**. Je nutné **detektovat změnu** stavových podmínek a **přesně dokročit** na čas, kdy ke změně dochází (problém příliš malého kroku). Ke **stavovým událostem** dochází při **změnách stavových podmínek**, nelze je naplánovat. Změnu stavové podmínky může být **obtížné detektovat** z důvodu **nepřesnosti numerického výpočtu** nebo **příliš dlouhého kroku**.



Algoritmus řízení kombinované simulace pracuje následovně:

1. **Inicializace** stavu, času, modelu atd.
2. **Kontrola**, že není dosažen čas **konce simulace** a její případné ukončení.
3. **Uložení** aktuálního stavu a času.
4. Provedení jednoho **kroku numerické integrace**.
5. Kontrola, jestli nedošlo ke změně stavových podmínek. Pokud ne, pokračuje

se bodem 2, jinak bod 6.

6. Hledání **okamžiku** změny **stavové podmínky** (využití uložených hodnot v **kroce 3**) např. metodou půlení intervalů. Okamžik se hledá **s přesností minimálního kroku** (menší krok by vedl na nepřesnost, stejně jako na nepřesnost vede nepřesné určení času změny stavové podmínky)

Pseudokód algoritmu:

```
Inicializace stavu a podmínek
while ( čas < koncový_čas) {
    Uložení stavu a času ***

    Krok numerické integrace a posun času
    Vyhodnocení podmínek
    if ( podmínka změněna )
        if ( krok <= minimální_krok)
            Potvrzení změn podmínek
            Stavová událost ===
            krok = běžná_velikost_kroku
        else
            Obnova stavu a času ***
            krok = krok/2
            if (krok < minimální_krok)
                krok = minimální_krok
}
```

Řízení simulace číslicových obvodů

Tato simulace je řízená událostmi a ukládání **velkého množství** událostí do kalendáře je **nepraktické/problematické**. Používá se proto princip **selektivního sledování**, kdy dochází k vyhodnocování pouze těch prvků, na které má vliv změna na vstupu. Používá se např. **pevný krok** pro změnu času) Problematické mohou být zpětné vazby v obvodech a nastavení počátečních hodnot signálů.

Princip algoritmu:

1. **Inicializace modelu, plánování, ...**
2. Dokud je naplánovaná událost, tak pokračuj s dalším bodem, jinak konec simulace.
3. Nastav hodnotu modelového času na **T** (pevným krokem, na základě právní naplánované události, ...)
4. Pro všechny události, které jsou **naplánované** na tento čas **T** proved:
 - a. **odeber** událost z plánovaných událostí (kalendáře),
 - b. **aktualizuj** hodnoty signálů,
 - c. všechny připojené prvky na tyto signály **zařaď do množiny M**.

5. Projdi všechny prvky v množině **M** a jestli změna na jeho vstupu způsobí **změnu jeho výstupu, naplánuj** jeho obsluhu jako novou událost.
6. Pokračuj bodem 2.

Pseudokód:

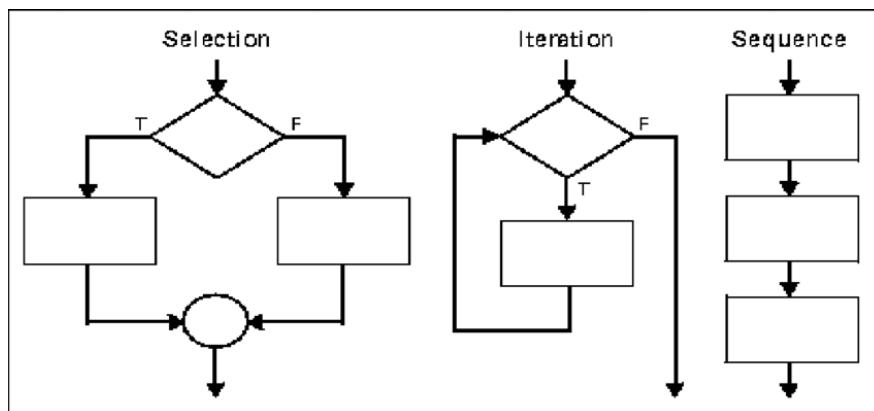
```
 inicializace, plánování události pro nový vstup
 while (je plánována událost) {
     nastavit hodnotu modelového času na T
     for (u in všechny plánované události na čas T) {
         výběr záznamu události u z kalendáře
         aktualizace hodnoty signálu
         přidat všechny připojené prvky do množiny M
     }
     for (p in množina prvků M) {
         vyhodnocení prvku p
         if (změna jeho výstupu)
             plánování nové události
     }
 }
```

29. Datové a řídicí struktury imperativních programovacích jazyků.

Řídící struktury

- **Sekvence** - umožňuje provádět příkazy (**přiřazení, volání funkce, aritmetické operace, ...**) jeden po druhém. Obecněji umožňuje provádění podprogramů jeden po druhý.
- **Selekce** - Umožňuje volbu mezi bloky kódu, které budou provedeny na základě hodnoty booleovského výrazu (**if - else if - else, switch, ...**). Obecněji umožňuje vybrat jeden ze dvou podprogramů, který bude prováděn, na základě hodnoty booleovského výrazu.
- **Iterace** - Opakování bloku kódu na základě hodnoty booleovského výrazu (**for, while, repeat, do while, goto, ...**). Obecněji umožňuje opakování vykonávání podprogramu.

Všechny programovací jazyky, které podporují tyto řídící struktury (a jsou vybaveny **nekonečným** sekundárním úložištěm - tento fakt je většinou ignorován) jsou **Turing Complete**. Existují i jazyky, které mají pouze jedinou instrukci a jsou Turing Complete (pomocí této instrukce je však simulována manipulace s pamětí, větvení a iterace). Tato trojice je také základem strukturovaného programování.



Výraz/příkaz

Například **aritmetický výraz**, prostý **výskyt konstanty** či **proměnné, funkční volání, přiřazení, ...**. Výrazový příkaz se získá **ukončením výrazu patřičným symbolem** (v C, C++, C# aj. je to středník, v Pythonu je to konec řádku). Zvláštním případem je prázdný příkaz (samostatný středník).

Podmíněný příkaz if-else

Umožňuje rozhodnutí na základě hodnoty proměnné (**true/false**), jestli **bude** kód

vykonán (true), nebo **nebude** (false). Respektive v případě nepravdy (false) se vykoná **else** větev, pokud existuje.

Podmíněný příkaz switch

Slouží k **větvení** podle hodnoty **celočíselného výrazu**, **řetězce**, **hodnoty výčtu** atd. Řízení programu přechází do úseku kódu jehož konstantní hodnota (která jej uvádí - hodnota **case**) odpovídá hodnotě proměnné, podle které se přepíná ("switchuje" v příkazu **switch**). Switch také umožňuje provést výchozí chování (větev **default**), pokud žádný z **case** nevyhovuje.

Cykly

Umožňují několikanásobné (**for** cyklus) opakování kódu. Nekonečné cykly (**while(true)**) je možné obvykle ukončit příkazem **break**. Přeskočení zbytku nynější iterace je možné obvykle příkazem **continue**.

- **while** - Vhodné například pro předem neznámý počet provádění. Vyhodnocuje se poprvé podmínka, tedy **tělo nemusí být vykonáno**.
- **do while (repeat until)** - Tělo je provedeno alespoň jednou.
- **for** - Tělo se provede **přesně stanovený počet krát**, pokud neuvažujeme modifikaci ve vyšších programovacích jazycích na podmínku.
- **goto** - v případě skoku na **návěští**, které **goto** **předchází**. Skákat lze pouze lokálně (tedy nelze skočit pryč z funkce - na to je např. longjump v C - to je nebezpečné narozdíl od goto).

Složený příkaz (blok)

Posloupnost libovolných příkazů (**včetně dalších bloků**). Často se **syntakticky odlišuje** (**{}** v C, **begin end** v Pascalu, **odsazení** v Python). **Proměnné** zde definované mají často **platnost pouze v tomto bloku - lokální proměnné**.

Funkce

Umožňují rozložení složitého problému na jednodušší **podprogramy**, které mají jedno vstupní místo (obvykle standardizovaným způsobem). **Deklarace** funkce se skládá z **návratového typu**, **názvu** funkce a **seznamu parametrů**. Deklaraci je často možné vynechat a funkci přímo definovat. Funkce jsou jedním z hlavních předpokladů pro **strukturované jazyky**.

Rekurze

Metoda definování určitého objektu pomocí sebe sama.

- Umožňuje definovat **nekonečnou množinu** objektů **konečným popisem - rekurentní definice**.
- Rekurzivní struktura dat (lineární seznam)
- Rekurzivní struktura algoritmu (DFS, InOrderTraversal)
- Rekurzivní volání funkce – **funkce je volána v těle sebe samé** (obecně se

tomuto případu snažíme vyhnout, **jednoduché** na implementaci, ale **nebezpečné**).

Datové struktury

Datová struktura určuje konkrétní způsob **organizace dat v paměti počítače**.

Datový typ

Určuje **množinu hodnot** a **množinu operací nad těmito hodnotami**. Typ proměnné se zavádí v **její deklaraci**. Je určen:

- Názvem
- Množinou hodnot, kterých může nabývat
- Množinou operací nad hodnotami

Dělení datových typů (dle několika způsobů):

- **standardní** (int, double, char, string, boolean)
- **definované uživatelem** (enum, pole, záznam, množina, soubor, ukazatel)
- **ordinální** - prvky daného typu mají jasně stanovené pořadí. Každý prvek má předchůdce a následovníka (integer, char, ...)
- **neordinální** (string, real, pointer, ...)
- **jednoduché/základní** - nemají vnitřní strukturu, s proměnnou lze pracovat jako s celkem (integer, char, real, ...)
- **strukturované** - hodnoty mají vnitřní strukturu (členění na komponenty), s proměnnou strukt. typu lze pracovat buď jako s celkem nebo s jeho jednotl. komponenty (array, struct, record, ...)

Strukturovaný datový typ

Sestává z komponent jiného (dřív definovaného) typu = kompoziční typ. Má strukturovanou hodnotu a musí mít definované hodnoty všech komponent.

- **Homogenní** - Všechny komponenty (položky) jsou **stejného typu - pole**.
- **Heterogenní** - Komponenty (složky) jsou **rozdílného typu - struktura**.
- **Statický** - Nemůže měnit v průběhu výpočtu počet komponent ani způsob uspořádání.
- **Dynamický** - Může měnit v průběhu počet i uspořádání.

Pole

Homogenní ortogonální (pravoúhlý) datový typ. Typ je **specifikovaný velikostí svých dimenzí a komponentním typem**. K prvkům se přistupuje pomocí **identifikátoru pole a indexu prvku**.

- **Vektor** - jednorozměrné pole.
- **Matice** - dvourozměrné pole.

Řetězec

Strukturovaný homogenní datový typ. Položky mají **typ znak** (char). Má vlastnosti (a

často je tak i implementován) jako **jednorozměrné pole znaků**, ale ve většině vyšších programovacích jazyků umožňuje více operací (např. porovnání, konkatenace, ...).

Záznam (record/struct)

Strukturovaný statický heterogenní datový typ. Komponenty mohou být libovolného datového typu. Jména a počet komponent je dán při definici typu a nemohou se měnit za běhu programu.

Abstraktní datové typy

Při tvorbě reálných aplikací lze využít **obecného modelu datové struktury** vyjádřeného pomocí abstraktního datového typu:

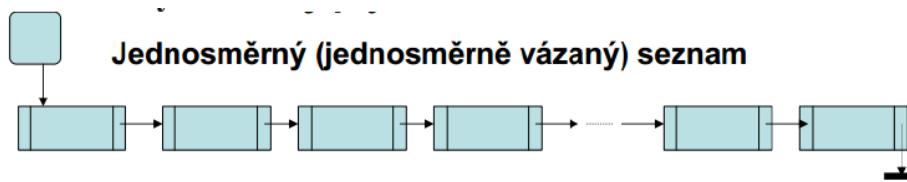
- určíme použité **datové komponenty**,
- určíme **operace** a jejich **vlastnosti**,
- **abstrahujeme** od způsobu implementace.

ADT zdůrazňuje **co** dělá, ale **potlačuje jak** to dělá. Smyslem je **zvýšit** datovou **abstrakci** a **snížit** algoritmickou **složitost** programu. ADT je definován **množinou hodnot**, kterých smí nabýt každý prvek tohoto typu, a **množinou operací** nad tímto typem

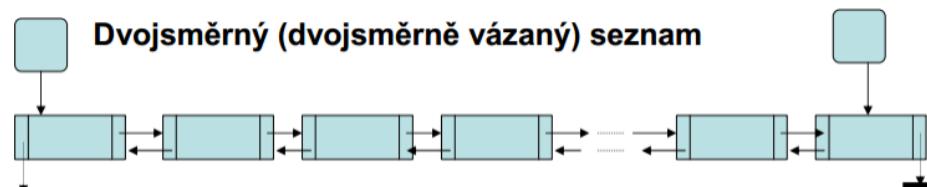
Spojový seznam (linked list)

Homogenní, lineární, dynamická struktura. Prvkem seznamu může být **jakýkoliv typ** (také strukturovaný). Seznam může být prázdný.

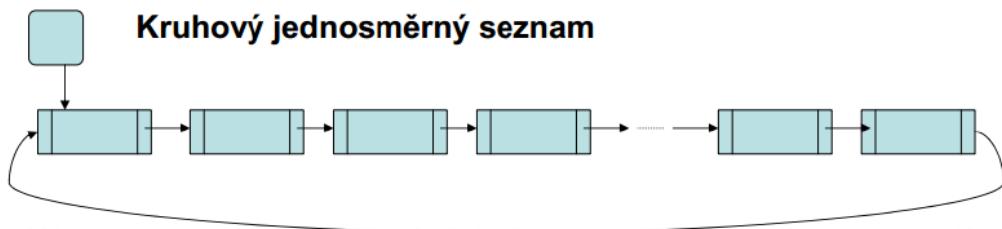
- **Ekvivalence** - Když jsou oba seznamy prázdné a nebo když se rovnají jejich první prvky a také jejich zbytky (**rekurentní definice pomocí rekurze**).
- **Jednosměrný** - Prvek ukazuje pouze na svého **následovníka**.



- **Dvousměrný** - Prvek ukazuje na svého **následovníka i předchůdce**.



- **Kruhový jednosměrný** - Jednosměrný, kde **poslední** prvek má za **následovníka** první prvek.



- **Kruhový dvousměrný** - Dvousměrný, kde **poslední prvek** má za **následovníka** **první prvek** a **první prvek** má za **předchůdce** **poslední prvek**.

Operace nad seznamem:

- **inicializace**,
- **vložení** prvku na **začátek seznamu** (lze vložit do prázdného seznamu),
- **získání** hodnoty **prvního** prvku seznamu,
- **odstranění** prvního prvku seznamu,
- **aktivace prvního** prvku,
- **aktivace následujícího** nebo předcházejícího (u dvousměrného) prvku,
- **získání** hodnoty **aktivního** prvku,
- **změna** hodnoty **aktivního** prvku,
- **vložení** prvku **za/před aktivní** (před pouze u dvousměrného) prvek
- **test na prázdnost**,
- **test na aktivní prvek**.

Zásobník

Homogenní, lineární, dynamická struktura typu **LIFO (Last in - First out)**. Využívá se pro reverzaci pořadí, konstrukci rekurzivních programů bez rekurze - **DFS**.

Implicitně používá zásobník operačního systému každý program při volání funkcí.

Lze implementovat pomocí **pole** nebo **spojového seznamu**.

Použití:

- Vyčíslování aritmetických výrazů v postfixové notaci
- Převod z infixové do postfixové notace

Operace se zásobníkem:

- **Init** - Vytvoří prázdný zásobník.
- **Push** - Vloží prvek na vrchol zásobníku.
- **Pop** - Výjme prvek z vrcholu zásobníku.
- **Top** - Přečte hodnotu z vrcholu zásobníku.
- **Empty** - True pokud je prázdný.

Fronta

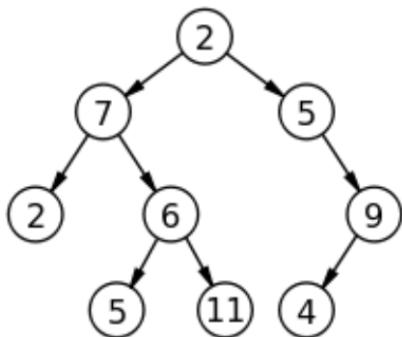
Dynamická, homogenní a lineární struktura typu **FIFO (First in - First out)**. Na jedné straně se přidává (konec fronty) a na druhé se čte a odebírá – obsluhuje (začátek fronty). Využívá se např. v **systémech hromadné obsluhy**, implementaci **BFS**, návrhový vzor **producent konzument**. Lze implementovat pomocí **pole** nebo **spojového seznamu**. Existují prioritní fronty, kde mohou prvky s vyšší prioritou předbíhat.

- **Init** - Vytvoří prázdnou frontu.
- **Enqueue** - Vloží prvek na konec fronty.
- **Dequeue** - Výjme prvek ze začátku.
- **Front** - Přečte hodnotu ze začátku fronty.
- **Empty** - True pokud je prázdná.

Binární strom

Homogenní dynamická struktura. Rekurentní definice:

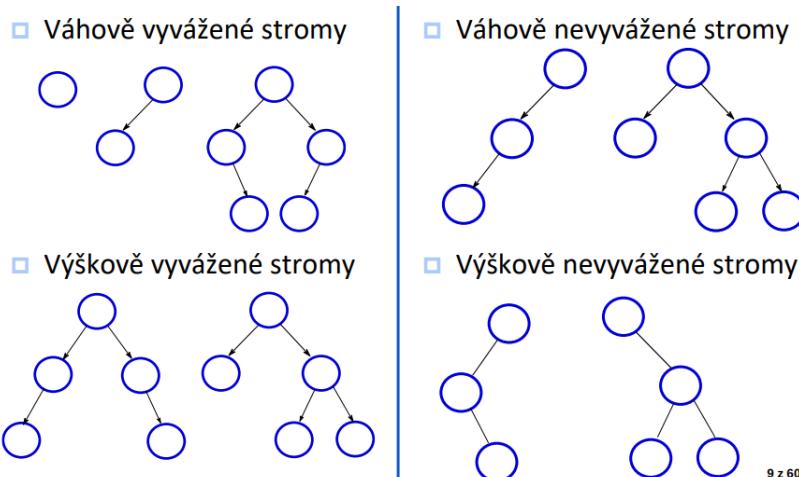
Binární strom je **buď prázdný** nebo se sestává z **jednoho uzlu** zvaného **kořen** a **dvojí podstromů** (levého a pravého) a oba podstromy mají vlastnost stromu.



- **Váhově vyvážený** - Pokud pro **všechny jeho uzly** platí, že počty uzelů jejich levého podstromu a pravého podstromu **se rovnají nebo se liší právě o 1**.
- **Výškově vyvážený** - Když pro **všechny jeho uzly** platí, že **výška levého podstromu se rovná výšce pravého podstromu nebo se liší právě o 1**.

Při zajištění vyváženosti **nemůže dojít k degradaci** stromu na seznam.

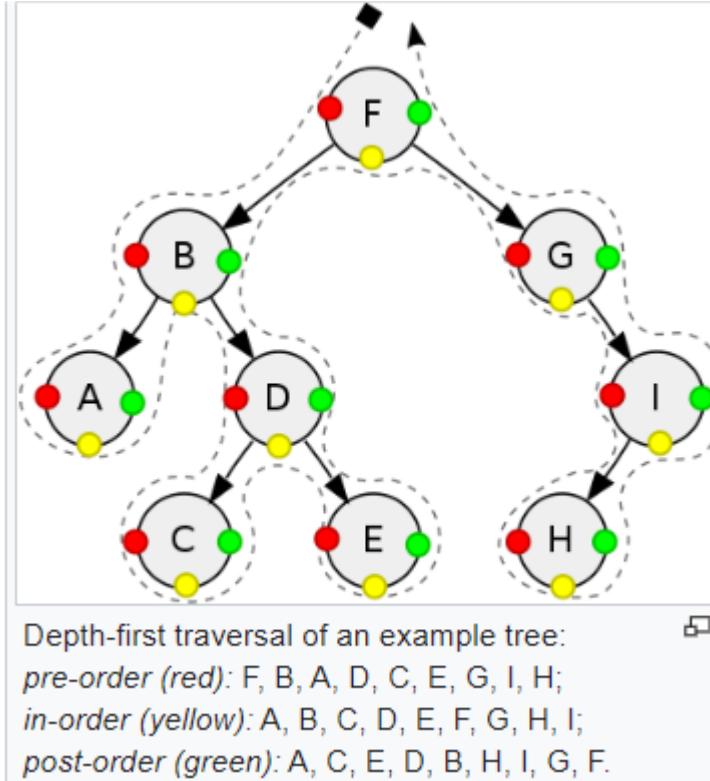
Samovyužívající se stromy: **red-black tree** (červený uzel nemá červeného následníka a na každé cestě z libovolného uzel k listu je stejný počet černých uzelů), **AVL tree** (uzly mají váhu - **0: zcela vyvážený** uzel, **-1: výška levého** podstromu je o jedna **větší**, **1: výška pravého** podstromu je o jedna **větší**, pokud dojde ke změně váhy na **-2/2**, je nutné situaci napravit - **rotace**).



9 z 60

- **Preorder** - aktuální uzel, levý podstrom, pravý podstrom.

- **Inorder** - levý podstrom, aktuální uzel, pravý podstrom.
- **Postorder** - levý podstrom, pravý podstrom, aktuální uzel.
- **InvPreOrder** - aktuální uzel, pravý podstrom, levý podstrom.
- **InvInOrder** - pravý podstrom, aktuální uzel, levý podstrom.
- **InvPostOrder** - pravý podstrom, levý podstrom, aktuální uzel.



Vyhledávací tabulka (Look-up table, hash table)

Homogenní, obecně dynamická struktura, ve které má každá položka zvláštní složku - **klíč**. Klíč by měl být v tabulce unikátní (s jedinečnou hodnotou), aby bylo podle něj možné provádět (ostré) vyhledávání. V ideálním případě je vyhledávání s konstantní časovou složitostí.

Tabulka s přímým přístupem

Implementuje se pomocí pole, ve kterém jsou klíče mapovány na indexy pole. To vyžaduje vzájemně jednoznačné zobrazení (**bijekce**) mapující každý prvek množiny klíčů **K** do množiny indexů pole **H**. Jedná se o ideální strukturu pro vyhledávání, která je ale v praxi prakticky nepoužitelná (stejně velké pole jako počet klíčů je nereálné). Proto se používá mapovací (hashovací) funkce, která obvykle mapuje větší počet klíčů na menší počet hodnot. Vznikají tak **kolize** - dva různé klíče jsou namapovány do stejného místa (na stejný index). **Synonyma** jsou poté – dva nebo více klíčů, které jsou namapovány do téhož místa. Tabulky je vhodné navrhovat tak, aby bylo jejich očekávané naplnění **70%-75% (load factor)**. Problém kolizí a synonym lze řešit:

- **Implicitní zřetězení**: adresa následníka se získá pomocí funkce z adresy předchůdce (otevřená adresace). V praxi se **umístí prvek s klíčem**

mapovaným na již obsazenou pozici na **první volnou pozici**, která následuje za získaným indexem. Vyhledávání se poté provádí **sekvenčně** (případně s nějakým **krokem** - jednotkový má tendenci tvořit shluky, ten se může zvětšovat, velikost tabulky by poté měla odpovídat **prvočíslu**) od této pozice, dokud se nenarazí na požadovaný prvek nebo na prázdné místo (musí se uvažovat přechod z konce na začátek, realizováno pomocí modulo).

- **Explicitní zřetězení:** adresa následníka je obsažena v jeho předchůdci (zřetězení záznamů). Lze řešit **spojovým seznamem** na každém indexu pole, položky s klíči, které se mapují na stejný index jsou poté ukládány do těchto seznamů. Při použití nekvalitní mapovací funkce může degradovat na seznam s lineární složitostí. To lze řešit použitím **stromu místo seznamu** na každém indexu. Pak bude v nejhorším případě složitost vyhledávání logaritmická (při vyváženém stromu).

V obou případech lze tabulku v případě nutnosti zvětšit a záznamy přemapovat. Jedná se ale o složitou operaci.

Mapovací (hashovací) funkce

Kvalitní mapovací funkce by měla splňovat tyto požadavky:

- Determinismus: Pro daný klíč vrátí vždy stejnou hodnotu.
- Rovnoměrné (uniformní) rozložení: Na každé místo se mapuje přibližně stejně velké množství klíčů.
- Využití celých vstupních dat: Zohlednění každého bitu, viz následující bod.
- Vyhnutí se kolizím podobných klíčů: V praxi bývá řada klíčů velice podobných. Rychlý výpočet.

Výsledek mapovací funkce musí být v rozsahu pole, do kterého jsou prvky ukládány (lze zajistit operací modulo délky pole)

Odkazy:

- <http://dudka.cz/studyIAL>

30. Vyhledávání a řazení

Řazení

- **Třídění (sorting)** položek neuspořádané množiny je **uspořádání do tříd podle** hodnoty daného **atributu** – klíče položky. **Mezi třídami nemusí být definovaná relace uspořádání** (jablka, hrušky, švestky lze třídit). Často se používá **řazení (sorting) pro třídění**.
- **Řazení (ordering, sequencing)** je **uspořádání** položek podle **relace lineárního uspořádání** nad klíči. Termín se nepoužívá často.
- **Setřídění (merging)** je vytváření souboru seřazených položek **sjednocením** několika souborů položek téhož typu, které **jsou již seřazeny**. Příkladem je algoritmus Merge sort.

Vlastnosti řadících algoritmů

Slouží pro výběr vhodného algoritmu pro kníkrétní implementaci.

Přirozenost

Algoritmus se chová přirozeně pokud:

- doba potřebná k seřazení **náhodně uspořádaného pole je větší, než k seřazení již uspořádaného pole**,
- doba potřebná k seřazení **opačně seřazeného pole je větší, než doba k seřazení náhodně uspořádaného pole**.

Jinak říkáme, že se algoritmus nechová přirozeně.

Stabilita

Stabilita vyjadřuje, zda mechanismus algoritmu zachovává **relativní pořadí klíčů se stejnou hodnotou**. Sekvence 7,5',3,1,5'',9,2,5''',8,4,6 pro následující příklad.

- **stabilní** algoritmus: 1,2,3,4,5',5'',5''',6,7,8,9.
- **nestabilní** algoritmus: 1,2,3,4,5'',5',5''',6,7,8,9 (nebo jinak).

Algoritmy řazení

Podle principu řazení je dělíme na:

- Princip **výběru (selection)** – **přesouvají maximum/minimum** do výstupní posloupnosti. Např.: **Selection sort**, **Bubble sort** a jeho modifikace (**Ripple sort**, **Shaker sort**, **Shuttle sort**)
- Princip **vkládání (insertion)** – **vkládají** postupně prvky **do seřazené** výstupní posloupnosti. Např.: **Insertion sort**, ...
- Princip **rozdělování (partition)** – **rozdělují** postupně množinu prvků na **dvě podmnožiny** tak, že prvky jedné jsou **menší než prvky druhé**. Např.: **Quick**

sort, Shell sort, ...

- Princip **slučování (merging)** – setříděují se postupně dvě seřazené posloupnosti do jedné. Např.: **Merge sort, ...**
- **Jiné principy.** Např.: **řazení tříděním, ...**

Podle typu procesoru:

- **sériové** (jeden procesor) – jedna operace v daném okamžiku,
- **paralelní** (více procesorů) – více souběžných operací

Podle přístupu k paměti:

- **přímý (náhodný)** přístup – metody vnitřního řazení (řazení polí)
- **sekvenční** přístup – metody vnějšího řazení (řazení souborů a seznamů)

Řazení tříděním - Radix sort

Řazení tříděním musí probíhat od **nejnižší po nejvyšší** prioritu (**od LSB po MSB**, od jednotek přes desítky, stovky, tisíce, ... tj. podle základu). Jedná se o **třídění pomocí přihrádek** (u desítkového základu jich je 10 a třídění provádíme pro každou číslici klíče). Využití pro řazení děrných štítků.

43 27 31 15 37 80 03



80 31 43 03 15 27 37



03 15 27 31 37 43 80

Příklad **správného** (vlevo) a **špatného** (vpravo) postupu řazení tříděním na množině {342, 835, 942, 178, 256, 493, 884, 635, 728}:

342	728	178
942	835	256
493	635	342
884	342	493
835	942	635
635	256	728
256	178	835
178	884	884
728	493	942



178	728	342
256	635	942
342	835	493
493	342	884
635	942	635
728	256	835
835	178	256
884	884	728
942	493	178



Vlastnosti:

- **stabilní**,
- **nechová se přirozeně**,
- **nepracuje in situ**,
- **linearitmická $O(n \cdot \log n)$** časová složitost.

Selection sort

Princip metody je postaven na **nalezení extrémního prvku** v zadaném segmentu pole a jeho **výměna na konec (začátek) seřazené části pole**. Příklad řazení od největšího k nejmenšímu, viz <https://www.algoritmy.net/article/4/Selection-sort>:

1. (3 2 8 7 6) - zadané pole,
2. (3 2 8 7 6) - nejvyšší číslo je 8, prohoďme ho tedy s číslem 3 na indexu **0**,
3. (8 2 3 7 6) - nejvyšší číslo je 7, prohoďme ho tedy s číslem 2 na indexu **1**,
4. (8 7 3 2 6) - nejvyšší číslo je 6, prohoďme ho tedy s číslem 3 na indexu **2**,
5. (8 7 6 2 3) - nejvyšší číslo je 3, prohoďme ho tedy s číslem 2 na indexu **3**,
6. (8 7 6 3 2) - seřazeno

Vlastnosti:

- **nestabilní** - z důvodu výměny prvků,
- měla by být **přirozená**,
- složitost: **kvadratická $O(n^2)$** ,
- pracuje **in situ**.

Bubble sort

Stejně jako selection sort pracuje metoda na principu **nalezení extrémního prvku** a jeho **umístění na konec (začátek) již seřazené části**. Liší se ale v principu **nalezení extrému výměny prvků**, viz <https://www.algoritmy.net/article/3/Bubble-sort>.

Postup:

- Porovnává se každá dvojice a v případě **obráceného uspořádání** (záleží, jestli řadíme od největšího nebo od nejmenšího) **se přehodí**.
- Při pohybu **zleva doprava a řazení od nejmenšího k největšímu** se tak **maximum dostane na poslední pozici**. **Minimum se posune o jedno místo směrem ke své konečné pozici**.

Vlastnosti:

- **stabilní** - na rozdíl od selection sort,
- **přirozená** - nejrychlejší metoda pro již seřazené pole,
- složitost: **kvadratická $O(n^2)$** ,
- pracuje **in situ**.

Další **odvozené metody** od Bubble sort (Ripple sort, Shaker sort, Shuttle sort) jsou obecně rychlejší ale né z pohledu **časové složitosti**, ta zůstává **$O(n^2)$** .

Heap sort

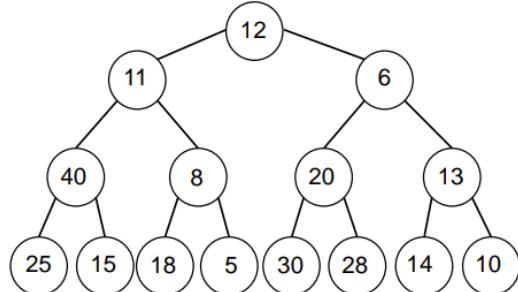
Princip algoritmu je založený na **hromadě**. Hromada je struktura **stromového typu**, pro niž platí, že mezi **otcovským uzlem a všemi jeho synovskými** (to platí i v

libovolných podstromech) uzly platí **stejná relace uspořádání** (buď je menší, nebo větší). Nejčastěji se používá **binární hromada**, která je založena na **binárním stromu**, který je konstruován tak, aby:

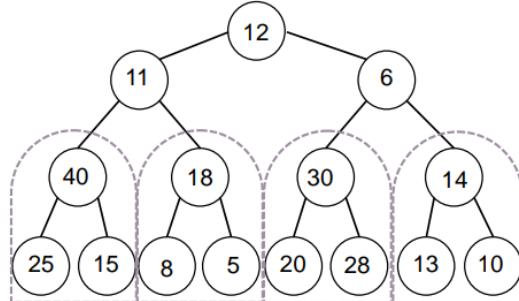
- všechny hladiny kromě poslední **byly plně obsazene**,
- **poslední hladina je zaplněna zleva**.

Binární hromadu lze také použít např. k implementaci prioritní fronty (pokud neexistují prvky se stejnou prioritou a stačí nám přístup k nejprioritnějšímu prvku).

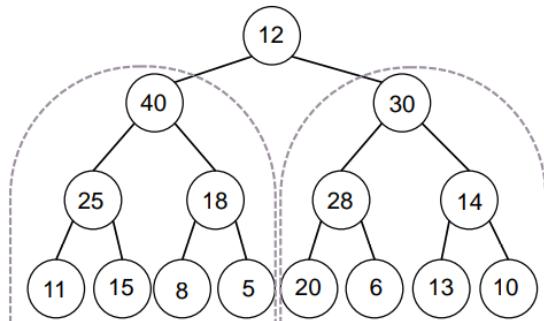
Vytvoření hromady



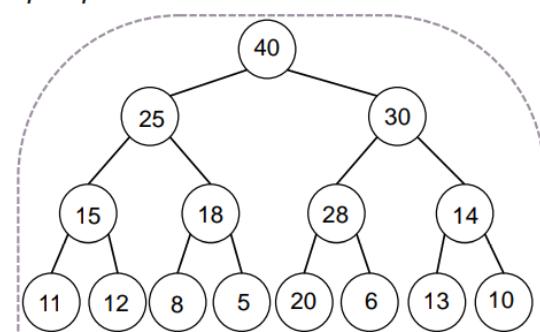
Neuspořádané pole



Vytvoření 4 hromad (zprava doleva) na předposlední úrovni



Vytvoření 2 hromad (zprava doleva) na 2. úrovni



V posledním kroku lze ze 2 hromad 46 z vytvořit jedinou hromadu

Rekonstrukce hromady

Rekonstrukce **hromady** se realizuje pomocí **prosetí/zatřesení** (sift), která **znovuustanoví** hromadu **porušenou pouze v kořeni** [Heap sort in 4 minutes](#):

- na místo kořenového uzlu přesuneme **nejnižší a nejpravější uzel**,
- zatřeseme s hromadou, což způsobí **propadnutí se prvku v kořeni** na místo, kam patří a **vypropagování největšího prvku** do kořene. Děje se tak **porovnáním s levým a pravým synem** a **výměnou** za toho **většího/menšího** (podle toho, jak řadíme).

Princip algoritmu

1. **Vytvoření hromady** (lze uchovávat v poli),
2. **Odebrání kořene** a umístění jej na konec pole (a zmenšení hromady, uspořádanou část pole již ignorujeme).
3. Pokud není hromada prázdná, **prosetí/zatřesení** s hromadou a pokračování s

bodem 2, jinak máme seřazeno.

Vlastnosti:

- **nestabilní,**
- **nechová se přirozeně,**
- **in situ,**
- **linearitmická složitost $O(n * \log n)$.**

Insertion sort

Pracuje na **principu řazení karet** v ruce, vyberu neseřazenou a vložím ji na místo, kam patří. Princip algoritmu:

1. pole dělíme na **seřazenou část** (levou) a **neseřazenou část** (pravou), na začátku tvoří seřazenou část jeden prvek.
2. Z neseřazené části **vyber první prvek**.
3. V seřazené části **najdi jeho umístění** (místo, kde jeho pravý prvek je větší/menší a jeho levý prvek je menší/větší, případně jsou stejné)
4. **posuň prvky** na pravo od nalezené pozice k pozici umíštovaného prvku **o 1**.
5. Umísti tento prvek.
6. Pokračuj, dokud není neseřazená posloupnost prázdná.

Vlastnosti:

- **stabilní,**
- **přirozený,**
- **in situ,**
- **kvadratická složitost $O(n^2)$.**

Bubble-insert sort

Modifikace, u které je v průběhu porovnávání prováděn přesun položek doprava výměnou za porovnávaný prvek.

Binary-insert sort

Modifikace, u které je vyhledávání prováděno binárním způsobem (půlení intervalů). Pro zajištění stability metody musí binární vyhledávání při více stejných hodnotách vybrat místo za nejpravějším výskytem (Dijkstrova metoda).

<https://www.algoritmy.net/article/8/Insertion-sort>

Quick sort

Algoritmus fungující na principu **rozděl a panuj**. Jedná se o jeden z **nejrychlejších** algoritmů pro řazení (jeho rychlosť je ale nedeterministická). Princip algoritmu:

1. rozděl množinu prvků na dvě podmnožiny tak, aby:
 - a. první obsahovala prvky **menší/větší** nebo rovny **mediánu**,
 - b. druhá obsahovala prvky **větší/menší** nebo rovny **mediánu**.
2. Takto získané podmnožiny opět rozděl na další podmnožiny a obdobně uspořádek prvky.
3. Skonči, pokud množina obsahuje pouze 1 prvek.

Přesun prvků podle mediánu je realizován následovně:

- Procházíme pole současně zleva a zprava.
- Zleva hledáme prvek větší nebo roven mediánu, zprava prvek menší nebo roven mediánu.
- Nalezené prvky vyměníme a hledáme další prvky pro výměnu.
- Proces ukončíme až se dvojice indexů překříží

Medián je prvek, který dělí množinu na dvě části tak, že platí:

- nejméně 50 % hodnot je menších nebo rovných mediánu,
- nejméně 50 % hodnot je větších nebo rovných mediánu.

Samotné vyhledání mediánu je časově náročné, takže se používá **pseudomedián**, což může být **náhodná hodnota** z množiny nebo hodnota, která je ve **středu** množiny na **indexu ($\text{left}+\text{right})/2$** , případně lze vybrat medián ze tří prvků (na začátku, konci a uprostřed, či jinak).

Vlastnosti:

- **nestabilní**,
- **nepracuje přirozeně**,
- **linearitická** časová složitost - v **nejhorším** případě je složitost **kvadratická**,
- pracuje **in situ**, ALE **potřebuje zásobník** pro ukládání hranice ještě nezpracovaných částí. Důležitý trik pro zmenšení zásobníku je **na zásobník uložit (pouze) tu větší část** a dále, tedy dřív, **zpracovávat vždy menší** z rozdelených částí. To zaručí, že stačí zásobník velikosti **$\log_2(n)$** , tj. například pro miliardu prvků stačí hloubka 30 položek. Bez této optimalizace by byla paměť potřebná pro zásobník **až lineární**, takhle je logaritmická **$O(\log(n))$** .

<https://www.algoritmy.net/article/10/Quicksort>

Shell sort

Algoritmus fungující na principu **rozděl a panuj**. Vytváří sekvence prvků, které seřazuje bublinovým průchodem. Princip:

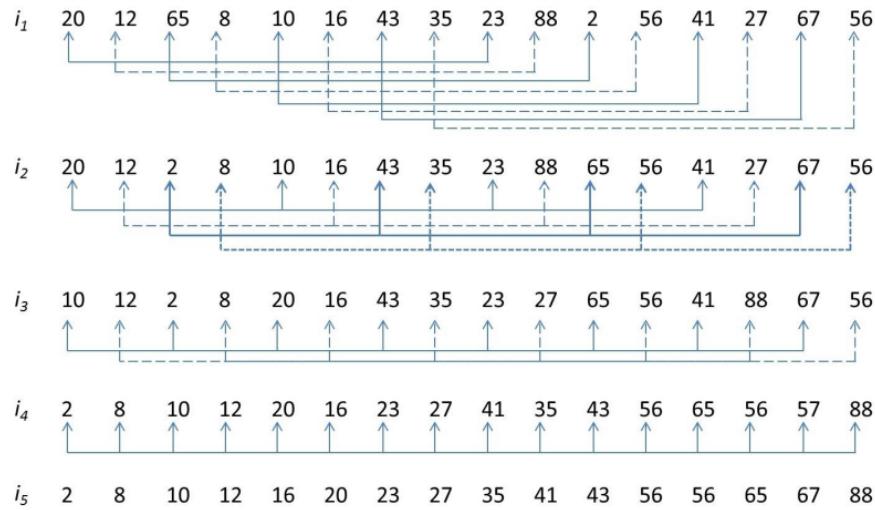
1. Rozděl vstupní pole na posloupnosti o **délce 2** (prvky musí být co nejdále od sebe),
2. Každou posloupnost seřaď pomocí bubble sort,
3. Rozděl vstupní pole na posloupnosti o **délce 4** (prvky musí být co nejdále od sebe),
4. ...
5. Rozděl vstupní pole na posloupnost o **délce pole** a seřaď je pomocí bubble sort.

nejlepší délky posloupností - 1, 4, 10, 23, 57, 132, 301, 701, 1750.

Vlastnosti:

- **nestabilní**,
- měla by být **přirozená**,
- pracuje **in situ**,
- časová složitost je v nejhorším případě **kvadratická**, ale vybráním vhodné řady lze snížit na **$O(n^{(3/2)})$** nebo **$O(n^*(\log n)^2)$** .

<https://www.algoritmy.net/article/154/Shell-sort>



Merge sort

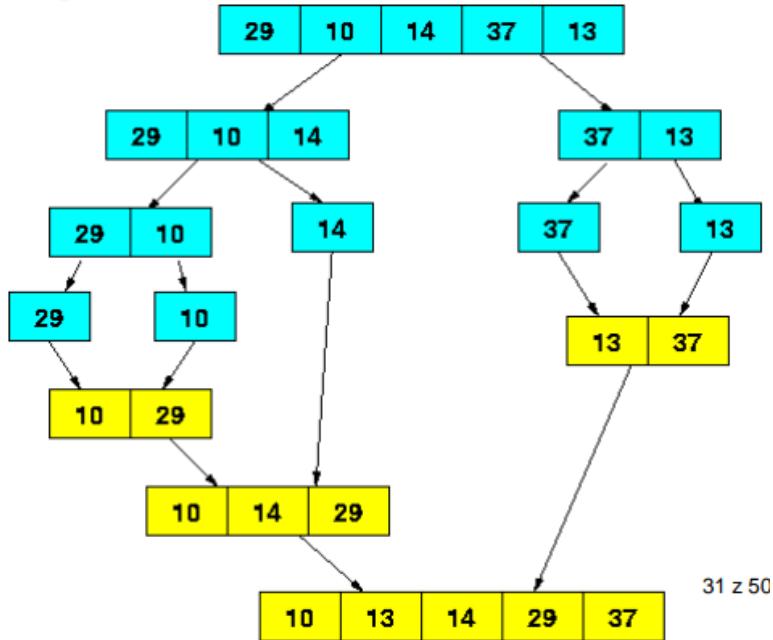
Další z algoritmů typu **rozděl a panuj**. Princip:

1. Postupně **půlíme** pole (v případě lichého počtu má jedna část o prvek navíc) dokud není o velikosti 1.
2. Poté **spojujeme** vždy dvě sousední pole tak, aby **vzniklo jedno seřazené** pole. Toto seřazování je jednoduché, protože již **spojujeme seřazená pole**, tudíž máme dva ukazatele do těchto polí a vždy vybíráme prvek, který je mezi polí **nejmenší/největší**.

Vlastnosti:

- **stabilní**,
- spíš není přirozená, vždy se musí dělit a poté setřídovat,
- **in situ**, ale prostorová složitost je **logaritmická O(log(n))** pro uchovávání hranic polí po dělení - lze použít zásobník, ale jednodušší je použití **rekurze**,
- **linearitmická O(n*logn)** časová složitost

<https://www.algoritmy.net/article/13/Merge-sort>



31 z 50

Shrnutí

Název		Časová složitost			Dodatečná paměť	Stabilní	Přirozená	Metoda
Název	Znám jako	Minimum	Průměrně	Maximum				
bublinkové řazení	bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	záměna
řazení haldou	heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	ne	ne	halda, záměna
řazení vkládáním	insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	vkládání
řazení slučováním	merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ano	ano	slučování
rychlé řazení	quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	ne	ne	záměna
řazení výběrem	selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	zprav. ne	ne	výběr
Shellovo řazení	shell sort	$O(n^{1+\frac{c}{\sqrt{m}}})$ [10]		$O(n \log^2 n)$ [10]	$O(1)$	ne	ano	vkládání
comb sort	(hřebenové řazení)	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(1)$	ne	ano	záměna
introspektivní třídění	Introsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ne	?	záměna, výběr

Řazení dle více klíčů

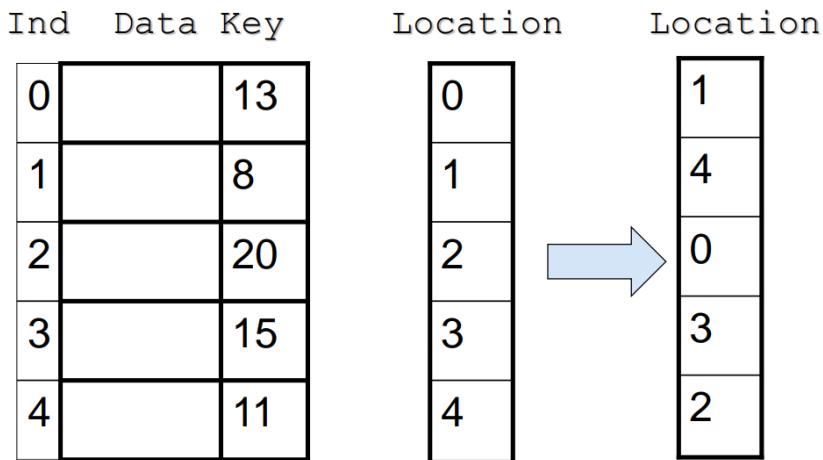
Lze řešit opakováním řazením, které se podobá se radixovému (přihrádkovému) řazení např. pro den, měsíc a rok. Musí platit:

- řadíme **od klíče s nejmenší prioritou po klíč s největší prioritou**,
- nutné použít **stabilní řadící metodu**.

Druhou možností je provádět řazené **pouze jednou** ale současně **porovnávat všechny klíče** počínaje od toho s největší prioritou. Z čehož plyne třetí možnost, a to sloučit klíče do jednoho (aglomerovaný klíč). U příkladu se dny, měsíci a roky se jedná o rodné číslo bez posledních 4 znaků - RRMMDD (pozor ženy mají MM zvýšené o 50)

Řazení bez přesunu položek

Vhodné v případech, kdy řadíme **velké objekty** a jejich **přesuny v paměti** jsou **drahé**. K tomu, abychom položky museli přesunovat, používáme **pomocné pole - pořadník**. Po dokončení řazení **pořadník udává**, v jakém **pořadí** by měly být seřazeny **položky původního pole**. Tj. na první pozici pořadníku je index prvního prvku seřazeného pole atd.



Pole seřazené pomocí **pořadníku** lze také **zřetězit**, v jednotlivých prvcích musí být **vyhrazená paměť** pro **index následujícího prvku**. Zřetězené prvky pak lze **procházet sekvenčně principem spojových seznamů**, nebo je lze **převést do seřazeného pole** (to je možné i bez zřetězení podle pořadníku - MacLarenův algoritmus)

Vyhledávání

Vyhledávání lze realizovat jako:

- sekvenční vyhledávání v **neseřazeném poli**,
- sekvenční vyhledávání v **neseřazeném poli se zarážkou**,
- sekvenční vyhledávání v **seřazeném poli**,
- sekvenční vyhledávání v poli **seřazeném podle pravděpodobnosti vyhledání** klíče (nejpravděpodobnější klíče jsou na začátku pole),
- sekvenční vyhledávání v poli s **adaptivním uspořádáním podle četnosti vyhledání** (prvky se s četností vyhledávání přesouvají k začátku pole),
- **binární vyhledávání v seřazeném poli**,
- **binární vyhledávání pomocí stromů**,
- vyhledávání pomocí **stromů s více klíči ve vrcholech**,
- vyhledávání pomocí tabulky s rozptýlenými položkami (hash table).

Při vyhledávání sledujeme doby vyhledání (**kritéria**) při **úspěšném/neúspěšném** vyhledávání.

- **minimální**,
- **maximální**,
- **průměrná/střední**.

Sekvenční vyhledávání

Prvky pole se prochází jeden po druhém.

V neseřazeném poli

Nejrychleji jsou nalezeny položky na počátku vyhledávání. Jednoduchá implementace (**procházení polem a test na hodnotu**).

- **minimální čas úspěšného vyhledání = 1**
- **maximální čas úspěšného vyhledání = N**
- **průměrný čas úspěšného vyhledání = N/2**
- Čas **neúspěšného** vyhledání = N

V neseřazeném poli se zarážkou

Na konci struktury je "**zarážka**", což je **hledaný klíč**, tedy hodnota je vždy nalezena. Toto přidává **urychljení** v tom, že není potřeba po každém porovnání klíčů **testovat konec struktury**, pouze se otestuje na konci vyhledávání jestli se **nalezla zarážka nebo hledaná položka**.

V seřazeném poli

Jakmile se narazí na hodnotu klíče, která je **větší než hledaný klíč**, vyhledávání se ukončuje jako **neúspěšné**. Tato metoda se **urychlí pouze pro neúspěšné hledání**. Vyhledávání v seřazeném poli lze ale urychlit např. **půlením intervalů**, viz dále. Funguje pouze pro klíče, u kterých **existuje relace uspořádání**. Problematické jsou operace vkládání a odebrání prvků, je nutné pole **znovu seřadit**, respektive **posunout segment pole v paměti**. Odebírání lze řešit **zaslepěním** (označením indexu za nevyužitý, respektive nastavení jeho hodnoty na hodnotu, která nebude hledána).

Podle četnosti přístupu

Nejčastěji hledané položky **jsou na začátku struktury - pole/seznamu** (používá se k tomu **počítadlo vyhledávání** a jednou za čas je pole podle počítadla seřazeno). Druhou variantou je, že po nalezení položky se **nalezená položka prohodí se svým levým sousedem** (není potřeba počítadlo, nejčastěji hledané položky se tak přesouvají k začátku struktury) - **adaptivní rekonfigurace podle četnosti vyhledávání**. Postupy lze využít i u prvků, které mezi sebou nemají definovanou relaci uspořádání.

Podle pravděpodobnosti

Jedná se o podobný případ viz sekce výše, kde pravděpodobnosti vyhledávání jsou známé předem a podle toho je pole seřazené (poté se již uspořádání nemění).

Nesekvenční vyhledávání

Přístup k hledaným položkám může být náhodný.

Binární vyhledávání v poli (seřazeném)

Provádí se nad **seřazenou množinou klíčů** s náhodným přístupem (pole). Metoda připomíná metodu **půlení intervalů**. Složitost je při nejhorším **logaritmická O(log n)**. Pole **půlíme** a podle **prostřední hodnoty** prohledáváme **levou nebo pravou část**. Opět je zde stejný problém s **vkládáním a odebíráním prvků** z pole.

Dijkstrova varianta binárního vyhledávání

Vychází z předpokladu, že v poli může být **více položek se shodným klíčem**. Hledá se tedy **nejpravější nebo nejlevější klíč** (ostatní pak lze získat jednoduše sekvenčním průchodem od nalezeného indexu). Příklady pro vyhledávání nepravějšího výskytu:

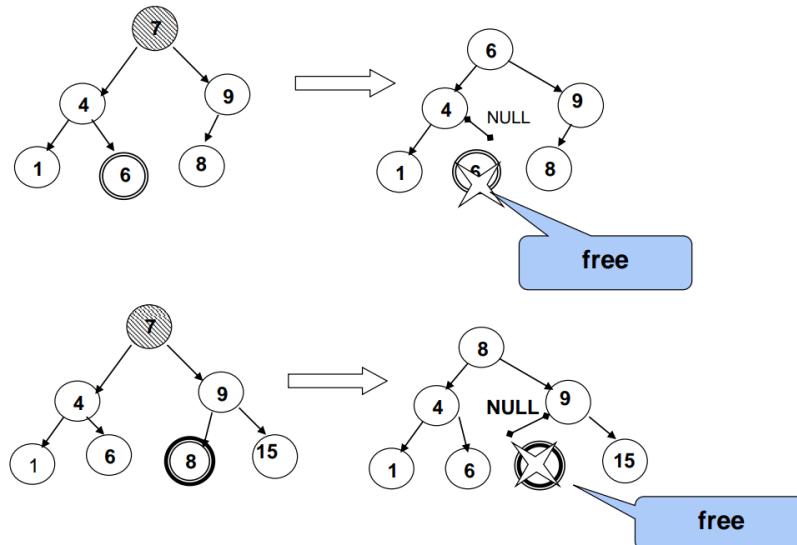
- V poli: 1,2,3,4,5,5,6,6,6,8,9,13 najde algoritmus klíč **K=6** na pozici **8** (počítáno od 0).
- V poli: 1,1,1,1,1,1,1,1,1,1,2 najde algoritmus klíč **K=1** na pozici **9**.

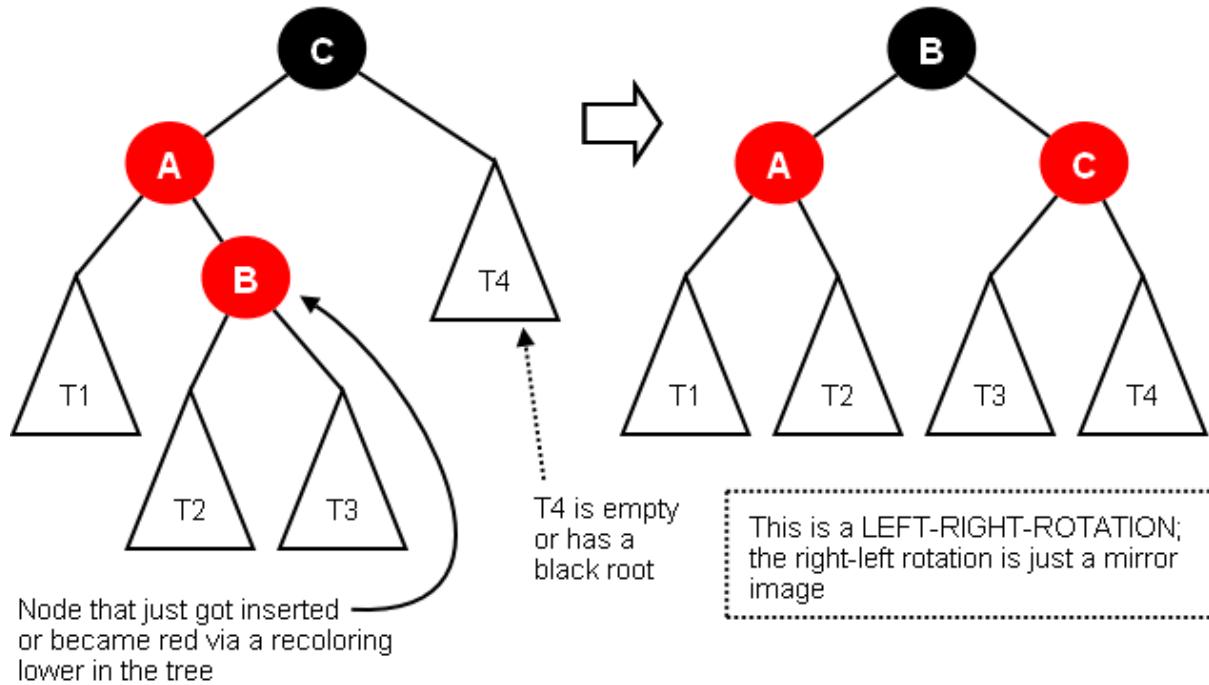
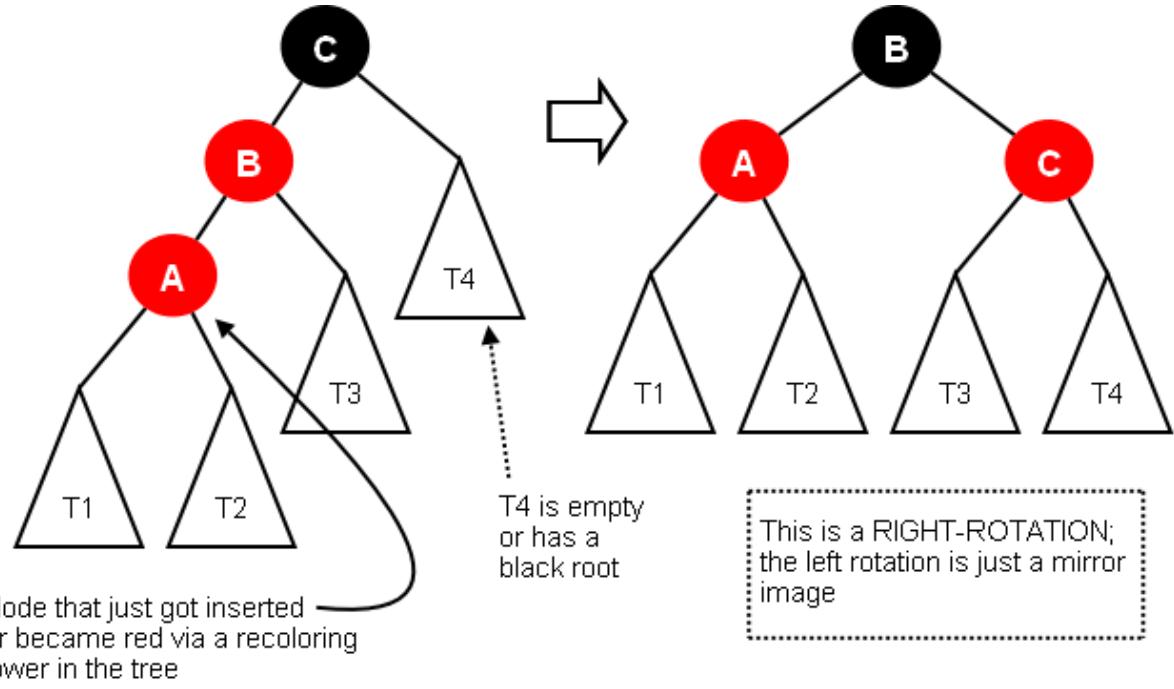
Binární vyhledávání v binárním vyhledávacím stromu (binary search tree)

Podobné binárnímu vyhledávání v seřazeném poli. Je-li **vyhledaný klíč roven kořeni**, vyhledávání končí **úspěšně**. Je-li **klíč menší než klíč kořene**, pokračuje vyhledávání v **levém podstromu**, je-li **větší**, pokračuje v **pravém podstromu**.

Vyhledávání končí **neúspěšně**, je-li **prohledávaný (pod)strom prázdný**. Výhodou uspořádání do BVS je **snazší vkládání a mazání prvků**. Při vkládání uzlu je uzel vložen na listovou úroveň (pokud se jedná o samovyvažující se strom může být nutné **provést rotace**, aby **bylo dodrženo vyvážení**, viz obrázek). U mazání je nutné dodržet uspořádání stromu a pokud je mazán vnitřní uzel, musí být nahrazen některým uzel z listové úrovni takto:

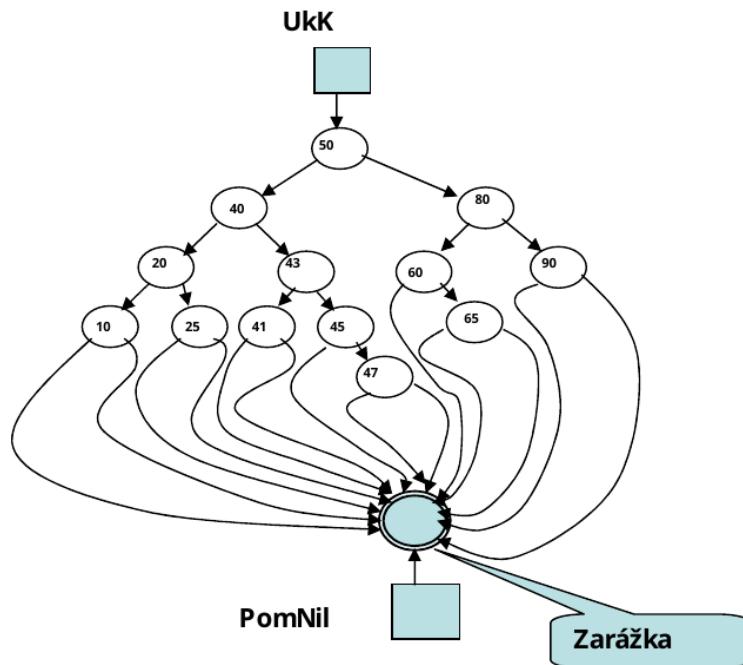
- **nejpravější uzel levého podstromu** rušeného uzlu (**maximum v levém podstromu**),
- **nejlevější uzel pravého podstromu** rušeného uzlu (**minimum v pravém podstromu**).





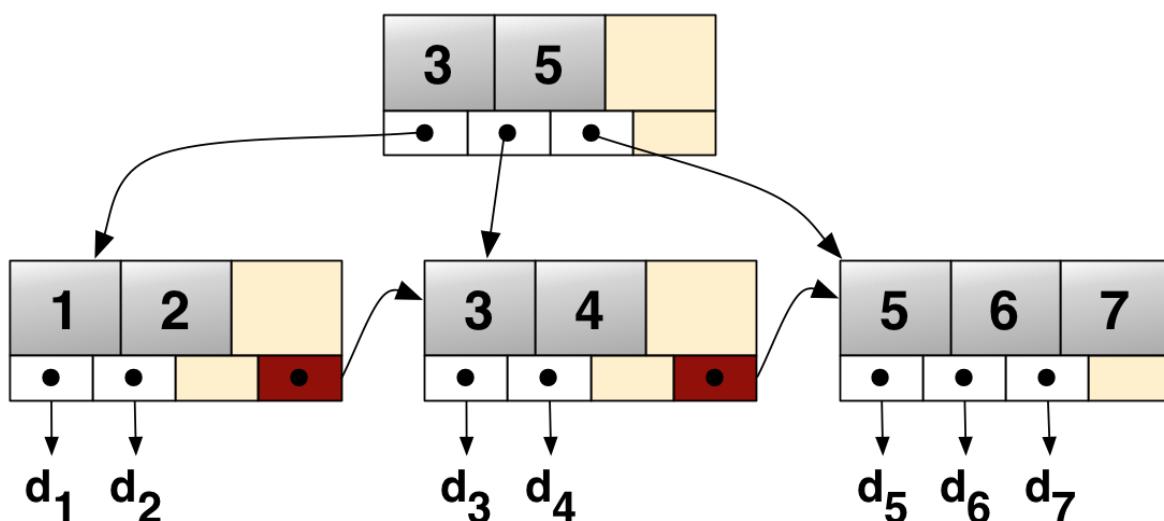
BVS se zarážkou

Funguje podobně jako u pole, všechny listové uzly ukazují na zarážku. Na konci vyhledávání musí být provedený test, jestli se vyhledala hledaná hodnota nebo zarážka.



Vyhledávání pomocí stromů s více klíči ve vrcholech

Vnitřní (interní) vrcholy obsahují libovolný nenulový počet klíčů. Pokud ve vrcholu (uzlu) leží k klíčů, pak má $k+1$ synů (s_0, \dots, s_k). Hledání cesty v uzlu se provádí obvykle sekvenčně/binárně (už nelze rozlišit pouze pravý/levý podstrom). **Snižuje se hloubka** stromu ale **zvyšuje se náročnost hledání cesty**. Vnitřní uzly nemusí nést data a slouží pouze pro vyhledávání, data jsou uložena až na listové úrovni (**B+ stromy**, které se využívají v DB, file systémech, ...). Počet klíčů ve vrcholech se většinou může pohybovat od do určitého počtu - např. (**a,b**) stromy (kořen má **2** až **b** synů, ostatní vnitřní vrcholy **a** až **b** synů, **a >= 2, b >= 2a-1**). Při **vkládání** se uzly **štěpí** (je-li potřeba), při **mazání** se **slučují** (je-li potřeba). Slovník lze například implementovat pomocí **písmenkového stromu** (trie), kde se každý uzel větví dle počtu písmen v abecedě, což umožňuje sdílet části stromů mezi více slov a vyhledání je velmi rychlé.



Vyhledávání v tabulkách s rozptýlenými položkami (TRP - hash table)

Základem je princip **tabulky s přímým přístupem**. Využívá se zde mapovací funkce, která **rovnoměrně mapuje klíče na indexy** v paměti (políčka tabulky). Což v ideálním případě bez kolizí znamená **konstantní $O(1)$** časovou složitost vyhledávání. Problém je vznik kolizí (klíče, které jsou mapovány na stejný index), poté musí být vyhledávání dokončeno sekvenčně. V **extrémním případě** tak může být rychlosť vyhledávání **lineární $O(n)$** . Vyhledávání v TRP má **index-sekvenční** charakter.

- **explicitní** řetězení synonym: spojový seznam, binární vyhledávací strom,
- **implicitní** řetězení synonym: ukládání synonym na první/další volné místo (dle kroku) v tabulce (poli).

Vyhledávání v textu

Hledaným klíčem je řetězec (několik znaků) textu.

Naivní (Brute-force) algoritmus

Algoritmus **porovnává** symboly textu a vzorku **zleva doprava**, při **neshodě** symbolů **posune** vzorek o **jednu** pozici doprava.

- složitost: **$O(m*n)$** , kde **m** je počet znaků textu a **n** je počet znaků vzorku, jedná se ale o **nejhorší případ** a většinou je porovnání daleko rychlejší.

Knuth-Morris-Prattův algoritmus

Algoritmus využívá princip **konečného automatu**. Porovnávání textu stále probíhá **zleva doprava**. Při neshodě se nevrací v textu zpět (posun o 1 v předchozím algoritmu), ale snaží se **posunout vzorek** tak, aby symbol textu, na kterém došlo k neshodě, porovnal s **jiným symbolem vzorku**, který předchází symbolu s neshodou (pokud žádný takový neexistuje, posouvá vzorek na začátek).

Text: clanekokokosu

Aktuální přiložení vzorku: kokos

Další možné přiložení: kokos

- složitost **$O(m)$** pro konstrukci automatu, **$O(n)$** pro porovnání, celkově **$O(m+n)$** .

Boyer-Mooreův algoritmus

Přikládá vzorek k textu **zleva doprava**, ALE porovnává **zprava doleva**, což umožňuje provádět větší skoky (některé **symboly textu nemusí být vůbec porovnány** se symboly vzorku). Pokud se při porovnávání narazí na **symbol**, který se ve vzorku **nenachází**, lze vzorek **posunout o jeho délku**, **jinak** je nutné vzorek **posunout** tak, aby byl **přiložen** na další **nejpravější výskyt** daného symbolu. Dále lze přidat **posun na základě podřetězce**.

Rabin-Karpův algoritmus

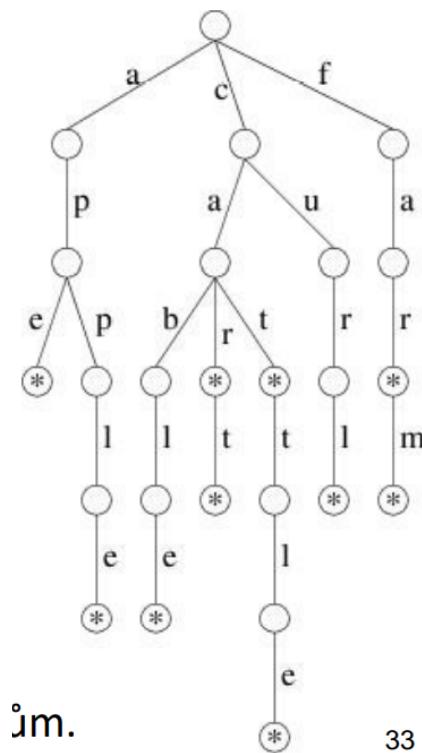
Vyhledávání vzorku založené na hashování. Postup:

- Posouváme okénko délky **m** (délky vzorku) po textu a **počítáme hash** pro danou část textu.
- Je-li **hash shodný s hashem vzorku**, porovnáme danou část textu se vzorkem **znak po znaku**.

Aby tato metoda byla efektivní, musíme umět **rychle počítat hash** (v konstantním čase). Lze zařídit tak, že který **opouštíme** od hashe určitým způsobem **odečteme** a znak, na který **najízdíme** určitým způsobem **přičteme** (Ordinální hodnota asi nebude úplně efektivní ale bude taky fungovat, většinou jsou ale znaky násobeny nějakým **polynomem**).

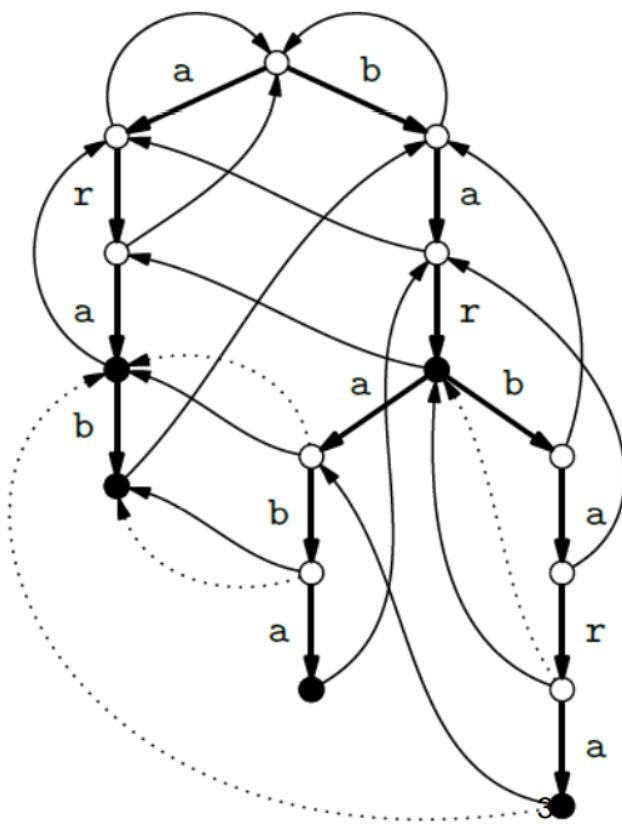
Písmenkové stromy - hledání více vzorků

Umožňují současné vyhledávání více vzorků v textu. Na jedné cestě od kořene se může nacházet více vzorků (na obrázku jsou konce slov označeny hvězdičkou).



Algoritmus Aho-Corasicková

- Postupujeme automatem po dopředných hranách, pokud můžeme.
- Nelze-li použít žádnou dopřednou hranu, vracíme se po zpětných hranách.
- Pokud se dostaneme zpět až do kořene a ani zde nelze jít s daným symbolem žádnou dopřednou hranou, symbol je zahozen.
- V každém stavu zkонтrolujeme, zda neodpovídá konci slova. Pokud ano, ohlášíme výskyt. Z každého stavu pomocí zkratek nalezneme také všechny sufiksy, které jsou také slovem a ohlášíme.



odkazy

- Animace řazení:
<https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

31. Pravděpodobnost a statistika (základní pojmy, náhodná veličina a vektor, rozdělení pravděpodobnosti, generování pseudonáhodných čísel, bodové a intervalové odhady parametrů, testování hypotéz, regresní a korelační analýza).

Základní pojmy

Ω - množina všech hodnot kterých může **náhodná veličina** X nabývat - **základní prostor**. Základní prostor pro dva po sobě následující hody vypadá následovně $\Omega = \{ (\text{rub}, \text{rub}), (\text{líc}, \text{líc}), (\text{rub}, \text{líc}), (\text{líc}, \text{rub}) \}$

Náhodný jev

libovolná podmnožina Ω

- jev nemožný: \emptyset za daných podmínek jev **nemůže nastat** (na dvou tradičních (šestibokých) hracích kostkách nám padne součet 30),
- jev jistý: Ω - za daných podmínek jev **nastane vždy** (při hodu mincí padne rub nebo líc).

Průnik jevů

Průnik jevů **A** a **B** je jev, který nastane právě tehdy, když **nastanou** jevy **A** a **B** **současně**. Značíme jej $A \cap B$. Pokud je $A \cap B = \emptyset$, mluvíme o jevech **disjunktních** (**neslučitelných**)

Sjednocení jevů

Sjednocení jevů **A** a **B** je jev, který nastane právě tehdy, když **nastane alespoň jeden** z jevů A a B. Značíme jej $A \cup B$.

Opačný jev

Opačný jev (doplňek) k jevu **A** je jev, který nastane právě tehdy, když **nenastane** jev **A**. Značíme **A'** a platí $A' = \Omega \setminus A$.

Úplný systém jevů

Jevy **A₁**, **A₂**, ... tvoří úplný systém jevů, jestliže $A_1 \cup A_2 \cup \dots = \Omega$. Pokud navíc platí $A_i \cap A_j = \emptyset$, $\forall i \neq j$, jedná se o **úplný systém neslučitelných** jevů.

Axiomatická definice pravděpodobnosti

Nechť (Ω, \mathcal{A}) je jevové pole a P je množinová funkce definovaná na \mathcal{A} s vlastnostmi:

- 1) $P(\Omega) = 1$,
- 2) $P(A) \geq 0 \forall A \in \mathcal{A}$,
- 3) jestliže $A_k \in \mathcal{A}, k = 1, 2, \dots$, jsou navzájem disjunktní jevy $(A_i \cap A_j = \emptyset \text{ pro } i \neq j)$, pak

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i).$$

Funkci P nazveme **pravděpodobností** a trojici (Ω, \mathcal{A}, P) nazveme **pravděpodobnostním prostorem**.

Klasická pravděpodobnost

Klasická pravděpodobnost je definována jako **podíl počtu příznivých jevů ku počtu všech možných jevů**. $P(A) = |A|/|\Omega|$, kde:

- $|A|$ značí počet prvků množiny příznivých jevů,
- $|\Omega|$ značí počet prvků všech možných jevů - velikost základního prostoru.

Vlastnosti pravděpodobnosti

Nechť (Ω, \mathcal{A}, P) je pravděpodobnostní prostor. Pak pravděpodobnost P má následující vlastnosti:

- (1) $P(\emptyset) = 0$,
- (2) $A, B \in \mathcal{A}, A \cap B = \emptyset \Rightarrow P(A \cup B) = P(A) + P(B)$
- (3) $A, B \in \mathcal{A}, A \subseteq B \Rightarrow P(B - A) = P(B) - P(A)$
- (4) $A, B \in \mathcal{A}, A \subseteq B \Rightarrow P(A) \leq P(B)$
- (5) $A \in \mathcal{A} \Rightarrow 0 \leq P(A) \leq 1$
- (6) $A \in \mathcal{A} \Rightarrow P(\bar{A}) = 1 - P(A)$
- (7) $A, B \in \mathcal{A} \Rightarrow P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- (8) $A_1, \dots, A_n \in \mathcal{A} \Rightarrow P(\bigcup_{i=1}^n A_i) \leq \sum_{i=1}^n P(A_i)$

$$(9) \quad A_1, \dots, A_n \in \mathcal{A} \Rightarrow P(\bigcup_{i=1}^n A_i) = \sum_{i=1}^n P(A_i) - \sum_{i=1}^{n-1} \sum_{j=i+1}^n P(A_i \cap A_j) \\ + \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^n P(A_i \cap A_j \cap A_k) \\ + \dots (-1)^{n-1} P(A_1 \cap \dots \cap A_n)$$

Podmíněná pravděpodobnost

O proběhlém pokusu máme doplňující informace a může tak lépe určit jeho pravděpodobnost.

Nechť (Ω, \mathcal{A}, P) je pravděpodobnostní prostor, $B \in \mathcal{A}$, $P(B) > 0$. Pak číslo

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

nazveme **podmíněnou pravděpodobností** jevu A za podmínky, že nastal jev B .

Bayesův vzorec

Mějme dva náhodné jevy A a B , přičemž $P(B) \neq 0$. Potom platí:

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}, \text{ kde}$$

- $P(A | B)$ je podmíněná pravděpodobnost jevu A za předpokladu, že nastal jev B
- $P(B | A)$ je podmíněná pravděpodobnost jevu B podmíněná výskytem jevu A
- $P(A)$ a $P(B)$ jsou pravděpodobnosti jevů A a B

Lze jednoduše vyjádřit z podmíněné pravděpodobnosti dosazením za průnik.

Sčítání pravděpodobností

$$\begin{aligned} P(A \cup B \cup C) &= \\ P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C) \end{aligned}$$

Pro **nezávislé** jevy jsou průniky **nulové**.

Náhodná veličina a vektor

Náhodná veličina je číselné ohodnocení náhodného pokusu (který dokážeme číselně ohodnotit). **Náhodnou veličinu** značíme **velkými písmeny X, Y, Z**.

Pravděpodobnost se značí **malými x, y, z, p**.

Pravděpodobnost, že náhodná veličina X nabývá hodnoty x zapíšeme jako

$$P(X = x).$$

Podobně lze interpretovat $P(X < x)$, $P(X \geq x)$, atd.

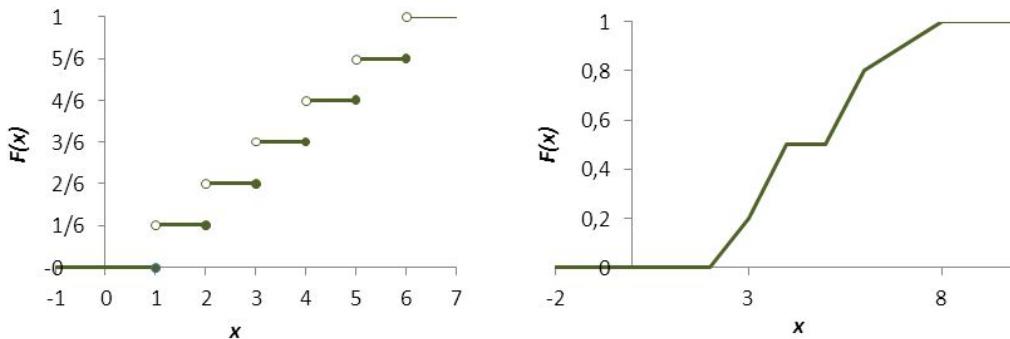
Rozlišujeme dva základní typy náhodných veličin:

- **diskrétní**,
- **spojitá**

Distribuční funkce

Popisuje **rozdělení (pravděpodobnostního chování)** náhodných veličin. Je **neklesající a zprava spojitá**. Distribuční funkce $F(x)$ náhodné veličiny X přiřazuje každému reálnému číslu x pravděpodobnost, že náhodná veličina X nabude hodnoty **menší nebo rovné číslu x** .

$$F(x) = P(X \leq x)$$



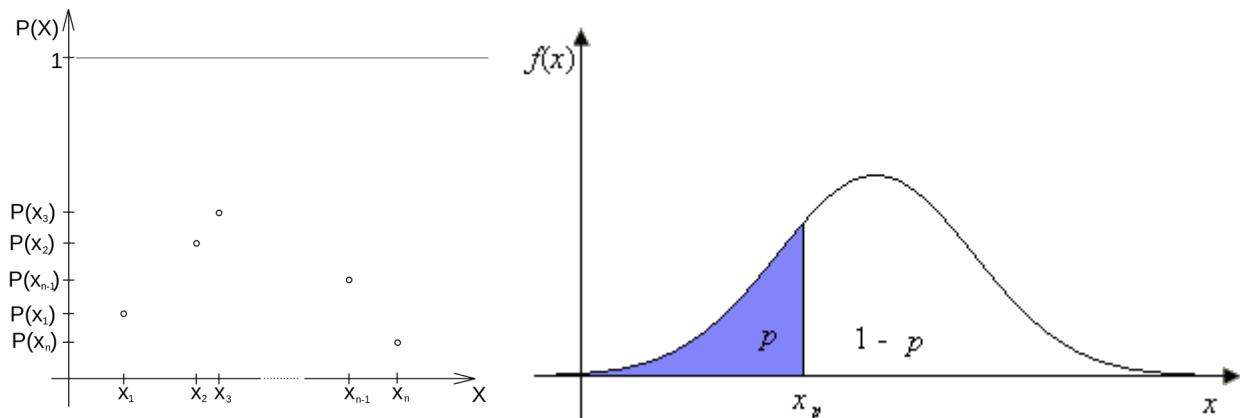
U **diskrétní** náhodné veličiny je distribuční funkce **schodovitá** (vlevo).

Vlastnosti:

- $0 \leq F(x) \leq 1$ pro $\forall x \in \mathbb{R}$
- $F(x)$ je neklesající a zprava spojitá funkce.
- $\lim_{x \rightarrow -\infty} F(x) = 0, \lim_{x \rightarrow \infty} F(x) = 1$
- $P(a < X \leq b) = F(b) - F(a)$ pro každé $a, b \in \mathbb{R}, a < b$
- $P(X = x) = F(x) - \lim_{t \rightarrow x^-} F(t)$
- F má nejvýše spočetně mnoho bodů nespojitosti

Funkce hustoty pravděpodobnosti (Pravděpodobnostní funkce u diskrétní)

Určuje **rozdělení pravděpodobnosti náhodné veličiny** (u **diskrétní** náhodné veličiny lze vyjádřit tak, že se určí pravděpodobnost $P(x)$ pro všechna x definičního oboru veličiny X). Značíme jí $f(x)$ a získá se první derivací distribuční funkce $F'(x) = f(x)$.



- $f(x) \geq 0$
- $f(x) = \frac{dF(x)}{dx}$
- $\int_{-\infty}^{\infty} f(x) dx = 1$
- $P(a \leq X \leq b) = \int_a^b f(x) dx$

Vlastnosti:

Charakteristiky náhodné veličiny

Střední hodnota $E(X)$

Střední hodnota **diskrétní** náhodné veličiny (1. obrázek) je **pravděpodobnostně vážený průměr všech jejích možných hodnot**. Pro **spojitou** náhodnou proměnnou je součet nahrazen **integrálem**.

x	0	100	200	400
$p(x)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

$$E(X) = \sum_x x \cdot p(x) = 100$$

$$E(X) = \int_M xf(x)dx$$

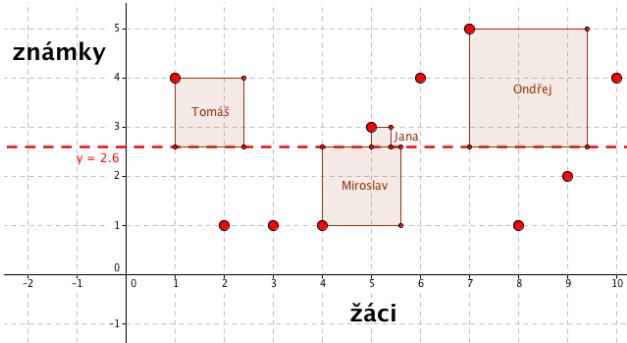
$$E(a) = a$$

$$E(aX + b) = aE(X) + b$$

$$E(X \pm Y) = E(X) \pm E(Y)$$

Rozptyl $D(X)$

Také **střední kvadratická odchylka**. Jedná se o **charakteristiku variability** rozdělení pravděpodobnosti **náhodné veličiny**, která vyjadřuje variabilitu rozdělení **souboru náhodných hodnot kolem její střední hodnoty**. Jedná se o **součet obsahů čtverců jednotlivých hodnot dle vzdálenosti od střední hodnoty**.



$$D(X) = E(X^2) - [E(X)]^2,$$

$$E(X^2) = \sum_{x \in M} x^2 \cdot p(x).$$

$$\sigma^2 = \int_{-\infty}^{\infty} [x - E(X)]^2 p(x) dx = \int_{-\infty}^{\infty} x^2 p(x) dx - [E(X)]^2$$

$$D(X) \geq 0$$

$$D(aX + b) = a^2 D(X)$$

$$D(X \pm Y) = D(X) + D(Y) \text{ pro nezávislé náhodné veličiny } X, Y$$

Směrodatná odchylka $\sigma(X)$

Směrodatná odchylka vypovídá o tom, **nakolik se od sebe navzájem typicky liší jednotlivé případy** v souboru zkoumaných hodnot. Vypočítá se jako odmocnina z rozptylu

$$\sigma(X) = \sqrt{D(X)}$$

Náhodný vektor

Často je výsledkem pokusu **n-tice** reálných čísel - **vektor**. Poté můžeme zjednodušeně říct, že se jedná o **vektor více náhodných veličin**, které jsou definované na pravděpodobnostním prostoru.

Nechť X, Y jsou náhodné veličiny definované na stejném pravděpodobnostním prostoru (Ω, \mathcal{A}, P) . Pak $(X, Y)'$ nazveme **náhodným vektorem**.

Sdružená distribuční funkce

Obdobná distribuční funkci pro jednu náhodnou veličinu s tím rozdílem, že je **n rozměrná** (n je počet veličin náhodného vektoru). Platí také obdobná pravidla.

$$\lim_{x \rightarrow -\infty} F(x, y) = \lim_{y \rightarrow -\infty} F(x, y) = 0$$

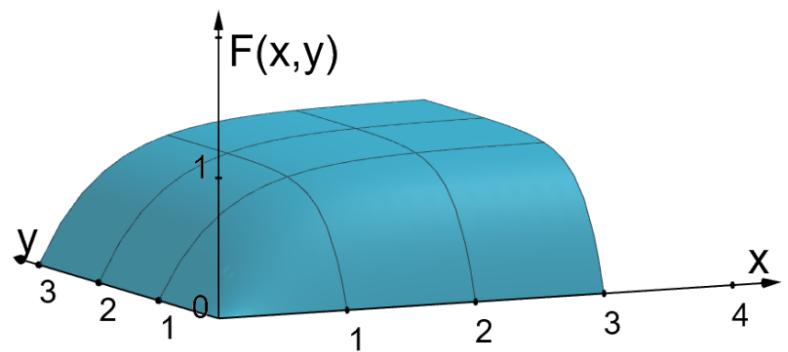
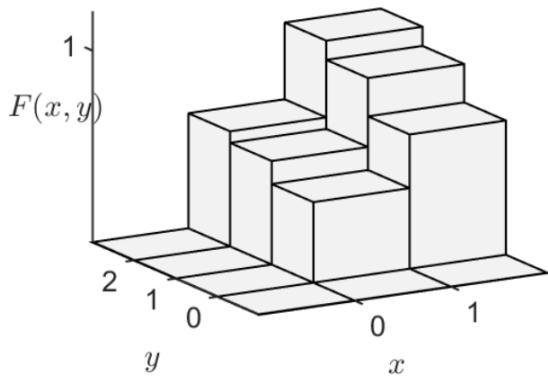
$$\lim_{(x,y) \rightarrow (\infty, \infty)} F(x, y) = 1$$

$$\sum_x \sum_y p(x, y) = 1.$$

$$F(x, y) = \sum_{u \leq x} \sum_{v \leq y} p(u, v).$$

$$F(x, y) = \int_{-\infty}^x \int_{-\infty}^y f(u, v) \, dv \, du.$$

Příklad **diskrétní vlevo** a **spojité vpravo**:



Marginální distribuční funkce

Jedná se o distribuční funkci **jedné z náhodných proměnných vektoru**, pokud jsou náhodné jevy popsané **zbylými proměnnými** jisté (100%).

Je-li $F(x, y)$ sdružená distribuční funkce veličin X a Y , pak **marginální distribuční funkce** veličiny X a veličiny Y je

$$F_X(x) = P(X \leq x) = \lim_{y \rightarrow \infty} F(x, y), \quad x \in \mathbb{R},$$

$$F_Y(y) = P(Y \leq y) = \lim_{x \rightarrow \infty} F(x, y), \quad y \in \mathbb{R}.$$

Sdružená pravděpodobnostní funkce (sdružená hustota pst.)

Marginální pravděpodobnostní funkce

Všechny ostatní náhodné veličiny, až na jednu (ta, u které zjišťujeme marginální pravděpodobnostní funkci), jsou jisté (100%)

$$p_X(x) = P(X = x) = \sum_y p(x, y), \quad x \in \mathbb{R},$$

$$p_Y(y) = P(Y = y) = \sum_x p(x, y), \quad y \in \mathbb{R}.$$

$x \setminus y$	0	1	2
0	0,42	0,12	0,06
1	0,28	0,08	0,04

$$p_X(x) = \sum_y p(x, y)$$

x	0	1
$p_X(x)$	0,6	0,4

$$p_Y(y) = \sum_x p(x, y)$$

y	0	1	2
$p_Y(y)$	0,7	0,2	0,1

$$f_X(x) = \int_{-\infty}^{\infty} f(x, y) \, dy = \int_0^{\infty} e^{-x-y} \, dy = e^{-x}, \quad x > 0$$

$$f_Y(y) = \int_{-\infty}^{\infty} f(x, y) \, dx = \int_0^{\infty} e^{-x-y} \, dx = e^{-y}, \quad y > 0$$

Podmíněné rozdělení

Mějme diskrétní náhodný vektor $(X, Y)'$ se sdruženou pravděpodobnostní funkcí $p(x, y)$. Funkci

$$p(x|y) = P(X = x | Y = y) = \frac{p(x, y)}{p_Y(y)}, \quad p_Y(y) > 0,$$

nazveme **podmíněnou pravděpodobnostní funkcí** veličiny X za podmínky, že veličina Y nabyla hodnoty y .

Podobně

$$p(y|x) = P(Y = y | X = x) = \frac{p(x, y)}{p_X(x)}, \quad p_X(x) > 0,$$

nazveme podmíněnou pravděpodobnostní funkcí veličiny Y za podmínky, že veličina X nabyla hodnoty x .

Mějme spojitý náhodný vektor $(X, Y)'$ se sdruženou hustotou pravděpodobnosti $f(x, y)$. Funkci

$$f(x|y) = \frac{f(x, y)}{f_Y(y)}, \quad f_Y(y) > 0,$$

nazveme **podmíněnou hustotou pravděpodobnosti** veličiny X za podmínky, že veličina Y nabyla hodnoty y .

Podobně

$$f(y|x) = \frac{f(x, y)}{f_X(x)}, \quad f_X(x) > 0,$$

nazveme podmíněnou hustotou pravděpodobnosti veličiny Y za podmínky, že veličina X nabyla hodnoty x .

Mějme (diskrétní nebo spojitý) náhodný vektor $(X, Y)'$. Pak veličiny X, Y jsou **nezávislé**, právě když
 Mějme (diskrétní nebo spojitý) náhodný vektor $(X, Y)'$. Pak veličiny X, Y jsou **nezávislé**, právě když

$$F(x, y) = F_X(x) \cdot F_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

$$F(x, y) = f_X(x) \cdot f_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

Mějme diskrétní vektor $(X, Y)'$. Pak veličiny X, Y jsou **nezávislé**, právě když

$$p(x, y) = p_X(x) \cdot p_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

Mějme spojitý náhodný vektor $(X, Y)'$. Pak veličiny X, Y jsou **nezávislé**, právě když

$$f(x, y) = f_X(x) \cdot f_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2.$$

Nezávislost

Střední hodnota a rozptyl

$$E(X) = \sum_x x \cdot p_X(x), \quad E(Y) = \sum_y y \cdot p_Y(y),$$

$$E(X) = \int_{-\infty}^{\infty} x \cdot f_X(x) dx, \quad E(Y) = \int_{-\infty}^{\infty} y \cdot f_Y(y) dy,$$

$$E(XY) = \sum_x \sum_y xy \cdot p(x, y),$$

$$E(XY) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} xv \cdot f(x, y) dx dy$$

$$D(X) = E(X^2) - [E(X)]^2$$

$$D(Y) = E(Y^2) - [E(Y)]^2,$$

kde

$$E(X^2) = \sum_x x^2 \cdot p$$

$x \setminus y$	3	4	5	$p_X(x)$
1	0,3	0,24	0,06	0,6
2	0,2	0,16	0,04	0,4
$p_Y(y)$	0,5	0,4	0,1	1

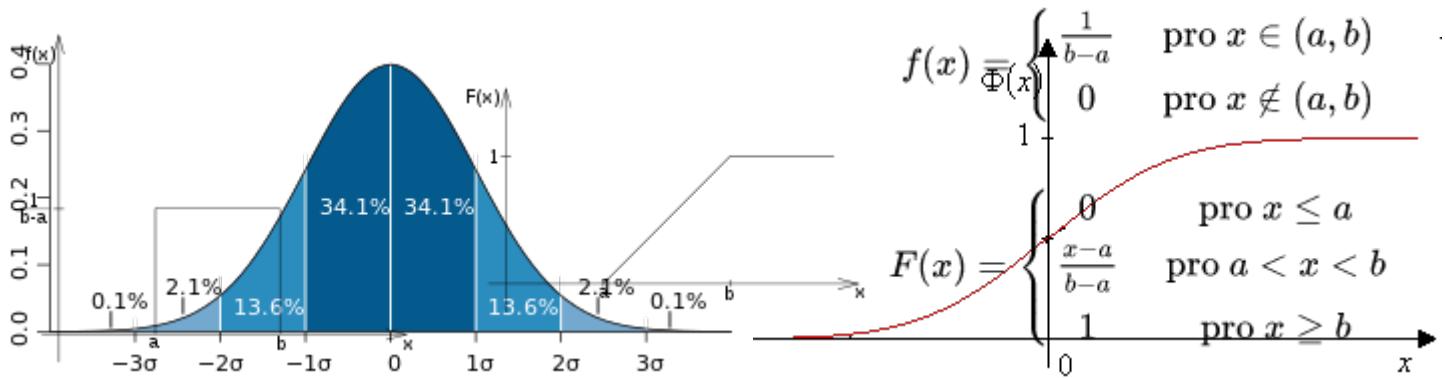
$$D(Y) = E(Y^2) - [E(Y)]^2,$$

$$p(x, y) = p_X(x) \cdot p_Y(y) \text{ pro } \forall (x, y) \in \mathbb{R}^2$$

Náhodné veličiny X a Y jsou nezávislé.

Rozdělení pravděpodobnosti

Rovnoměrné rozložení



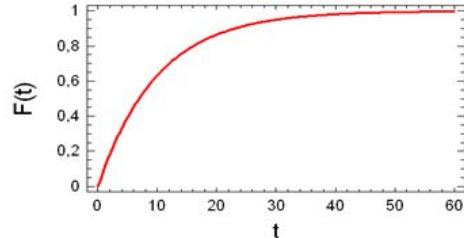
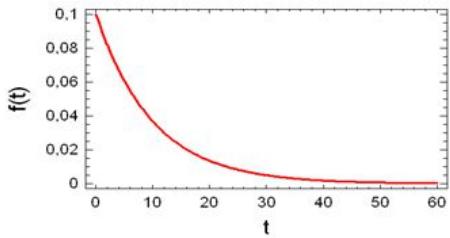
Normální rozdělení

Jeho důležitost ukazuje **centrální limitní věta** (CLV), jež zhruba řečeno tvrdí, že součet či aritmetický průměr **velkého počtu libovolných** vzájemně nezávislých a nepříliš „divokých“ náhodných veličin se vždy **podobá normálně rozdělené** náhodné veličině.

Exponenciální rozdělení

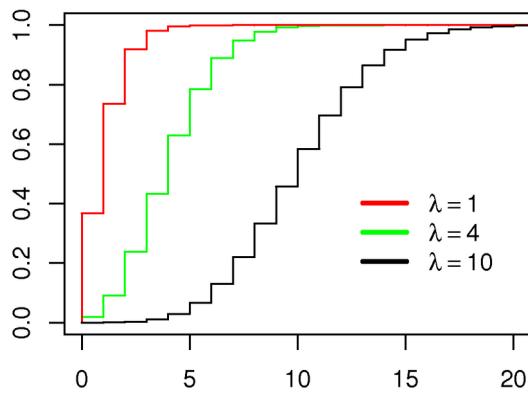
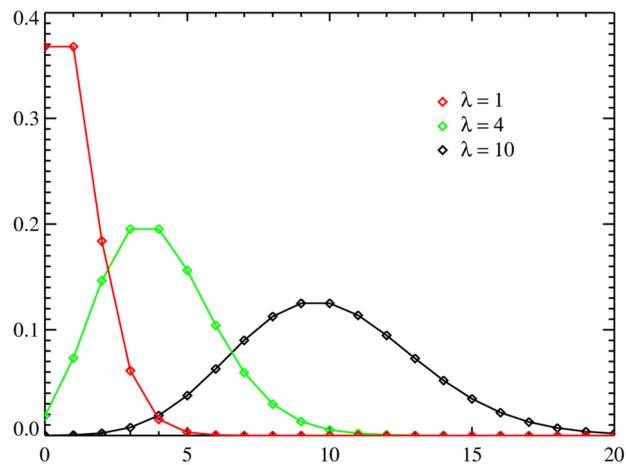
Určuje **dobu mezi dvěma následnými výskytu dané náhodné události**.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad F(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$



$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & ; x > 0, \\ 0 & ; x \leq 0. \end{cases} \quad F(x) = \begin{cases} 1 - e^{-\lambda x} & ; x > 0, \\ 0 & ; x \leq 0. \end{cases}$$

Poissonovo rozdělení



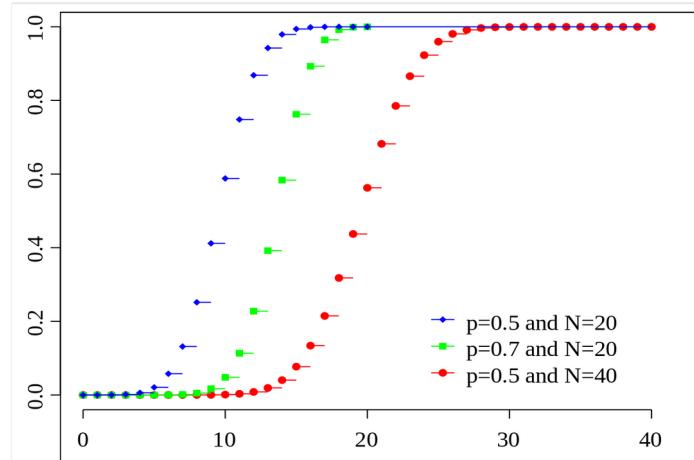
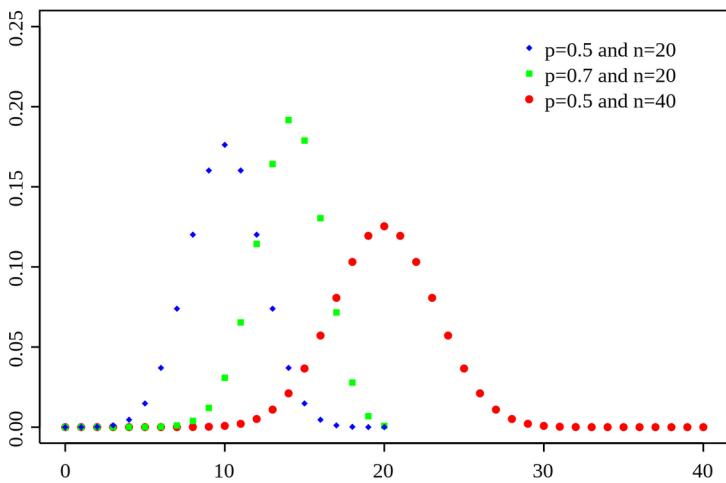
$$P(X = x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

$$F(x) = \sum_{x_i \leq x} \frac{\lambda^{x_i}}{x_i!} e^{-\lambda}$$

Jedná se o **diskrétní rozdělení**. Udává počet výskytů dané náhodné události za jednotku času.

Binomické rozdělení

Opět se jedná o **diskrétní rozdělení**.



$$P[X = x] = \binom{n}{x} p^x (1 - p)^{n-x}$$

Generování pseudonáhodných čísel

- **fyzikální zdroje náhodnosti:** využívají senzory zařízení (teplota, akcelerace, poloha, ...), vygenerovaná čísla jsou opravdu náhodná (**nedeterministické** generování). Problémem je rychlosť, generují jen málo bitů za sekundu.
- **algoritmické generátory:** pseudonáhodné (**deterministické**), generují řádově miliardy bitů za sekundu.

Kongruentní generátor

$$x_{n+1} = (ax_n + b) \bmod m$$

konstanty a , b , m musí být vhodně zvolené, jinak budou generovaná čísla nekvalitní (závislost mezi čísly).

- generují **rovnoměrné rozložení**,
- generují **konečnou posloupnost** čísel - perioda generátoru.

```
static uint32_t ix = SEED; // počáteční hodnota, 32b
double Random(void) {
    ix = ix * 69069u + 1u; // mod 2^32 je implicitní
    return ix / ((double)UINT32_MAX + 1.0);
}
```

Požadavky na generátor:

- **Rovnoměrnost rozložení**,
- Statistická **nezávislost generované posloupnosti**,
- Co **nejdelší perioda**,
- Rychlosť.

Další algoritmy generování:

- Mersenne twister (perioda $2^{19937} - 1$),
- Xorshift.

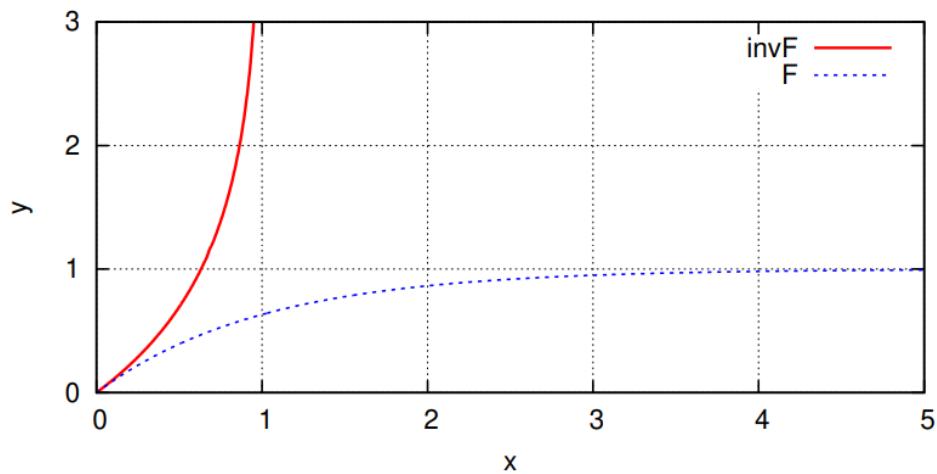
Transformace na jiná rozložení

Náhodné a pseudonáhodné generátory generují ideálně čísla dle **rovnoměrného rozložení**. Při výpočtech ale často potřebujeme generovat čísla dle jiného rozložení.

Metoda inverzní transformace

Ideální metoda pro generování rozložení, rozložení je určeno **analyticky pomocí inverzní distribuční funkce cílového rozložení**. Často ale distribuční funkci nemusíme znát nebo **nelze vyjádřit její inverzní funkce elementárními funkcemi**.

Příklad: Exponenciální rozložení $F(x) = 1 - e^{-\frac{x-x_0}{A}}$
 $y = x_0 - A * \ln(1 - x)$ viz. obrázek pro $x_0 = 0, A = 1$



Vylučovací metoda

Sérií pokusů hledáme číslo, které vyhovuje funkci hustoty cílového rozložení. Metoda je **nevzhodná pro neomezená rozložení** a rozložení, kde je **velká část hustoty pravděpodobnosti koncentrovaná do malého intervalu** (špičaté normální rozložení). Abychom byli schopni touto metodou generovat rozložení, musíme znát jeho ohraničení (osa x - **definiční obor** a osa y - **obor hodnot**). Funguje na principu, že se vygenerují 2 pseudonáhodná čísla x a y, x se dosadí **do funkce hustoty pravděpodobnosti f(x)** a její hodnota se **porovná s y**. Pokud je menší, je x vráceno jako **hodnota náhodné veličiny**, jinak se proces **opakuje**. Problém může být mnoho iterací při nesprávném rozložení.

Kompoziční metoda

Složitou funkci hustotu (případně distribuční) **rozdělíme** na několik jednodušších po **intervalech**. Na každém intervalu můžeme použít **jinou metodu** pro generování rozložení.

Bodové a intervalové odhady parametrů

- **Základní soubor** (populace) - obsahuje **všechny** vymezené jednotky.
- **Výběrový soubor** (výběr) - obsahuje pouze **některé** jednotky.

Vlastnosti výběrového souboru se snažíme **zobecnit pro celý základní** soubor.

Např. při volebním průzkumu se snažíme zobecnit **1000 dotázaných** voličů **na 8 milionu** voličů. Musíme použít **reprezentativní** výběr - **náhodný**.

Bodový odhad

Neznámý parametr (např. průměr) **základního souboru** odhadujeme pomocí **jediného čísla, bodu** na základě **výběrového souboru**. Bodovým odhadem parametru základního souboru jsou **popisné charakteristiky výběrového souboru**. pro odhady používáme **výběrový průměr** a **výběrový rozptyl**. Příklad:

- Pokud je **neznámým parametrem základního souboru** 10 000 nových baterek **průměrné napětí**, může jich náhodně vybrat **200 - výběrový soubor**. U těchto 200 baterek určíme **průměrné napětí**, např. 1.49 V a tuto hodnotu prohlásíme za odhad neznámého parametru (v tomto případě průměrného napětí) základního souboru. Takto bychom bodový odhad mohli opakovat např. i pro hmotnost.

Bodový odhad nebude témař nikdy **přesnou** hodnotou neznámého parametru. Za **lepší** považujeme ten, jehož **rozdělení je více koncentrované okolo neznámé hodnoty** parametru → má **menší rozptyl**, respektive **směrodatnou odchylku**.

- **NEstranný odhad**: Odhad je nestranný, pokud skutečnou hodnotu parametru **nepodhodnocuje ani nenadhodnocuje**. Nezaručí dobrý odhad, pouze **vyloučí systematickou chybu**. **Výběrový průměr a výběrový rozptyl** jsou nestranné odhady, pouhý **rozptyl není nestranný**.

Konzistentní (dobrý) odhad se **blíží** k skutečné hodnotě odhadovaného neznámého parametru, **čím větší počet** pozorování provádíme (čím větší je výběrový soubor).

Střední kvadratická chyba

Umožňuje měřit přesnost bodového odhadu, kombinuje **vychýlení a rozptyl**. Jsme poté schopni například říct, že odhadem je **1.49 V** se střední kvadratickou chybou **0.008V**.

Výběrový průměr

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^n X_i$$

Jedná se o **aritmetický průměr**.

Výběrový rozptyl

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2,$$

Intervalové odhady

Protože bodové odhady jsou poměrně nepřesné, používají se v praxi spíše intervalové odhady, u kterých můžeme říct, že s **určitou pravděpodobností** (vysokou **95-99%** - **interval spolehlivosti**) se zde **neznámý parametr základního souboru bude vyskytovat**.

- Spolehlivost odhadu je dána zvolenou **pravděpodobností** intervalu spolehlivosti, **čím** je pravděpodobnost **větší**, **tím** je daný odhad **spolehlivější**. **Čím** je ale **odhad spolehlivější**, **tím** se **zvětšuje** i příslušný **interval spolehlivosti**. Tj. **čím širší** interval spolehlivosti, **tím** je odhad **spolehlivější**, ale **méně přesný**, což je **nepraktické**. Mezi spolehlivostí a přesnosti je **nepřímá úměrnost**.

Pokud budu mnohokrát po sobě generovat náhodný výběr X_1, \dots, X_n a pro každý výběr sestojím 95% interval spolehlivosti pro parametr θ (resp. parametrickou funkci $\tau(\theta)$), pak přibližně 95 % z nich bude obsahovat skutečnou hodnotu parametru θ (resp. parametrickou funkci $\tau(\theta)$).

Např. pokud budu pro různé náhodné výběry 100x určovat 95% intervaly spolehlivosti pro parametr θ , pak v asi 5 případech dostanu interval, který neobsahuje skutečnou (správnou) hodnotu parametru θ . V některých případech nás může zajímat pouze **horní hranice - pravostranný interval spolehlivosti**, nebo jenom **dolní hranice - levostranný interval spolehlivosti** pro odhad parametru. Poté používáme **jednostranné intervaly spolehlivosti**.

Rozdělení používaná při intervalových odhadech:

- **Studentovo t** rozdělení,
- **Chí kvadrát** rozdělení,
- **Asymptotické normální** rozdělení.

Odhad relativní četnosti

Odhadovaná veličina je rozdělena **alternativně** - může **nabývat 2 stavů** (kluk - holka, rub - líc, ...). Na základě **výběrového souboru** poté **odhadujeme relativní četnost** výskytu (pravděpodobnost výskytu znaku) **v základním souboru** (např. kolik se za rok narodí holek a kolik kluků podle dat z 1 měsíce).

Testování hypotéz

Používá se v situacích, kdy **potřebujeme rozhodnout o správnosti nějakého tvrzení**. Např. jestli vede nová technologie výroby k zlepšení kvality vyráběných výrobků.

Statistická hypotéza (tvrzení)

Jedná se o **tvrzení o parametrech** rozdělení, z něhož pochází **náhodný výběr** (**střední hodnota**, **rozptyl**, **směrodatná odchylka**, ...) a o **typu** tohoto **rozdělení** (normální, exponenciální, rovnoměrné, ...). Dva druhy hypotéz:

- **Nulová hypotéza H₀ - předpoklad**, který vyslovíme o určitém parametru či tvaru rozdělení pravděpodobnosti sledované náhodné veličiny.
 - Pacient je nemocný
- **Alternativní hypotéza H₁** - popisuje, jaká situace nastává, **když nulová hypotéza neplatí**.
 - Pacient je zdravý

Testování statistických hypotéz je postup, kterým na základě hodnot náhodného výběru ověřujeme **platnost nulové hypotézy**. Testování může mít **dva závěry**:

- **H₀ zamítáme** a tím pádem **platí alternativní hypotéza H₁** (s určitou vysokou pravděpodobností).
- **H₀ nezamítáme**, to znamená, že H₀ buď platí, nebo nemáme dostatek informací k tomu, abychom mohli H₀ zamítнуть. Říkáme, že hypotéza **H₁ se neprokázala**.

Testovací statistika

Jedná se o **testovací kritérium**. Obor hodnot testovací statistiky rozdělíme na dvě části:

- **kritický obor W**: obor **zamítnutí** hypotézy, $T \in W \rightarrow \text{zamítáme } H_0$,
- **obor přijetí W'**: obor, ve kterém **nezamítáme** hypotézu. $T \notin W \rightarrow \text{nezamítáme } H_0$.

Platnost H₀ posuzujeme na základě **náhodného výběru** \Rightarrow **můžeme se dopustit chybných závěrů**. Chyby:

- **Chyba 1. druhu α** : zamítáme hypotézu, která platí. $\alpha = \text{hladina významnosti}$ testu (obvykle **0,05** \rightarrow zamítáme H₀ a s pravděpodobností aspoň **0,95** ($1-\alpha$) platí H₁).
- **Chyba 2. druhu β** : nezamítáme hypotézu, která neplatí. **$1 - \beta = síla\ testu$** - pravděpodobnost, že zamítneme hypotézu, která neplatí.

Chyby 1. a 2. druhu **jdou proti sobě** – nelze minimalizovat obě. (nepřímá úměrnost).

Testy

- **Studentův t-test**:
 - test střední hodnoty rozdělení se **známým rozptylem**,
 - test zda dvě normální rozdělení mající **stejný** (byť neznámý) **rozptyl**, z nichž pocházejí dva **nezávislé náhodné výběry**, mají **stejné střední hodnoty**.
- **F-test** je jakýkoliv statistický test, ve kterém má testová statistika rozdělení F.

Regresní a korelační analýza

- **regrese:** Hledáme funkční závislost náhodné **veličiny na jiné veličině** (jednostranná závislost). Umožňuje **předpovědi**, např. odhad výšky dcery na základě výšky matky.
- **korelace:** Hledáme **sílu** (těsnost) **závislosti dvou náhodných veličin** (vzájemná závislost). **Neslouží** k předpovědím. Např. určujeme, jak těsně spolu souvisí výška matky a výška dcery. Nebo investoři zkoumají, jak jsou nějaké akcie korelované a investují např. do těch které jsou korelované minimálně.

Regresní analýza

Umožňuje **vyjádřit vztah** mezi proměnnou, kterou chceme popisovat (**vysvětlovaná proměnná**), a množinou **vysvětlujících proměnných**.

- Hledáme vztah mezi množstvím zkonzumované zmrzliny (vysvětlovaná proměnná) a teplotou vzduchu (vysvětlující proměnná).
- Hledáme vztah mezi počtem bodů u zkoušky (vysvětlovaná proměnná) a počtem hodin, které student strávil přípravou na zkoušku (vysvětlující proměnná).

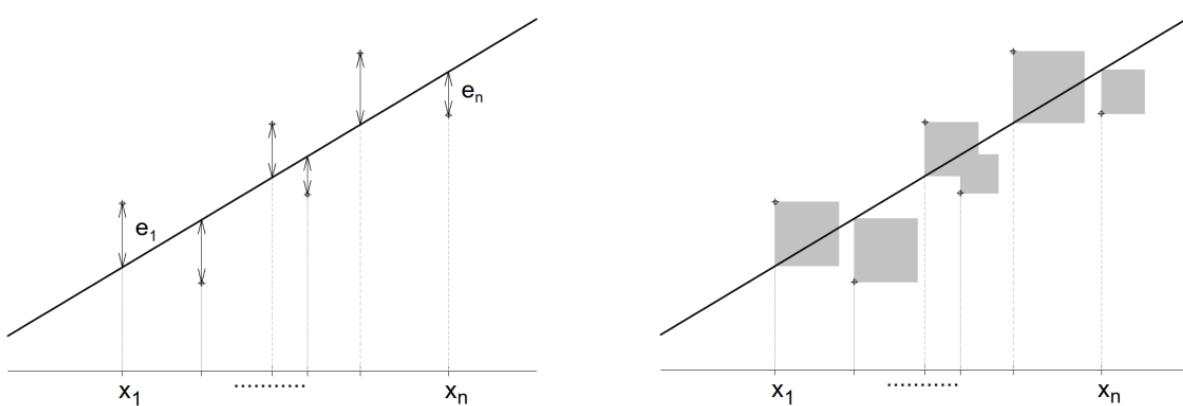
Při regresní analýze odhadujeme:

- **regresní přímku**,
- **regresní parabolu**,
- **regresní rovinu**.

Metoda nejmenších čtverců

Slouží pro odhad **regresní přímky**, **regresní paraboly** i **regresní roviny**.

$$Y_i = \beta_0 + \beta_1 x_i + e_i \quad \Rightarrow \quad e_i = Y_i - (\beta_0 + \beta_1 x_i)$$



Lineární regrese

Závislost mezi veličinami je popsána funkcí, která je **lineární v parametrech** (koeficientech).

Nelineární regrese

Závislost mezi veličinami je popsána funkcí **nelineární v parametrech**.

Korelační analýza

Úkolem je stanovit sílu závislosti mezi sledovanými kvantitativními znaky. Hledáme závislost mezi veličinami, které **spolu (logicky) mohou souviseť**. Např. přidávání olova do benzingu a jeho zvýšený výskyt v ovzduší a snižující se IQ a zvyšující se kriminalita. Naopak ale **korelace neimplikuje kauzalitu** - pokud existuje korelace mezi dvěma proměnnými, nelze z toho ještě vyvozovat, že jedna je příčinou a druhá důsledkem (Např. výdaje na podporu rozvoje vědy a počet sebevražd - obě v poslední době rostou, ale podpora vědy nezpůsobuje sebevraždy).

32. Hodnocení složitosti algoritmů (paměťová a časová složitost, asymptotická časová složitost, určování časové složitosti).

Složitost algoritmů definuje kritéria, podle kterých jednotlivé algoritmy hodnotíme. Nejčastějšími kritérii jsou **časové a paměťové nároky algoritmu**. Reálné doby běhu programů se liší na různých strojích a s různými vstupy. Proto zavádíme **časovou a prostorovou složitost algoritmů**, které umožňují:

- vyjádření vlastností algoritmu **nezávisle** na technických podrobnostech (počítači, na kterém běží)
- popis chování algoritmu pomocí **jednoduchých matematických funkcí**.

Časová složitost

Časová složitost je odvozena od počtu elementárních operací (sčítání, násobení, porovnání, ...). Udává **řádový počet provedených elementárních operací v závislosti na velikosti vstupu** (počet prvků řazeného pole, větší z čísel, u kterého hledáme společného dělitele). Příkladem mohou být 2 různé (nejedná se o stejnou funkcionality) algoritmy součtu:

Algoritmus Součet1:

```
1. for i ← (1, n) do
2.   for j ← (1, n) do
3.     s ← s + j
4.   end for
5. end for
6. return s
```

Počet elementárních kroků: $c_1 \cdot n^2 + c_2$
Zjednodušeně: n^2

Algoritmus Součet2:

```
1. while n >= 1 do
2.   s ← s + n
3.   n ← n / 2
4. end while
5. return s
```

Počet elementárních kroků: $c_3 \cdot \log_2 n + c_2$
Zjednodušeně: $\log_2 n$

- **Součet1:** U prvního algoritmu lze jednoduše **detektovat cyklus** podle **n** a **vnořený cyklus** podle **n** (obecně nemusí být jednoduché vnořené cykly detektovat). To znamená, že pro **každé n** se provede **n iterací**, celkově tedy $n \cdot n$ iterací, neboli n^2 iterací, a v každé iteraci je **provedeno sčítání**.
- **Součet2:** Zde je pouze **jeden cyklus** (while), tudíž hned na první pohled by se dalo **chybně** říct, že bude provedeno n operací. To ale není pravda, protože je nutné si v těle cyklu všimnout **dělení n** v každé iteraci, to znamená, že bude provedeno **$\log_2(n)$** operací.

Postup určení časové složitosti

1. Zhodnotíme, co je vstupem n a jak **měřit jeho velikost**.
2. Určíme maximální možný počet elementárních kroků algoritmu (elementárních operací jako je sčítání) provedených pro vstup o **velikosti n** .
3. Ve výsledné formuli **ponecháme pouze nejrychleji rostoucí člen**, ostatní zanedbáme.
4. Seškrťáme multiplikativní konstanty (**odebereme násobení konstantou**)

Průměrná časová složitost

Určujeme, pokud pro **různé vstupy stejné velikosti** vykoná algoritmus **různý počet kroků** (např. řazení seřazeného pole, náhodně uspořádaného pole a opačné seřazeného pole). Za časovou složitost poté považujeme **aritmetický průměr** časových nároků **přes všechny vstupy** (docela nereálné) dané velikosti. Proto u příkladu se řazením používáme jinou vlastnost, a to přirozenost algoritmu.

Asymptotická časová složitost

Jedná se o nejčastěji používané hodnotící kritérium, vychází z toho, že pro **malá n** se **nejrychleji rostoucí člen** nemusí výrazně projevovat. Popisujeme tedy chování algoritmu pro **n blížící se k nekonečnu**. Používají se tři různé asymptotické složitosti:

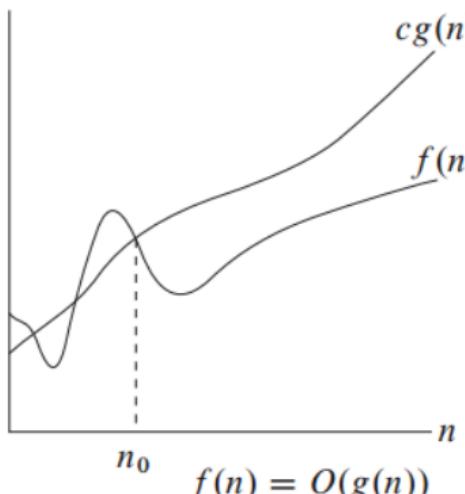
- O – Omikron (velké O , \mathcal{O} , big O) – horní hranice chování (**nejčastější**),
- Ω – Omega – dolní hranice chování,
- Θ – Théta – třída chování.

U definování složitostí se snažíme, aby co nejvíce odpovídaly složitosti algoritmu (tj. hledáme **nejmenší horní hranici** a **největší dolní hranici**). Není podstatná konkrétní přesná časová závislost, stačí **charakterizovat třídu**.

Složitost Omikron - **O**

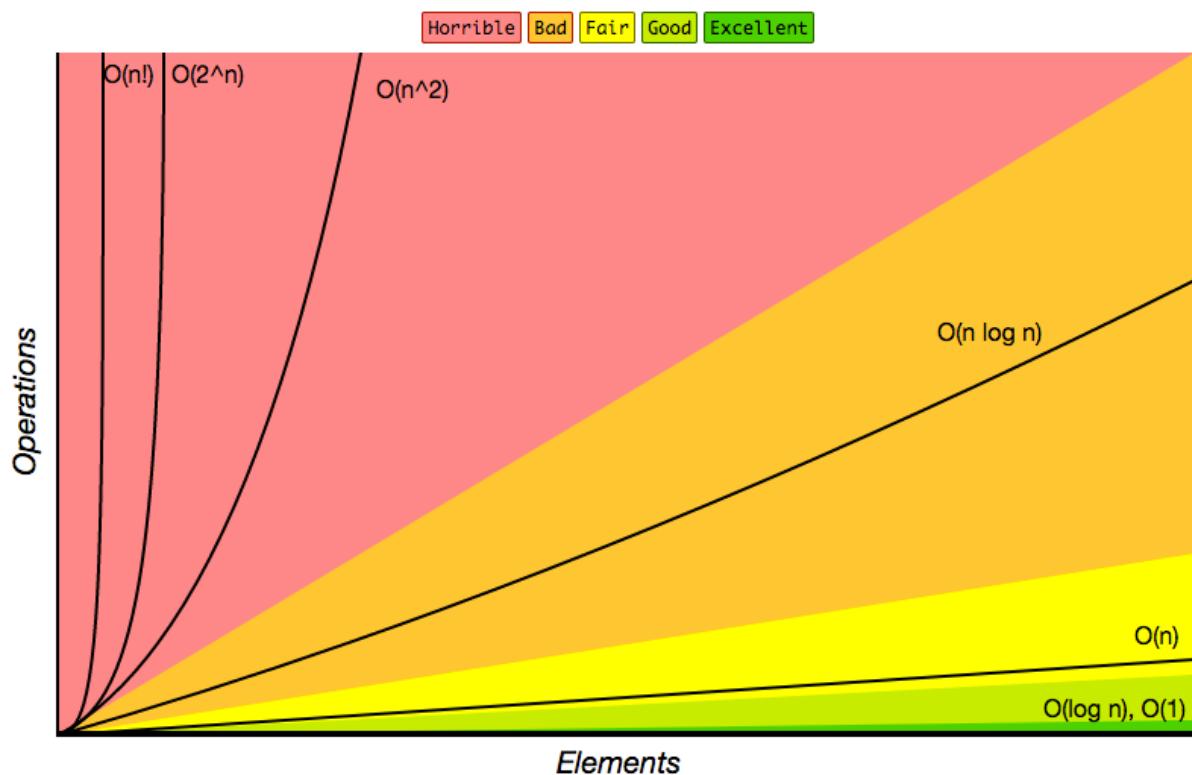
Vyjadřuje **horní hranici** časového chování algoritmu (omezuje chování algoritmu shora, tj. algoritmus **nebude nikdy pomalejší**, než dané omezení). Značíme jako **Omkron($g(n)$)** nebo ' **$O(g(n))$** nebo jen **$O(g(n))$** .

- Zápis $f(n) \in O(g(n))$, označuje, že funkce $f(n)$ je **rostoucí maximálně tak rychle** jako funkce $g(n)$. Dostatečně velký **násobek funkce $g(n)$ shora omezuje funkci $f(n)$** pro **dostatečně velké n** .



Nejčastěji používané složitosti:

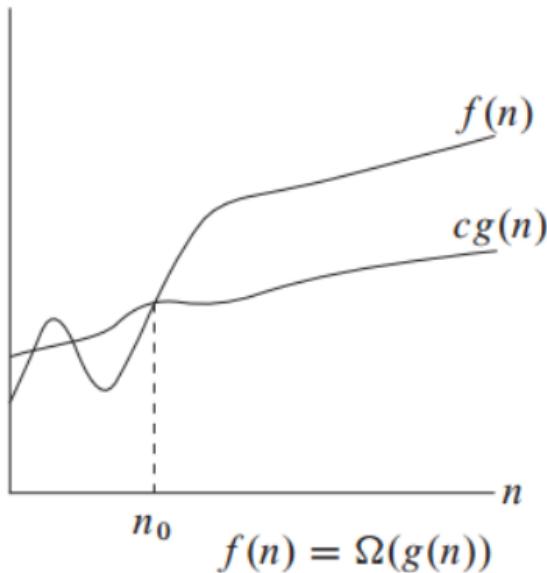
Asymptotická složitost	Vyjádření	Popis a nejčastější případ
konstantní	$O(1)$	Počet operací je pro libovolně velká vstupní data zhruba stejný. Typicky se jedná o nějaký jednoduchý matematický výpočet, jeden přístup k paměti, apod.
logaritmická	$O(\log n)$	Typicky ideální případ vyhledávání ve stromě s n prvky.
lineární	$O(n)$	Náročnost algoritmu se zvyšuje podobně jako velikost vstupních dat. Typicky jeden průchod polem. Některé algoritmy s touto složitostí mohou být implementovány i proudově.
lineárně logaritmická	$O(n \cdot \log(n))$	Typická složitost rozumného řadícího algoritmu.
kvadratická	$O(n^2)$	Náročnost algoritmu roste jako druhá mocnina velikosti vstupních dat. Typicky průchod všech dvojic v poli.
polynomiální	$O(n^k), k \in R$	
exponenciální	$O(k^n), k \in R$	
faktoriálová	$O(n!)$	Typicky vyhodnocování všech možných permutací n prvků, například v brute-force algoritmech.



Složitost Omega - Ω

Složitost Omega vyjadřuje **dolní hranici** časového chování algoritmu (omezuje chování algoritmu zdola, tj. algoritmus **nebude nikdy rychlejší**, než dané omezení). V praxi nemá moc využití. Značíme jako **Omega(g(n))**, $\Omega(g(n))$ nebo **jen** $\Omega(g(n))$.

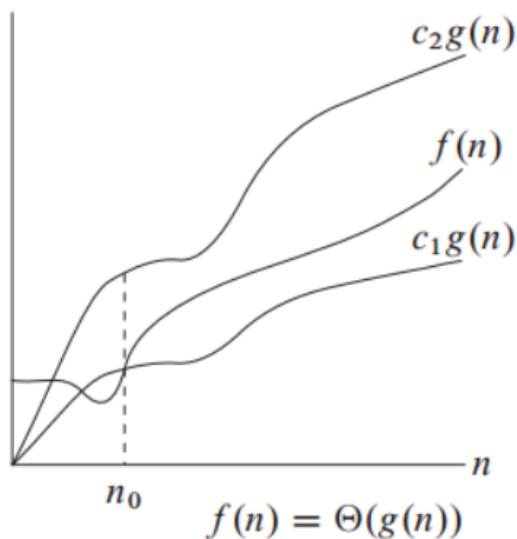
- Zápis $f(n) \in \Omega(g(n))$, označuje, že funkce $f(n)$ je **rostoucí minimálně tak rychle** jako funkce $g(n)$. Funkce $g(n)$ je tak **dolní hranicí množiny všech funkcí**, určených zápisem $\Omega(g(n))$.



Složitost Théta - Θ

Složitost **Theta** vyjadřuje třídu chování algoritmu – ohraničuje funkci $f(n)$ z obou stran (slouží jako **dolní i horní hranice**, tedy že funkce nebude **nikdy rychlejší** než $c1*g(n)$ a nebude **nikdy pomalejší** než $c2*g(n)$). Značíme jako $\Theta(g(n))$, nebo $\Theta(g(n))$.

- Zápis $f(n) \in \Theta(g(n))$ označuje, že funkce $f(n)$ roste **stejně tak rychle** jako funkce $g(n)$.



Třídy složitosti

- **P** - Pokud existuje Turingův stroj, který úlohu vyřeší v polynomiálním čase.

- **NP** - Pokud existuje nedeterministický turingův stroj, který rozhodne úlohu v polynomiálním čase

Orientační rychlosť výpočtu

Orientační typické hodnoty velikosti vstupu **n**, pro které algoritmus s danou časovou složitostí ještě většinou zvládne na běžném PC spočítat výsledek **ve zlomku sekundy nebo maximálně v řádu sekund**.

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1 000 000 – 100 000 000	100 000 – 1 000 000	1000 – 10 000	100 – 1000
$2^{O(n)}$	$O(n!)$		
20 – 30	10 – 15		

Paměťová (prostorová) složitost

Udává **spotřebu paměti**, případně **diskového prostoru** v závislosti na **vstupních datech**. Prostorovou složitost **nebývá kritická** a často ji neřešíme (paměť si narozdíl od času můžeme koupit). Nicméně může mít **záasadní** vliv na realizovatelnost algoritmu. Obvykle je u algoritmů ale **prostorová složitost menší**, než časová. Např. u většiny řadících algoritmů je prostorová složitost **konstantní O(1)**. U Quick sort je **logaritmická O(log(n))** - nutné si uchovat **hranice ještě nezpracovaných částí**. Stejně tak je nutné si uchovávat hranice polí u Merge sort - **rekurze znamená paměť**. U algoritmu pro prohledávání stavového prostoru **BFS** je např. prostorová složitost stejná jako časová, a to exponenciální **O(b^d)** (b je faktor větvení, d hloubka). U DFS je časová složitost stále exponenciální, ale prostorová pouze **O(b*d)**. U backtracking je pak složitost pouze **lineární O(d)**.

Asymptotická paměťová složitost

Obdobně jako u časové složitosti se používají **tři** různé asymptotické složitosti:

- \mathcal{O} – Omikron (velké O, 'O, big O) – horní hranice chování (**nejčastější**),
- Ω – Omega – dolní hranice chování,
- Θ – Théta – třída chování.

Mají i stejný význam.

Určování složitosti

Určování časové složitosti

Hlavní dva jevy, které řešíme při určování časové složitosti jsou **cykly** (zejména **vnořené**) a **rekurze** závisející na **vstupních datech**.

- **Vnoření cyklů** většinou znamená násobení složitosti → vede na polynomiální časovou složitost: dva vnořené cykly - $O(n^2)$, tři vnořené cykly - $O(n^3)$. Nutno sledovat úpravy řídící proměnné cyklu, např. její **dělení** vede na **logaritmickou složitost $O(\log(n))$** .
- **Rekurze** většinou znamená **exponenciální složitost**. Záleží, na počtu **rekurzivních volání funkce** v jejím těle. Pro dvě volání je složitost $O(2^n)$, pro tři volání $O(3^n)$, ... Počet volání můžeme označovat jako **faktor větvení** (zejména se používá u stromů).

Při více cyklech za sebou uvažujeme ten **nejhorší - nejrychleji rostoucí člen**.

Určování paměťové složitosti

Musíme identifikovat, jaké používáme proměnné

- **lokální statické proměnné**,
- **dynamické proměnné**,
- **REKURZE** - má **velký vliv** na paměťovou složitost programu, při každém rekurzivním volání funkce se **ukládají všechny její lokální proměnné**.

Globální statické proměnné můžeme ignorovat (považovat za **konstantní** paměťovou náročnost), lokální statické proměnné můžeme ignorovat jen v případě, že se funkce **nevola** rekurzivně. U **dynamických** proměnných **musíme sledovat alokaci** v cyklech **nebo při rekurzi** a jak **velká paměť** je alokována (pokud cyklem projdeme **n krát**, a **n krát** alokujeme **n prvků**, je **prostorová složitost kvadratická** (časová je lineární)).

Další zdroje:

- <https://docplayer.cz/108481191-Prostorova-pametova-slozitost-algoritmu.html>
- [Asymptotická složitost](#)
- [Třídy složitosti a Turingovy stroje](#)

33. Životní cyklus softwaru (charakteristika etap a základních modelů).

Životní cyklus softwaru

Mechanismus uplatňovaný při návrhu softwaru. Využívá se při vývoji velkých a složitých projektů. Je tvořena z **pěti etap**.

1. **Analýza a specifikace požadavků** (asi 8%),
2. **Architektonický a podrobný návrh** (asi 7%),
3. **Implementace a testování částí** (asi 12%),
4. **Integrace a testování** (asi 6%),
5. **Provoz a údržba** (asi 67%).

Bodům 1 a 2 je zejména nutné věnovat velkou pozornost, protože pozdější oprava chyb je **velmi drahá** (50x až 200x dražší, než její oprava při analýze a návrhu).

Při vývoji SW jsou spolupracují 3 strany:

- **Zákazník** - Objednal SW (může být i samotným uživatelem, většinou to tak ale není, SW objednává management, pracují s ním běžní zaměstnanci).
- **Dodavatel** - Vytváří SW (analytici, programátoři, testeři, management)
- **Uživatel** - Bude SW používat

Analýza a specifikace požadavků

Snažíme se **přesně specifikovat** co zákazník chce (ale neřešíme jak toho dosáhnout). Zákazník často neví, co přesně chce, nebo chce něco, co vlastně nepotřebuje. Úkolem analytiků je se zákazníkem požadavky probrat (**interview, dotazník, studium dokumentů, pozorování při práci, analýza existujícího SW**) a co nejvíce se přiblížit tomu, co opravdu potřebují. Výstupem je samotná **specifikace**, dále výstupem může být **analýza rizik** nebo **studie vhodnosti**, ale především **akceptační testy**, které vycházejí z jeho požadavků a které zákazník provede při převzetí. Výstupem také může být diagram případů užití, který slouží jako most mezi programátory a zákazníkem. Pokud jsou splněny, je SW v pořádku (**je verifikován**, otázkou je, jestli software po implementaci opravdu plní funkce, které zákazník potřeboval - validace. Následné změny jsou ale poté již nad rámec projektu a zákazník je musí zaplatit).

Kategorie požadavků:

- **Funkcionální** - co má projekt dělat.
- **Požadavky na provoz systému** - za jakých podmínek bude systém pracovat (např. bude jej současně obsluhovat 200 lidí).
- **Požadavky na výsledný systém** - podmínky pro vývoj a nasazení.
- **Požadavky na vývojový proces** - požadavky zákazníka na dodržování norem (např. použití nějakého standardizovaného postupu autentizace).

- **Požadavky na rozhraní** - komunikace se systémem (např. webová aplikace).
- **Externí požadavky** - požadavky dle charakteru aplikace (např. legislativní požadavky).

Architektonický a podrobný návrh

Architektonický návrh se spíše zaměřuje na to, co budeme dělat, podrobný návrh na to, jak to budeme dělat.

Architektonický návrh

Řeší se návrh aplikace od **fyzické úrovni** (např. že se bude jednat o třívrstvý informační systém s fyzicky odděleným DB serverem a klientem jako webový prohlížeč, i když část z toho může už být součástí specifikace - zákazník požaduje webovou aplikaci). Dále se může řešit jaké budeme používat nástroje pro vývoj - **programovací jazyky**. Následuje logické členění aplikace, např. výběr vhodného návrhového (architektonického) vzoru - **MVC**, **MVVM**, **MVP**. Následuje rozdělení celého projektu na podproblémy nejlépe nějak **logicky ucelené a rozhraní mezi nimi**. Můžeme je nazývat **moduly** (v případě MVC se může jednat o kontrolery). Příkladem může být modul pro správu uživatelů, ten bude skoro ve všech aplikacích. Členění do modulů může být výhodné také v tom, že na každém může pracovat jiný programátor - skupina programátorů. Nakonec by měly být produktem architektonického návrhu **integračních testů** a **testů celého systému** a **plán nasazení do provozu**.

Podrobný návrh

U informačního systému bude podrobný návrh spíše obnášet **návrh uložení dat v databázi** (např. pomocí ER diagramu) a **návrh uživatelského rozhraní**. U vědecké aplikace to bude spíše **návrh algoritmů a datových struktur**. Dále je součástí podrobného návrhu teké návrh způsobu **ošetřování neočekávaných a chybových stavů**. Výstupem by měl být také **podrobný odhad ceny a nároky na lidské zdroje**. Výstupem také bude návrh **testů jednotlivých modulů**.

Implementace a testování částí

Nejdříve se pouze o psaní kódu. V rámci implementace se také řeší **dokumentace** kódu, tvorba **uživatelské příručky/manuálu**. Chystáte se **testovací prostředí**, které bude simulovat prostředí zákazníka a na kterém se bude aplikace testovat.

Testovací prostředí je spojené s **tvorbou skriptů pro automatickou publikaci CI/CD**. **Testování jednotlivých modulů** a další...

Integrace a testování

Integrace je spojená se **začleňováním modulů** do jednoho celku - systému. Testování obnáší **integrační testy a testy systému jako celku**. Často se při provádění testů vracíme k předešlým dvoum bodům a **opravujeme vzniklé chyby**. Jakmile jsme přesvědčení, že SW odpovídá specifikaci, předáváme jej zákazníkovi k

akceptační testování. To opět může odhalit nějaké chyby, jakmile je ale akceptační testování provedeno a SW je tímto předán zákazníkovi, jsou následující změny již prováděny v rámci provozu a údržby.

Provoz a údržba

V prvé řadě se jedná o **opravu chyb**, které nebyly identifikovány ani akceptačním testováním. Dále jde o **aktualizaci použitých nástrojů** (např. z důvodu bezpečnostních rizik) a správu serveru, na kterém aplikace běží (někdy může provádět sám zákazník). Obecné řešení problémů při provozu, např. přesun na jiný server atd. Dále může jít o **změny** nebo o **přidání rozšiřujících funkcí** (postupně se přichází na to, že SW není validní - uživatelé doopravdy potřebuji něco jiného, nebo změny mohou být způsobené např. legislativou). Nakonec je potřeba pravidelně provádět **zálohy dat, kontrolovat stav úložiště, kontrolovat vytížení serveru, ...**

Pojmy

Přehled pojmu, které by se asi úplně nejsou součástí okruhu, ale mohou se hodit.

Důležité vlastnosti u vývoje SW

- Splnění požadavků
- Cena
- Čas

Typy SW produktu

- **Generický software** - SW pro širokou veřejnost, tzv. krabicový SW. Musí být velice dobře otestovaný (Ubuntu, Photoshop, Microsoft Office...).
- **Zákaznický software** - Vyuvíjen pro určitého zákazníka. Většinou pro danou specializovanou oblast neexistuje generický SW. Cena je mnohem větší (menší poptávka). Firma si ho přímo objedná a specifikuje požadavky (IS VUT, FIT KIT).

Vlastnosti software

- **Použití** - Správnost, spolehlivost, efektivnost, použitelnost, bezpečnost.
- **Přenos** - Přenositelnost, znovupoužitelnost, interoperabilita (spolupráce s jinými systémy).
- **Změny** - Udržovatelnost, modifikovatelnost, testovatelnost, dokumentovanost.

Problémy při vývoji software

- **Nevyhnutelné** - Složitost, přizpůsobivost (na změny zadání), nestálost, neviditelnost (nevíme přesně jak jsme blízko konci).
- **Ostatní** - Specifikace požadavků (nejasnosti v zadání), náchylnost k chybám (chyby se těžko odhalují), práce v týmu, nízká znovupoužitelnost, dokumentování, stárnutí softwaru (opravou chyb se zanesou nové chyby).



Modely životního cyklu SW

U každého projektu se etapy mohou lišit, společné rysy je však možné popsat modely životního cyklu SW. Definují etapy a jak po sobě jdou (ale ne však délku ani rozsah). Modely životního cyklu dělíme na **Heavyweight** a **Agilní**.

	Heavyweight	Agilní
přístup	prediktivní	adaptivní
velikost projektu	velká	malá
velikost týmu	velká	malá (kreativní)
styl řízení	centralizovaný příkaz-kontrola	decentralizovaný vedení-spolupráce
dokumentace	velký objem	malý objem
zdůraznění (důraz na)	process-oriented	people-oriented
fixní kritéria	<i>funkcionalita</i>	<i>čas a zdroje</i>
proměnná kritéria	<i>čas a zdroje</i>	<i>funkcionalita</i>

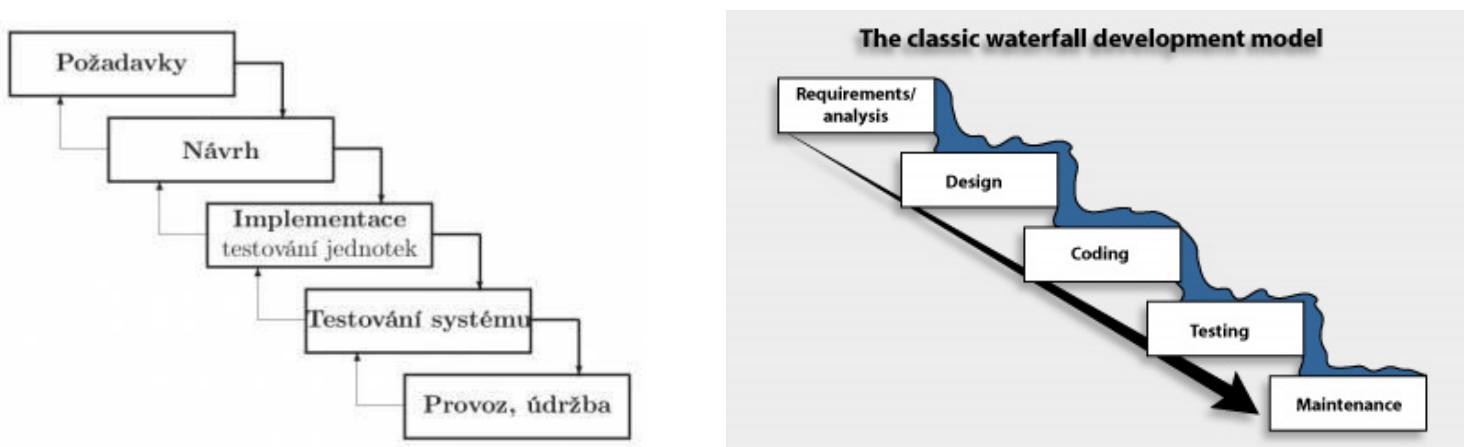
Heavyweight modely

Využívají se většinou u **velkých projektů** s **velkým počtem programátorů**, s **centralizovaným řízením** a **prediktivním přístupem** (velké množství detailního plánování pro dlouhý časový úsek). **Malá flexibilita**, špatná reakce na změny uprostřed vývoje - vyžaduje **stabilní požadavky**. Funkcionalita je daná potřebný čas a peníze je možné měnit. Malé nebo střední zapojení zákazníka při vývoji. Důraz je kladen na **procesy**, ty musí **běžet neustále** i při výměně zaměstnanců. Zaměstnanci jsou pouze zdroje dostupné v několika rolích (programátor, analytik, tester, ...) a lze je nahradit - **podstatná je role**. Předávání informací je založené především na dokumentaci (aby bylo možné obměňovat zaměstnance, u velkých projektů přes několik let to ale jinak nemusí jít) - **velká míra byrokracie**. Např. projekty NASA.

Vodopádový model

Etapy jsou řazeny za sebou **sekvenčně**, po skončení jedné začíná další. V případě nalezení chyb je potřeba se vrátit a projít znova. Neodpovídá reálnému vývoji, dnes už se jedná spíše o teoretický model.

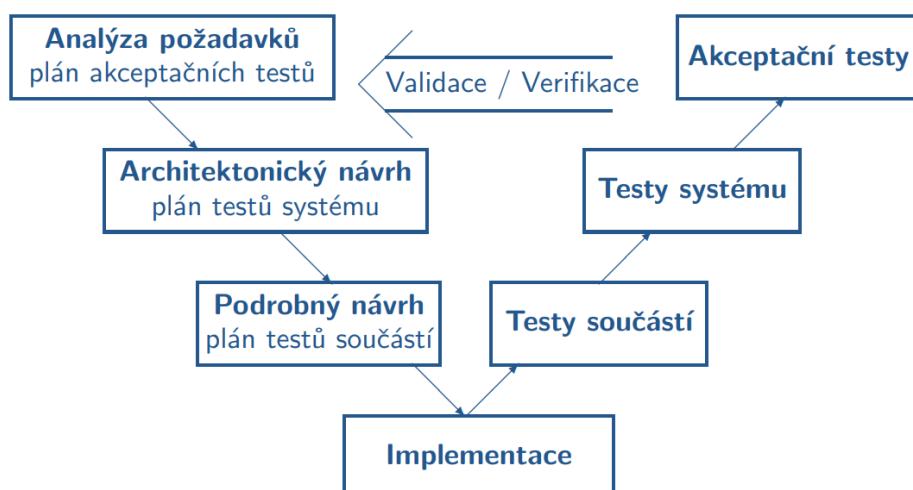
- **výhody:** jednoduchý na řízení - srozumitelný, dobrá výsledná struktura - vše se dělá na poprvé a většinou se nepředělává/nevylepšuje.
- **nevýhody:** k zákazníkovi se SW dostává až na úplném konci vývoje, **problém s validací** a detekováním chyb. Následné opravy mohou být velmi drahé.



V-model

Vychází z vodopádového modelu, má stejné základní vlastnosti a výhody (zachovává jednoduchost a srozumitelnost vodopádového modelu), snaží se řešit nevýhody, zejména **validaci**. Písmeno V symbolizuje grafické uspořádání etap, ale také zdůrazňuje **validaci** a **verifikaci**. Dělí se levou a pravou část

- **levá část:** vývojové aktivity a plánování testů,
- **pravá část:** testovací aktivity - provádění testů.



W-model

Vychází z V-modelu. Druhé V je spojeno s ověřováním jednotlivých aktivit (analýza, architektonický návrh, ...) a s testováním. Opět je model poměrně jednoduchý, ale k zákazníkovi se výsledný SW poprvé dostává až na konci vývoje, problém s validací přetrvává.

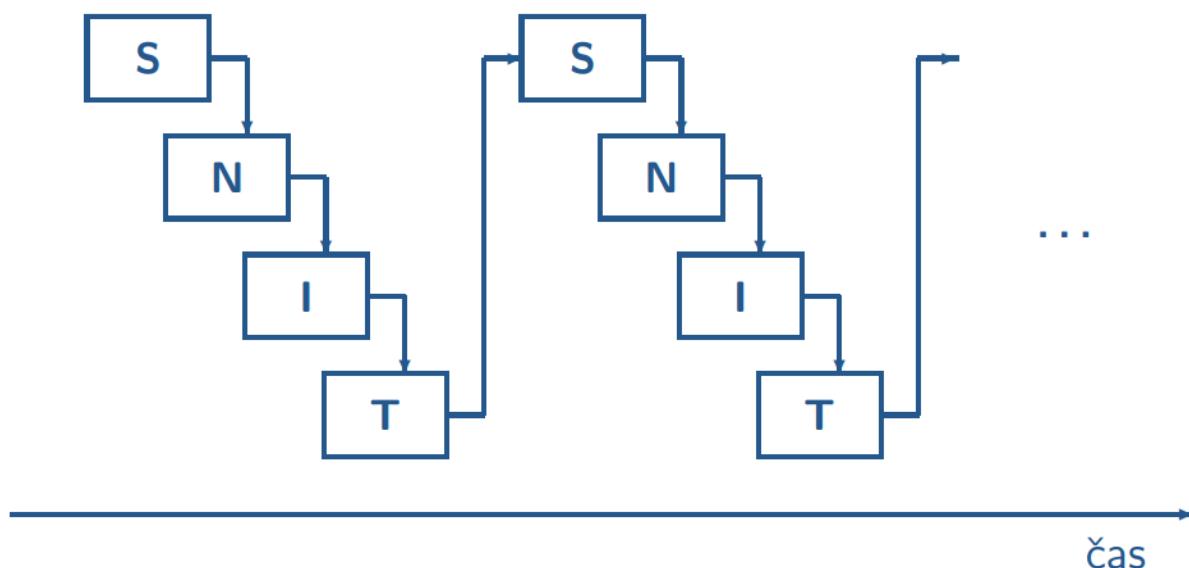
- **levá strana:**
 - **V1:** analýza, specifikace, architektonický návrh, podrobný návrh, implementace,
 - **V2:** ověřování výstupů etap z V1, plánování a návrh testů.
- **pravá strana:**
 - **V1:** provádění navržených testů,
 - **V2:** ladění, oprava chyb, změny v kódu, regresní testování.

Iterativní model

Proces je rozdělen do **iterací**, kde každá je **instance vodopádového modelu**.

Výsledkem každé iterace je reálný výsledek (rozšíření systému/aplikace o novou funkci, oprava chyb z předešlé iterace).

- **Výhody:** tvorba reálných výsledků v každé iteraci umožňuje zákazníkem výsledek validovat a lze rychleji odhalit chyby ve specifikaci, které mohou být napraveny v další iteraci.
- **Nevýhody:**
 - náročnější na řízení - vyžaduje účast zákazníka,
 - vede na hroší strukturu kódu, z důvodu postupného přidávání funkcí a vylepšení. Lze však eliminovat **refaktORIZACÍ** (změna kódu bez změny jeho funkce - otázka proč to ale platit, když to nic nedělá...)



Inkrementální model

Jedná se o kombinaci sekvenčních a iteračních metodik softwarového vývoje. Na základě specifikace se stanový **ucelené části systému**, např. **moduly**, které jsou zákazníkovi postupně předávány (možnost validace a oprav). Následně se pracuje jedním ze dvou způsobů:

- pro **každou** (malou) část systému je prováděna **série vodopádů** (iterační přístup) po jejím dokončení je **část systému předvedena zákazníkovi** a až poté se **začíná s další**.
- počáteční analýza, specifikace a návrh jsou provedeny **jedním vodopádem**. Vývoj (implementace) probíhá **iterativně** kombinovaný s **prototypováním**, v každé iteraci je systém rozšířen (vytvořen nový prototyp) a ten je prezentován zákazníkovi, nakonec je poslední prototyp považován za konečný systém.

Zhodnocení:

- **Výhody**: Inkrementální způsob umožňuje postupnou validaci a omezuje projektová rizika.
- **Nevýhody**: Vyžaduje zvýšenou pozornost při návrhu a implementaci rozhraní mezi moduly. Vývoj po částech může vést ke ztrátě vnímání logiky celého celku.

Spirálový model

Zavádí do vývoje prototypování a klade **důraz na analýzu rizik**. Jednotlivé etapy se opakují vždy na **vyšším stupni zvládnuté problematiky** (analýza, specifikace, arch. návrh, ...), výsledkem je **prototyp**. Spirála je rozdělena do 4 kvadrantů:

1. **Stanovení cílů**: určení **funkcionálních a výkonnostních požadavků**, určení omezujících podmínek (čas a cena), návrh možných alternativ.
2. **Vyhodnocení** stanovených cílů: ověření jejich splnitelnosti, **analýza rizik, prototypování a simulace**,
3. **Realizace** stanovených cílů a jejich testování,
4. **Plánování** následujícího cyklu.

Význam jednotlivých cyklů (jejich počet není pevně stanovený), po každém cyklu je dokončený **Milestone**.

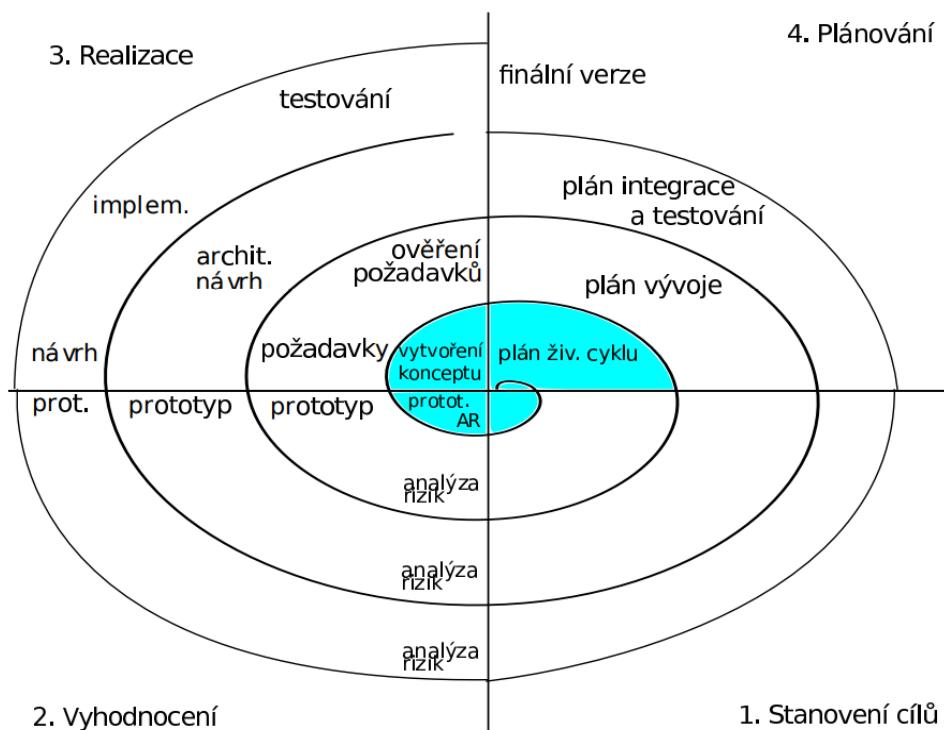
1. globální rizika, základní koncept vývoje, volba metod a nástrojů,
2. tvorba a ověřování specifikace požadavků,
 - a. vyhodnocení záměrů a cílů projektu, rozhodnutí o dalším pokračování. Jsou identifikovány všechny požadavky a jejich chápání je shodné mezi zákazníkem a dodavatelem.
 - b. Je vytyčena cena, plán priority aj.
 - c. Jsou identifikovány rizika a postupy pro jejich zmírnění.
3. vytvoření a ověření návrhu,
 - a. Vyhodnocení výběru architektury, požadavky a architektura jsou stabilní.
4. implementace, testování a integrace (vodopádový přístup)
 - a. Je vytvořena stabilní verze schopná testování u zákazníka, pokud

nejsou odhaleny problémy, je systém je připravený na distribuci pro uživatele.

- b. Porovnání plánovaných a skutečných výdajů.

Zhodnocení:

- Výhody:
 - Vhodný pro složité a velké projekty,
 - chyby a nevhovující postupy jsou odhaleny dříve pomocí analýzy rizik.
- Nevýhody:
 - Analýza rizik musí být na vysoké úrovni, jinak postup může selhat
 - výsledný SW je k dispozici až po posledním cyklu (problém s validací)



Analýza rizik

Zjištění možných ohrožení průběhu projektu a připravení reakce na tato ohrožení, lze včas vyloučit nevhodná řešení.

- **projektová:** odchod lidí, snížení rozpočtu,
- **technická:** neznámé technologie, selhání HW,
- **obchodní:** špatný odhad zájmů,

Rational Unified Process (RUP)

Nejedná se o model životního cyklu SW ani o konkrétní metodiku vývoje SW, jde o **rozšiřitelný framework**, který je možné **uzpůsobit organizaci a příslušnému projektu**. Jedná se o **komerční produkt** (firma Rational Software), který je dodávaný společně s patřičnými nástroji. Je založený na osvědčených praktikách:

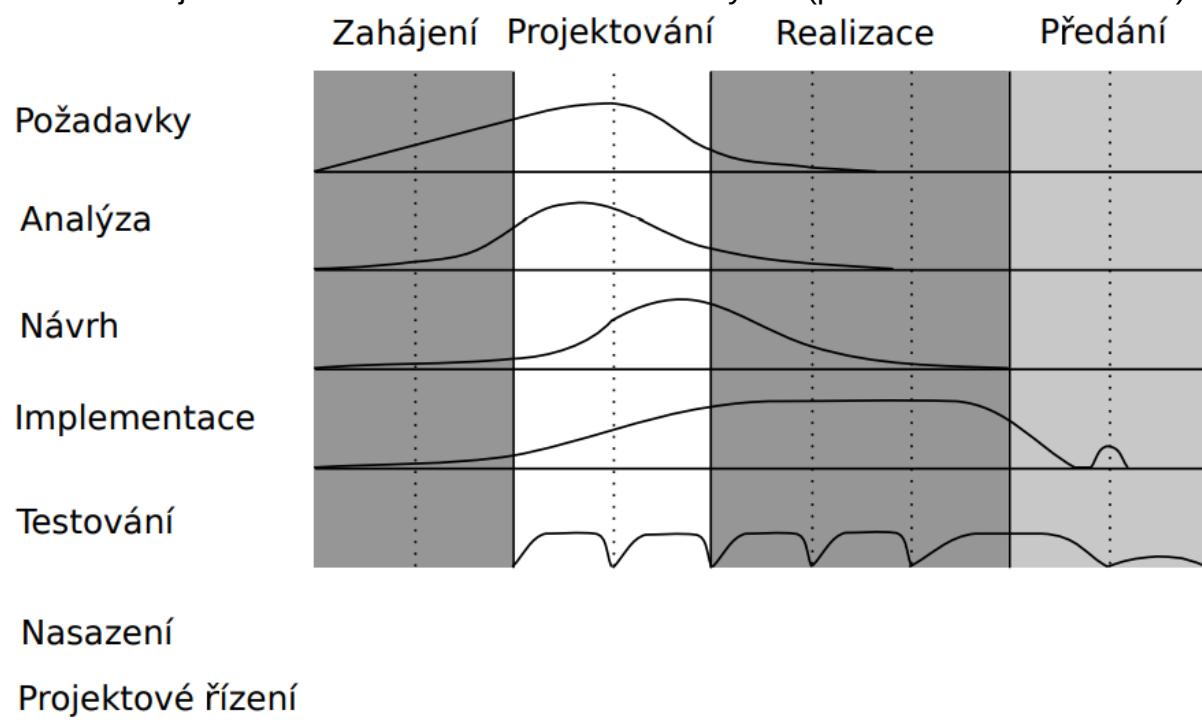
- **iterativní vývoj**, délka iterace je 2 až 6 týdnů,

- **vizualizovaný návrh systému** - využití UML a jiných nástrojů,
- **průběžná kontrola kvality**,
- **řízení změn**,
- **využití existujících komponent**.

Vývoj je rozdělen do několika cyklů. První nejdůležitější je **Initial Development Cycle** jehož výsledkem je funkční SW. Následný další vývoj (vylepšení, opravy) probíhá formou **Evolution Cycles**. Základní cyklus (Initial Development Cycle) je rozdělen na 4 fáze:

1. **Zahájení** (asi 10%) 1 až 2 iterace,
2. **projektování** (asi 30%) 2 až 4 iterace,
3. **realizace** (asi 50%) 2 až 4 iterace,
4. **předání** (asi 10%) aspoň 2 iterace (beta verze, plná verze).

Každá fáze je rozdělena na iterace o délce 2 až 6 týdnů (počet iterací se může lišit).



Ze spirálového modelu přebírá mezníky/milníky - **Milestones**.

Zhodnocení:

- **Výhody:**
 - robustní, lze upravit pro potřeby projektu,
 - iterativní přístup, včasné odhalení rizik,
 - grafický návrh - vazba na UML,
 - detailní propracovanost
- **Nevýhody:**
 - u malých projektů může představovat příliš velkou zátěž - neefektivní vývoj.
 - komerční produkt

Agilní metodiky

Využívají se většinou u **malých projektů s malým počtem programátorů**, s **decentralizovaným řízením** (vedení založené na spolupráci) a **adaptivním přístupem** (menší množství plánování, úpravy dle reakcí zákazníka) - **velká flexibilita**. Čas a peníze jsou obvykle dané a funkcionalita se implementuje na základě toho (co se stihne, co bude zaplaceno). Vyžadují **intenzivní zapojení zákazníka** do procesu vývoje. Agilní metodiky kladou důraz na lidi a jejich **individualitu** (people oriented), člověk je v dané oblasti profesionál (analytik, programátor, tester) a je schopen rozhodovat technické otázky práce, velmi důležitá je **komunikace v rámci týmu**. Omezuje se **byrokracie** a formální požadavky. Ověření správnosti je prováděno **zpětnou vazbou** od zákazníka. Odchod schopných zaměstnanců může být kritický. Agilní metodiky dobře reagují na změny v průběhu vývoje, což je běžné.

Rapid Application Development

Metodika založená na **rychlém iterativním vývoji prototypů**. Brzská dispozice funkčních verzí, vyžaduje **intenzivní zapojení zákazníka** do vývojového procesu - **zpětná vazba** na poskytnuté verze. Zaměřuje se zejména na **splnění potřeb a požadavků zákazníka** (business potřeb SW). Metodika je určena pro menší a středně velké týmy. Zhodnocení:

- **Výhody:** flexibilita a schopnost rychlé reakce na změny požadavků od zákazníka. Vyšší kvalita zpracování business potřeb (základní dostává doopravdy to, co chce). Více projektů splňuje termíny a ceny.
- **Nevýhody:** nižší kvalita návrhu a výsledného kódu způsobená změnami, což vede na problémy s udržovatelností.

Extrémní programování (XP)

Populární agilní metoda, která je založena na **komunikaci** (v týmu a se zákazníkem, zákazník se prakticky stává členem týmu) a **iterativním vývoji**. SW není dodán zákazníkovi celý v určitý termín v budoucnosti, ale je dodáván způsobem, který odráží aktuální potřebu uživatelů - **do systému vložíme to, co potřebujeme, když to potřebujeme**. Dva významní členové týmu:

- **kouč:** zajišťuje komunikaci v týmu, pomáhá programátorům s technickými dovednostmi, komunikuje s velkým šéfem a vyšším managementem.
- **velký šéf:** provádí zásadní rozhodnutí, komunikuje s vyšším managementem.

Extrémní programování klade velký důraz na **spolupráci v týmu**, manažeři, vývojáři a zákazník jsou si rovní. Je založeno na 5 základních principů:

1. **komunikace:** programátoři, zákazníci a manažeři spolu musí komunikovat - využití technik, které komunikaci vyžadují (pair programming, code review, ...),
2. **jednoduchost:** jednoduché věci se realizují a upravují s menším počtem chyb (v případě potřeby je lze rozšířit). Přírůstkové malé změny, uvolňování malých verzí systému (nejpodstatnější požadavky, které jsou postupně

vylepšované a doplňované)

3. **zpětná vazba a testování:** vše musí být otetováno, ke každé funkci píšeme testy, klidně i před tím, než začneme programovat (Test Driven Development). Jednotkové i integrační testy, zpětná vazba zákazníka.
4. **odvaha:** nebát se zahodit naprogramovaný kód, nebát se zkoušit neznámé. Pokud je potřeba, nebát se provést zásadní změny.
5. **respekt:** Každý malý úspěch prohlubuje respekt k jedinečným příspěvkům každého člena týmu.

Zhodnocení:

- **Výhody:**
 - iterativní inkrementální proces,
 - ladění na základě zpětné vazby - zapojení uživatelů,
 - založený na testování,
 - průběžná integrace
- **Nevýhody:**
 - vyžaduje dodržování základních principů (neustálé psaní testů, párové programování, atd)
 - nedefinuje přesný postup.

Collective-Code-Ownership

Každý člen týmu má **možnost a povinnost ovlivňovat kód** (přidání nových funkcí, odstranění chyb, refaktorizace), což snižuje riziko, že výpadek člena týmu výrazně zbrzdí práci a podporuje pocit odpovědnosti vývojáře za kvalitu kódu (kdokoliv to po něm bude čist). Vyžaduje **jednotný styl programování**.

Průběžná integrace

Automatizované sestavování, automatizované spouštění napsaných testů, automatizovaná publikace na testovací prostředí testování.

Scrum

Agilní metoda odvozená ze hry rugby, lze kombinovat s XP. Dělí se na tři základní fáze:

- **pre-game:** plánování a **architektonický návrh**, využívá se Product Backlog.
- **game:** vývoj, který probíhá v iteracích, iterace se nazývá **Sprint** a tvá okolo **30 dní**, výsledkem každé iterace je funkční inkrement. Na začátku každého sprintu je setkání mezi vývojáři, zákazníkem, uživateli atd. a definují se cíle sprintu - **Sprint Backlog** (seznam úloh nutných pro dosažení cíle). Každý den v průběhu sprintu se provádí **Scrum Meeting** - 15 min setkání členů vývojového týmu. **Scrum Master** je vedoucí týmu. Na konci sprintu se provádí **Sprint Review** - vyhodnocení proběhlého sprintu.
- **post-game:** integrace výsledků jednotlivých sprintů, testování (integrační, celého systému, akceptační), dokumentace, školení uživatelů.

Zhodnocení:

- **Výhody:**
 - iterativní inkrementální proces,
 - časté uvolňování verzí,
 - architektura je navržena před procesem vývoje,
 - jednoduchý proces,
 - zapojení uživatelů.
- **Nevýhody:**
 - nedefinuje přesný postup,
 - integrace až po vytvoření všech inkrementů.

Crystal

Jedná se o rodinu metodologií, základní přístupy a techniky sdílí s **XP**. Více lidí je ale schopno tento proces akceptovat. Rozděluje projekty do **kategorií dle kritičnosti a důležitosti**:

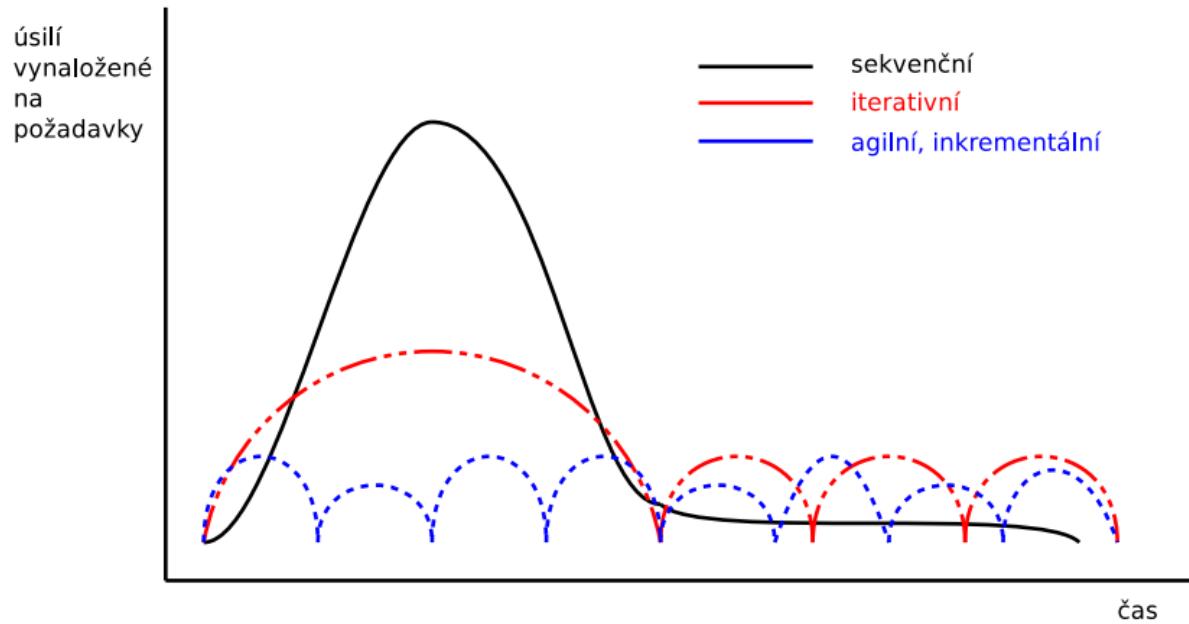
- Comfort **C**,
- Discretionary Money (volné uvážení) **D**,
- Essential Money **E**,
- Life **L**.

kategorie metodik, které se používají se označují barvou, liší se dle velikosti týmu a délce projektu



Zhodnocení:

- **Výhody:**
 - iterativní inkrementální proces - časté uvolňování verzí,
 - průběžná integrace,
 - zapojení uživatelů - požadavky se ladí během celého vývoje na základě zpětné vazby
- **Nevýhody:**
 - nedefinuje jasný společný proces,
 - není vhodný pro vysoce kritické projekty,
 - velká závislost na přímé komunikaci.



34. Jazyk UML

Unified Modeling Language je v softwarovém inženýrství **standardizovaný grafický jazyk** pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. UML nabízí standardní způsob zápisu a podporuje **objektově orientovaný přístup k analýze**. Základním cílem je **vizualizace, specifikace struktury a chování navrhovaného systému**. Napomáhá k dekompozici systému. Umožňuje modelovat **business procesy**, konkrétní **příkazy programovacího jazyka**, vztahy mezi třídami v OOP. UML diagramy dělíme do dvou skupin (respektive do 3, **ERD do UML NEPATŘÍ**):

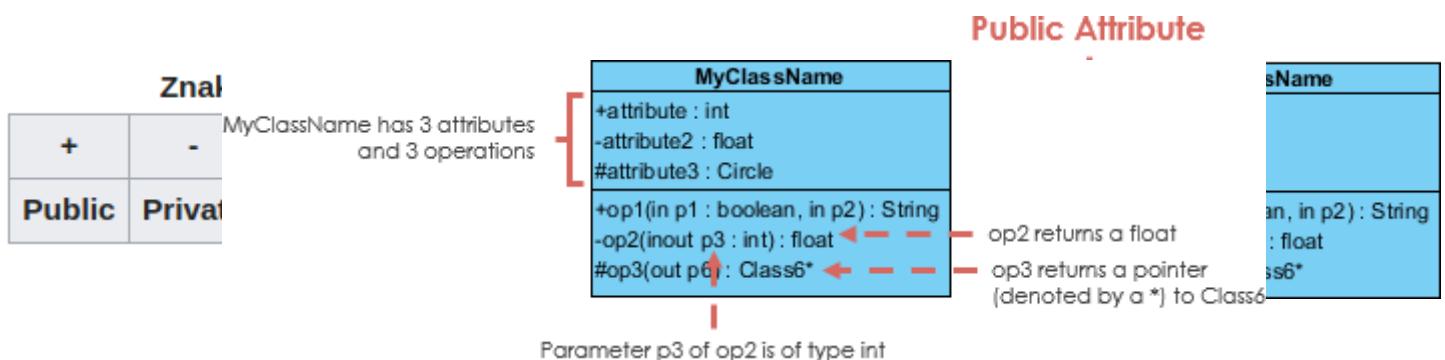
- **Structure diagrams** (Diagramy struktury)
- **Behaviour Diagram** (Diagramy chování)
- **Interaction diagrams** (Diagramy interakce)

Diagramy struktury (Structure diagrams)

Popisují strukturu systému, tedy **z čeho je systém složený**. Představují **statické aspekty** systému. Zdůrazní části/komponenty, které musí být přítomny v modelovaném systému. Jsou široce používány při **dokumentaci architektury softwarových systémů**. **Diagram komponent** například popisuje, jak je softwarový systém rozdělen na **komponenty** a ukazuje **závislosti** mezi těmito komponentami.

Diagram tříd

Zobrazuje **třídy a statické vztahy** mezi nimi. Popisuje **statickou strukturu systému** a znázorňuje datové struktury. Každá třída je tvořena **jménem, seznamem atributů a seznamem operací**. Atributy a operace (metody, funkce) mají stanovenou **viditelnost**. Atributy mají specifikovaný **typ** a operace mají **vstupní parametry a výstupní typy**. Diagramem tříd se je popisován **doménový model**.



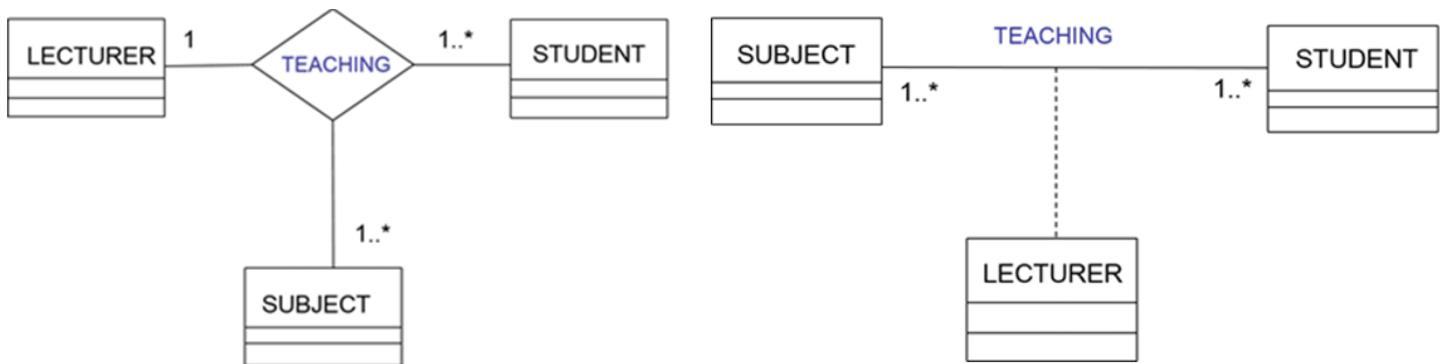
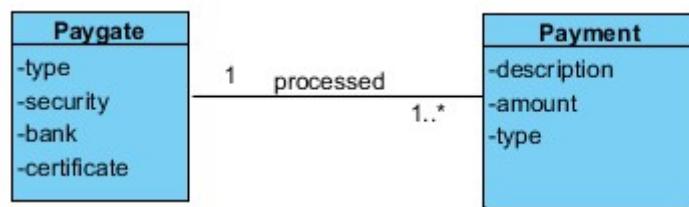
Mohutnost (Multiplicity)

Mohutnost vztahu udává, kolik instancí jedné třídy může být svázáno s instancí třídy druhé

0	0..1	1	0..*	1..*
Žádná instance (zcela výjimečně)	Žádná nebo právě jedna	Právě jedna instance	Žádná nebo více instancí	Jedna nebo více instancí

Asociace

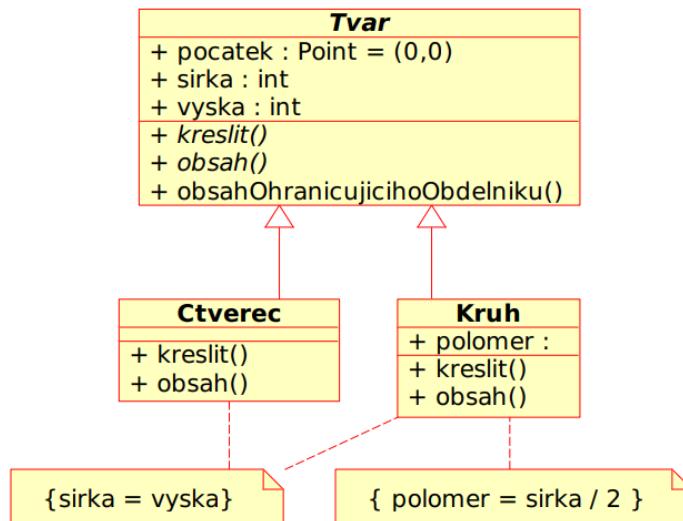
Asociace určuje **základní vztah** mezi dvěma entitami. Ty mohou existovat nezávisle na sobě. Zakreslujeme ji jednoduchou plnou čáru. Mohou mít název (sloveso, podstatné jméno). Asociace mohou být **binární** (unární je také brána jako binární ale **reflexivní**), **ternární** a **n-ární**. Jiné než binární asociace není moc doporučené používat (nepřehledné, obvykle lze převést na binární nebo na asociační třídu).



Převod ternární asociace na binární s asociační třídou:

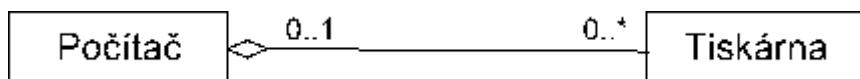
Dědičnost (Generalization)

Vyjadřuje vztah generalizace/specializace mezi třídami. Odvozená třída **sdílí atributy** (dle viditelnosti) **chování** (operace), **vztahy** a **omezení** obecnější třídy. Odvozená třída může navíc přidávat a **modifikovat** atributy a chování (operace).



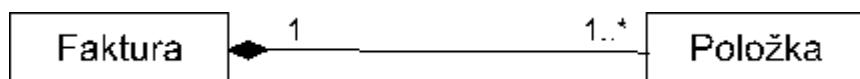
Agregace (Aggregation)

Agregace představuje **volnou vazbu** mezi celkem a součástí, kdy jeden objekt (**celek**) využívá služby dalších objektů (**součástí**). Například vztah mezi počítačem a tiskárnu je vztah typu **agregace**, kdy počítač s tiskárnou tvoří jeden celek, ale **tiskárna může existovat i bez počítače**. Dokonce může být **součást** (tiskárna) využita i jiným **celkem** (např. mobil) - může být součástí více kolekcí. Agregace je **formou asociace** a v grafické podobě se odlišuje **prázdným kosočtvercem** na straně celku.



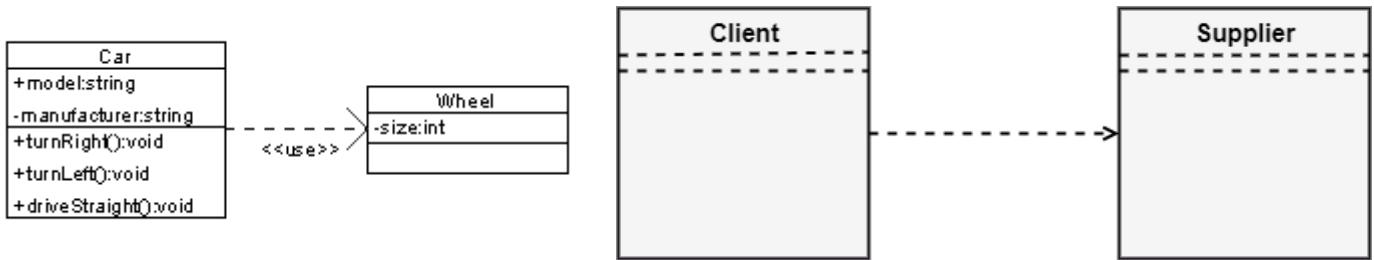
Kompozice (Composition)

Kompozice je podobná agregaci, avšak reprezentuje **silnější vztah**. Entita části (součást) **nemá bez celku smysl**, nemůže bez celku existovat. Pokud zanikne celek, zanikají automaticky i **jeho části**. Příkladem kompozice je vztah mezi fakturou (**celek**) a jejími položkami (**součásti**). Každá položka **musí být součástí právě jedné faktury**, faktura má jednu a více položek. Jestliže fakturu zahodíme, nezbudou nám po ní ani žádné položky. **Kompozice je formou asociace** a v grafické podobě se odlišuje **plným kosočtvercem** na straně celku.



Závislost (Dependency)

Závislost je slabší forma vztahu, která naznačuje, že **jedna třída závisí na jiné**.



Závislost bývá doplněna o **stereotyp**, který ji blíže specifikuje. Nejčastěji se používá stereotyp **<<use>>**. Pokud není uveden, jedná se automaticky o stereotyp **<<use>>**.

Další stereotypy na příkladu klient/dodavatel:

- **<<initiate>>** / **<<create>>**: klient vytváří instance dodavatele,
- **<<send>>**: operace klienta zasílá signál příjemci (dodavateli),
- **<<call>>**: klientská třída volá operaci dodavatele,
- **<<trace>>**: klient realizuje dodavatele,
- **<<refine>>**: klientská třída poskytuje detailnější informace než dodavatel.

Realizace

Udává vztah mezi **třídou** a **rozhraním** (interface). Označuje fakt, že třída **implementuje všechny operace z daného rozhraní** (respektive implementuje rozhraní). Objekt používající rozhraní pak umí používat i tuto (implementační) třídu.

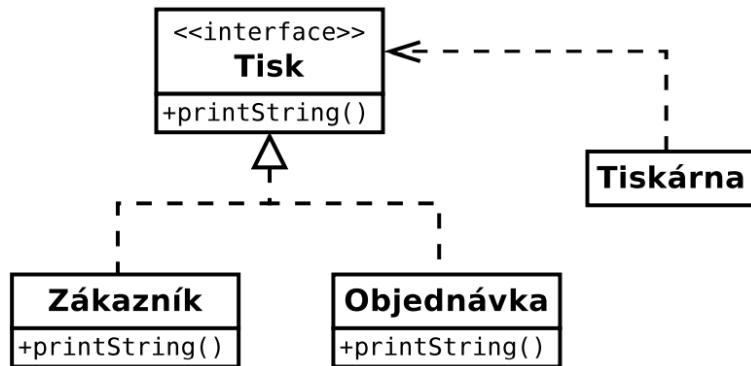
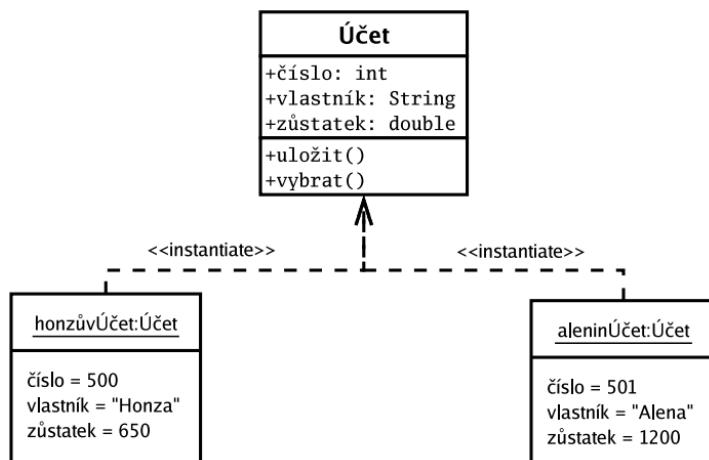


Diagram objektů (Object Diagram)

Je úzce svázán s diagramem tříd, **znázorňuje objekty a jejich relace** (vztahy) v určitém čase. Relace jsou **dynamické** (nemusí trvat po celou dobu existence objektů).



Při vytváření objektů dochází k vytváření spojení mezi nimi, spojení jsou **instance asociací**.

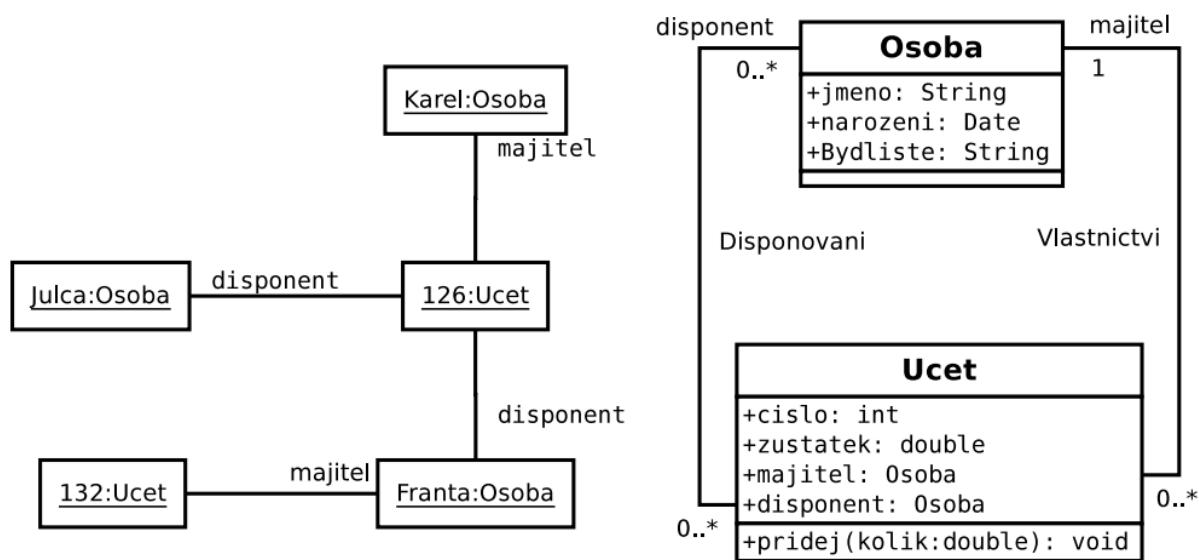


Diagram komponent

Diagram komponent znázorňuje **komponenty** použité v systému, tím mohou být **logické komponenty** (např. business komponenty, procesní komponenty) nebo také **fyzické komponenty**. Popisuje, jak je softwarový systém rozdělen na komponenty a ukazuje **závislosti mezi těmito komponentami**.

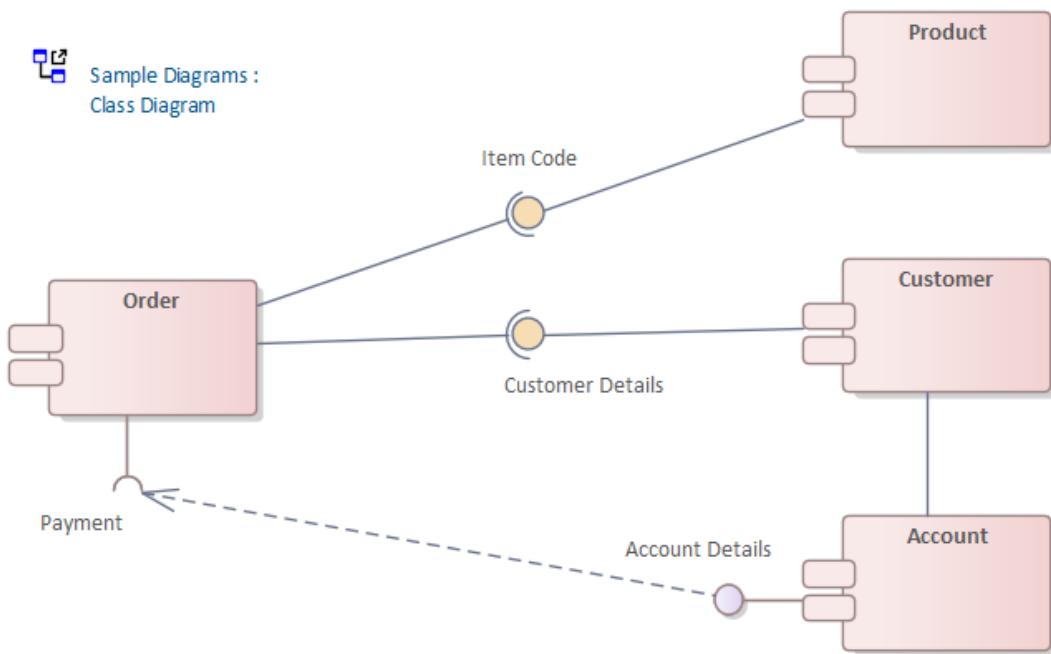
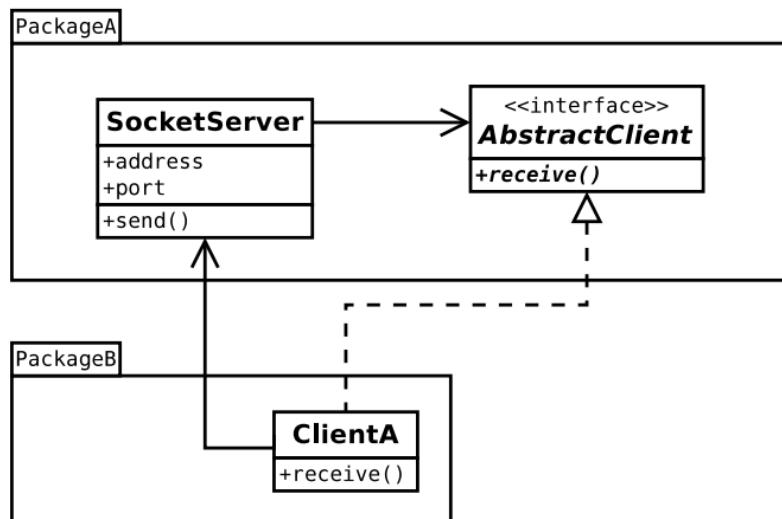


Diagram seskupení

Umožňuje **seskupit sémanticky související elementy**, umožňuje **zapouzdřit prostor jmen**. Definuje sémantické hranice modelu a umožňuje souběžnou práci v

etapě návrhu. Např. třídy a rozhraní řešící komunikaci dáme do jednoho balíčku a třídy obsluhující klienta do druhého.

Diagramy chování (Behaviour Diagram)



Popisují **chování systému**, tedy **jak systém funguje**. Diagramy chování představují **dynamický aspekt** systému. Zdůrazňují, co se může stát v modelovaném systému. Jsou široce používány k **popisu funkčnosti softwarových systémů**. Diagram případu užití například zobrazuje uživatele v nějaké roli a operace, které může provádět.

Diagram případů užití

Diagram případů užití zachycuje vnější pohled na modelovaný systém, specifikujeme pomocí něj **účastníky** a způsoby, jak budou modelovaný **systém používat** - případy užití. Používá se v Use-Case driven přístupech (metodika RUP). Prvky diagramů užití:

- **hranice systému**,
- **účastník (aktér/actor)**: subjekt (většinou uživatel, může mít i speciální podobu, např. čas, jiný systém), který se systémem pracuje,
- **případ užití**: funkce, kterou systém vykonává jménem (akcí) jednotlivých účastníků nebo v jejich prospěch,
- **interakce**: ukazuje účast autora (účastníka) na provádění případu užití.

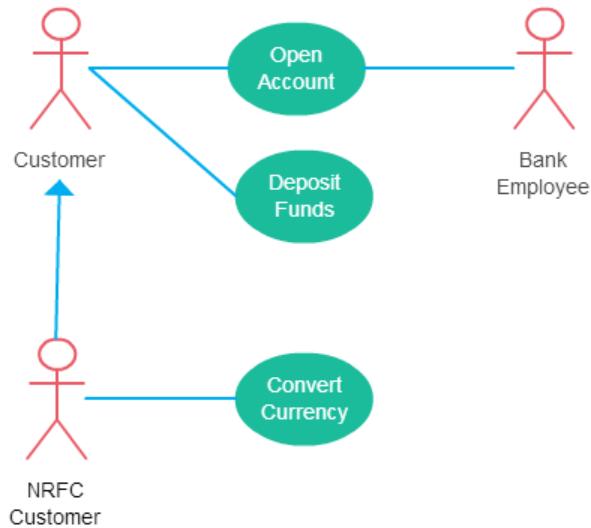
Asociace mezi účastníkem a případem užití

Účastník musí být pomocí asociace spojen aspoň s jedním případem užití. Více účastníků může být asociováno s jedním případem užití (viz obrázek na konci kapitoly).

Generalizace účastníka (dědičnost)

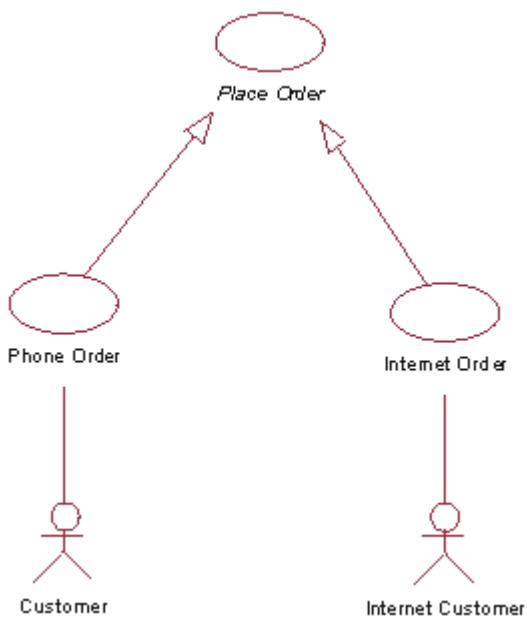
Zobecnění účastníka znamená, že jeden účastník může **zdědit roli druhého**.

Potomek zdědí všechny případy použití předka. Potomek má navíc jeden nebo více případů použití, které jsou pro něj specifické (pro danou roli) a předek ji nemá.



Generalizace případu užití

Je to podobné jako zobecnění účastníka. Chování předka dědí potomek. Používá se, když existuje **společné chování mezi dvěma případy užití** a také **specializované chování specifické pro každý případ použití**. Viz příklad na obrázku.

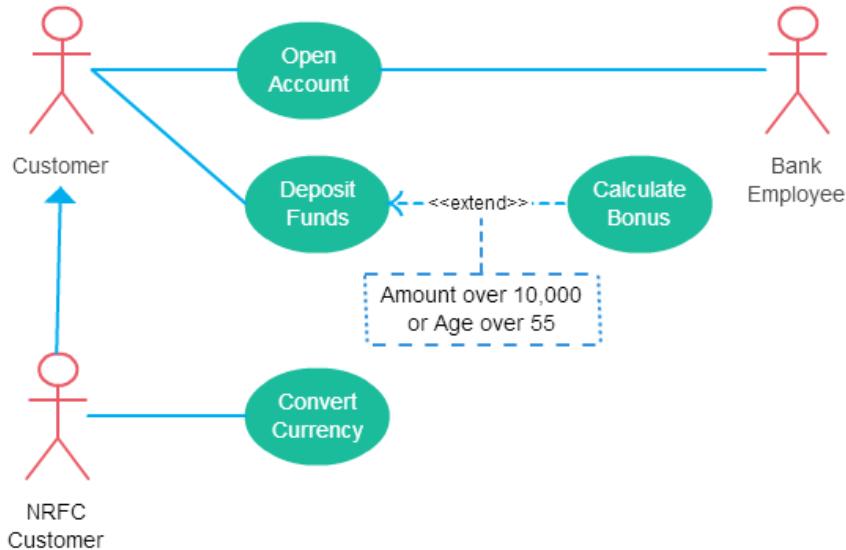


Extend mezi dvěma případy užití

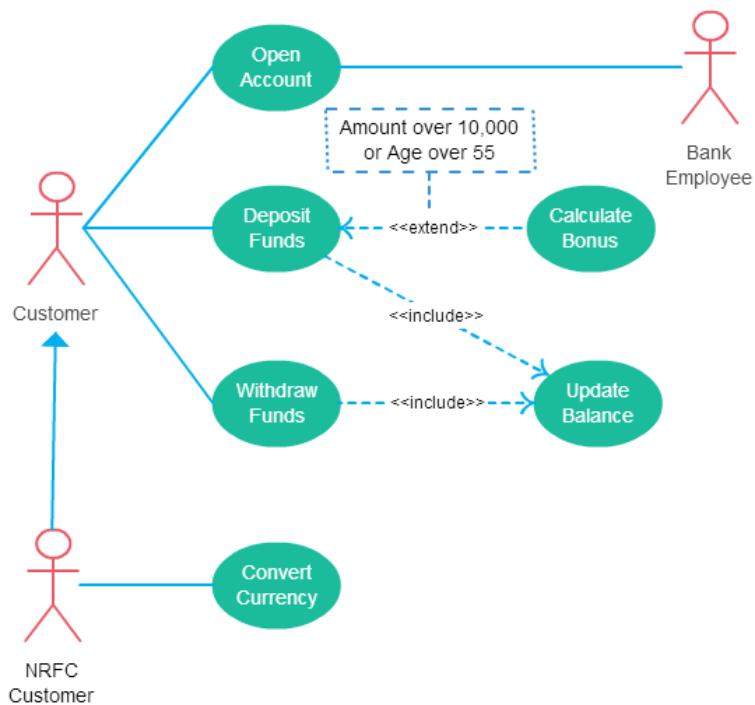
Vazba extend rozšiřuje funkcionality daného případu užití (přidává do systému další funkcionality). Případ užití vázaný pomocí extend (rozšiřující případ užití) se

může provést ale nemusí (záleží na okolnostech, případně se může provádět pouze podmíněně). Rozšiřující případ užití je **závislý** na rozšiřovaném případu užití (sám o sobě nedává smysl). Naopak **rozšiřovaný případ užití musí dávat** sám o sobě **smysl**.

Include mezi dvěma případy užití



Zahrnutý/přiložený (included) případ použití **je povinný a není volitelný**. Základní případ použití je **bez přiloženého případu použití neúplný**. Používá se v případě, že je nějaká funkcionálnita **důležitá natolik**, že nemůže být součástí nějakého případu užití. Chceme jí tímto zdůraznit.



Detaily případů užití

Slouží pro konkretizaci use case. Často se používá **tabulka** - má vstupní podmínky a tok událostí.

název identifikátor	Případ užití: Platit daň z přidané hodnoty
ID: UC1	
účastníci:	
Čas	
finanční úřad	
stav před	
Vstupní podmínky:	
1. Je konec fiskálního čtvrtletí?	
kroky	
Tok událostí:	
1. Případ užití začíná na konci fiskálního čtvrtletí.	
2. Systém určuje výši daně z přidané hodnoty, kterou je třeba odvést státu.	
3. Systém odesílá elektronickou platbu finančnímu úřadu.	
stav po	
Následné podmínky:	
1. Finanční úřad přijímá daň z přidané hodnoty.	

Příklad diagramu užití

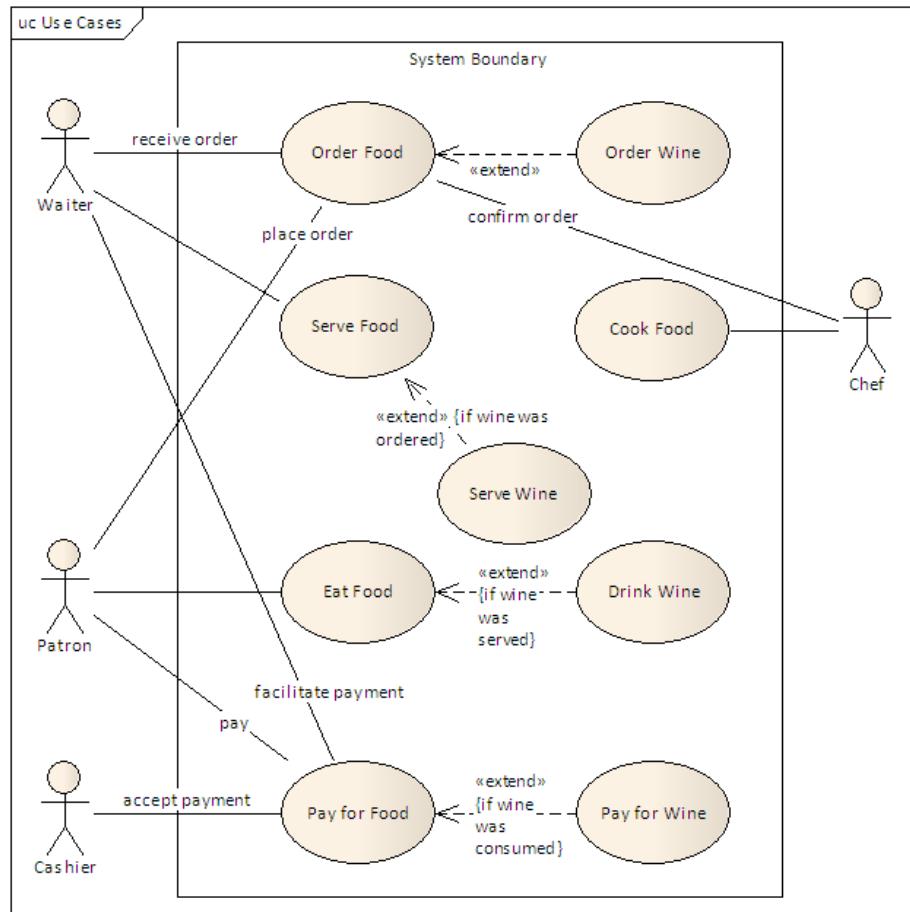


Diagram aktivit (Activity diagram)

Reprezentuje objektově orientovaný **vývojový diagram**. Umožňuje modelování:

- scénářů případů užití,
- detailů **algoritmů** a operací (funkcí),
- modelování **obchodních procesů**.

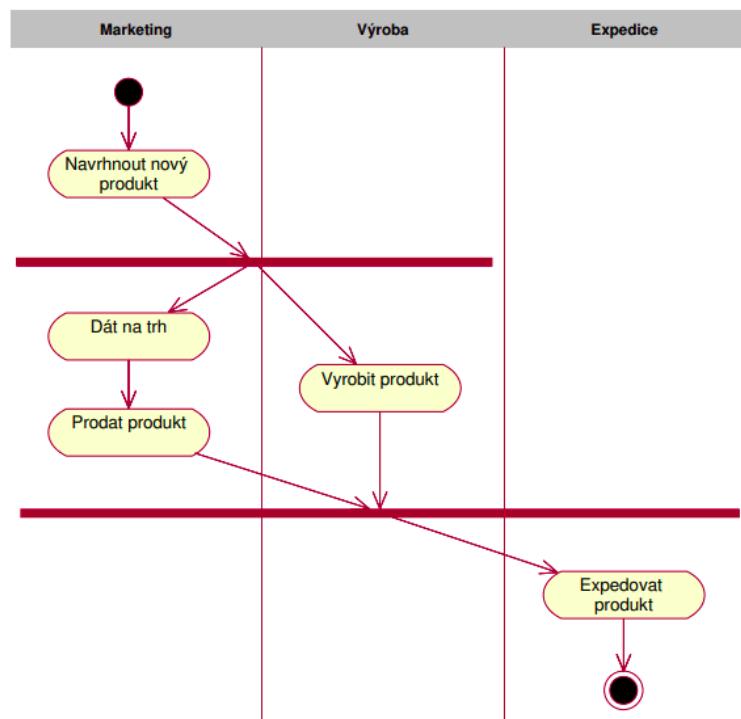
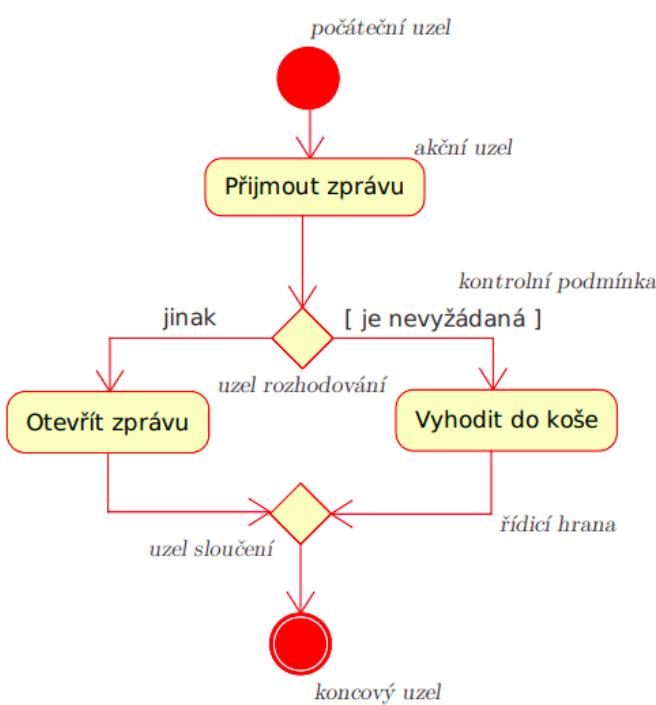
Prvky diagramu:

- **uzly**:
 - **akční uzly**: modelují aktivitu,
 - **řídící uzly**: modelují rozhodování, např. počáteční uzel, koncový uzel,
 - **objektové uzly**: modelují objekty podílející se na aktivitách.
- **hrany**:
 - **řídící hrany**: modelují přechody mezi uzly,
 - **objektové hrany**: modelují cesty objektů mezi uzly.

Možnosti modelování

- tok událostí a dat,
- rozhodování,

- větvení a slučování,
- iterace,



- paralelní toky.

Stavový diagram

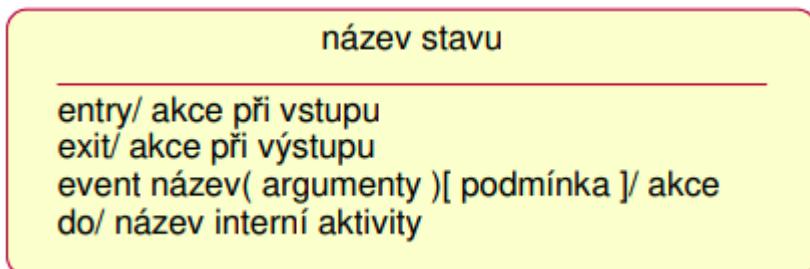
Umožňují modelování životního cyklu jednoho reaktivního (měnící se) objektu. Je to zvláštní případ stavového automatu. Mohou také modelovat dynamické chování těchto objektů:

- třídy, respektive instance tříd (objekty),
- případy užití,

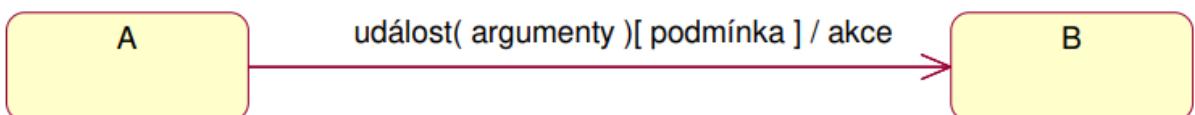
- podsystémy,
- systémy.

Je tvořen:

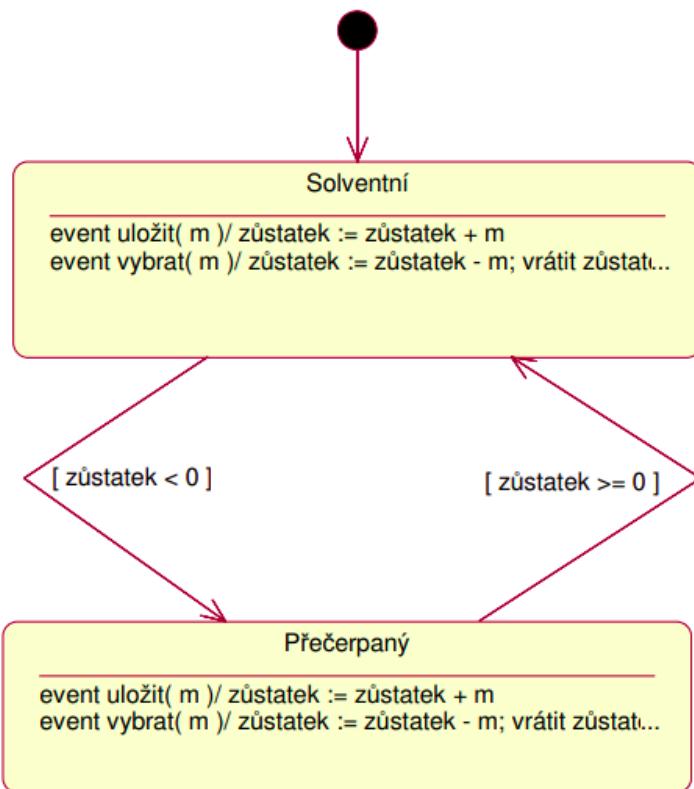
- **stavy**,



- **přechody**,



- **události**: udávají, kdy dojde k přechodu z jednoho stavu do druhého. Události mohou být i ve stavech.



Reaktivní objekt

- reaguje na vnější události,
- životní cyklus je modelován jako řada stavů, přechodů a událostí,
- chování je důsledkem předchozího chování, následující stav závisí na předchozím stavu.

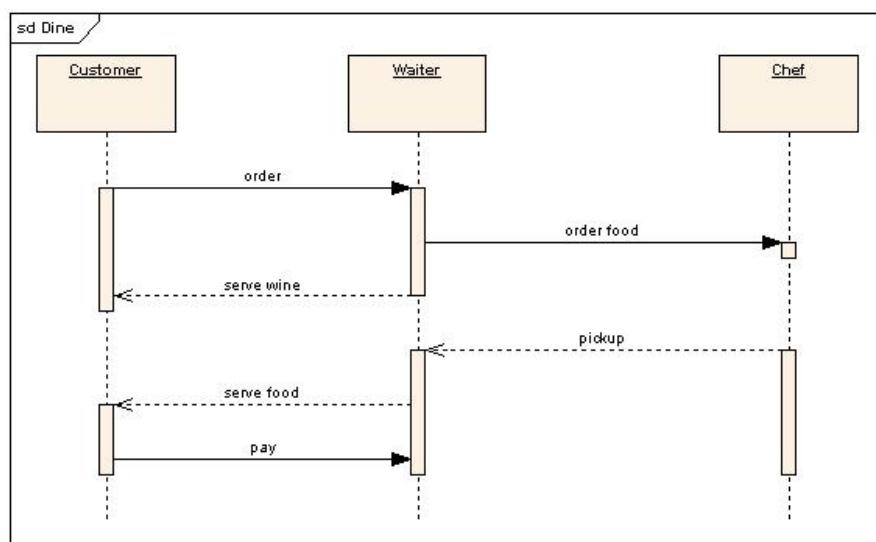
Diagramy interakce (Interaction diagrams)

Jedná se o **podskupinu** diagramů chování. Popisují **interakci mezi jednotlivými částmi systému** (včetně uživatele). Zdůrazňují **tok řízení** a dat mezi částmi v modelovaném systému. Sekvenční diagram například ukazuje, jak spolu objekty komunikují pomocí **sekvence zpráv**.

Sekvenční diagram

Zdůrazňuje časově orientovanou posloupnost předávání zpráv mezi objekty - chronologické zaslání zpráv. Sekvenční diagramy jsou většinou přehlednější a srozumitelnější než diagramy komunikace. Jsou tvořeny:

- **čarami života**: zobrazuje časovou osu určitého objektu,
- **vodorovné šipky**: reprezentují zprávy posílané mezi objekty, existuje více typů.



Sekvenční diagram může být například rozšířen pomocí **omezení**, **zobrazení stavů** atd.

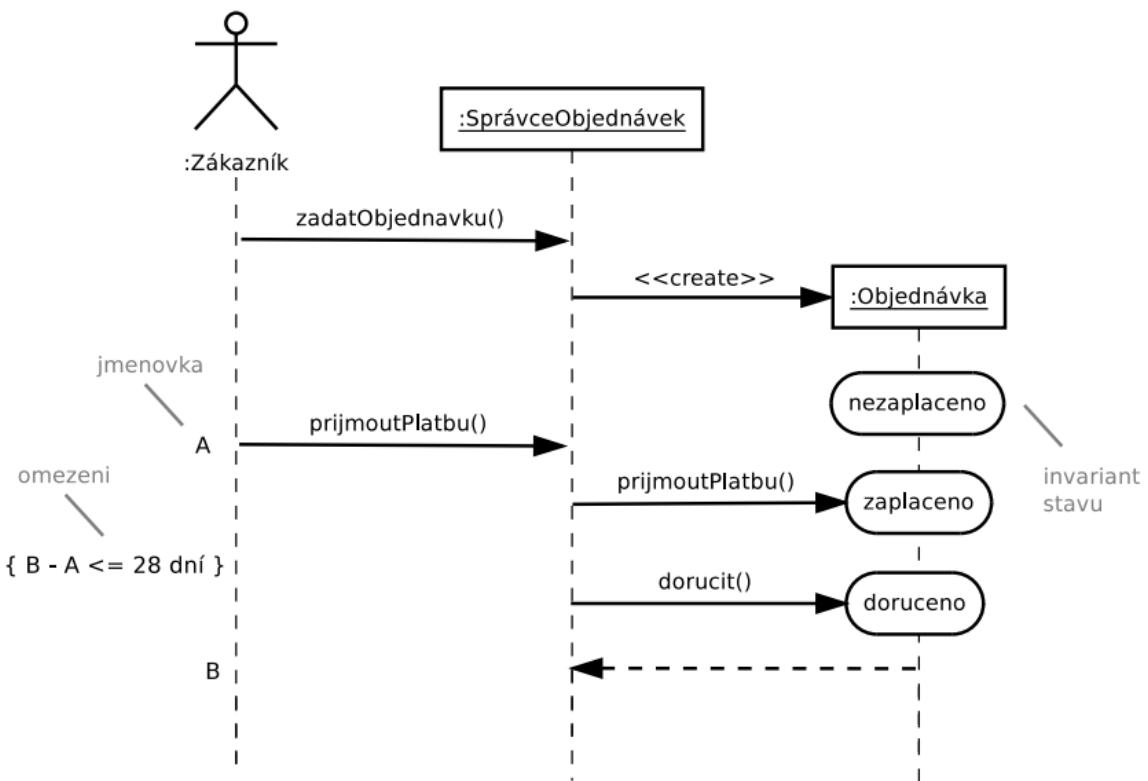
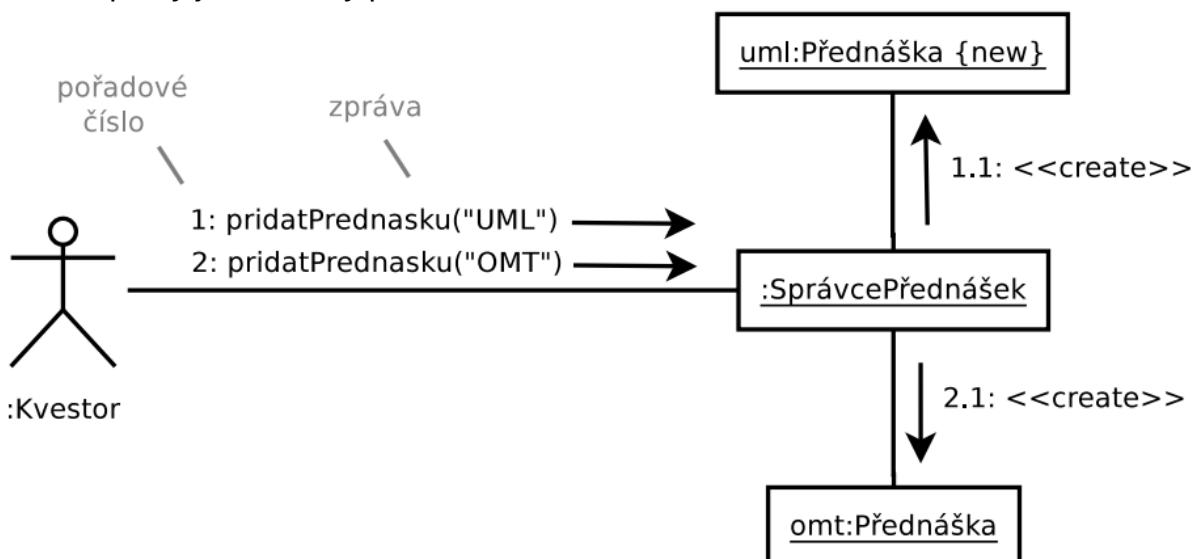


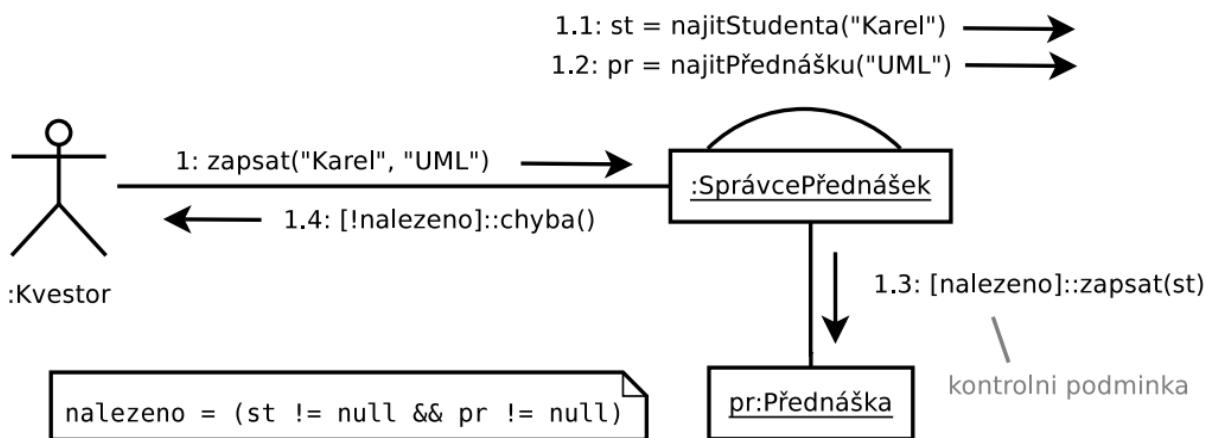
Diagram komunikace

Zdůrazňují strukturální vztahy mezi objekty, jsou vhodné spíše pro rychlé zobrazení komunikace mezi objekty. Jsou méně přehledné než sekvenční diagramy.

- objektu jsou **spojeny linkami**,
- zprávy jsou řazeny podle **hierarchického číslování**.



Diagramy komunikace mohou být doplněny o **větvení, kontrolní podmínky atd.**



Základní pohledy

Projekce systému na jeden z jeho klíčových aspektů.

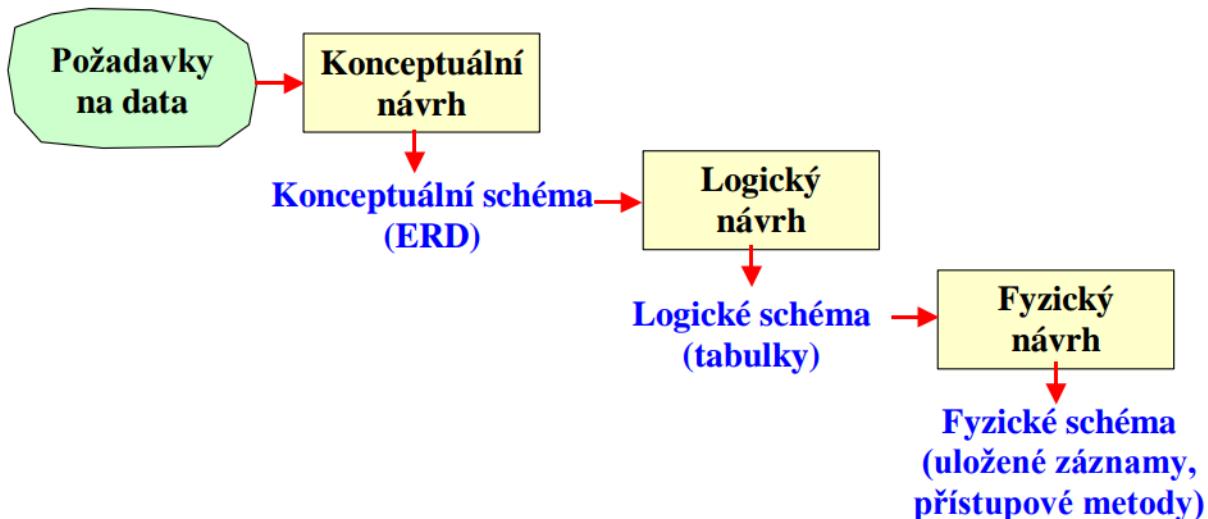
- **Strukturální** - Popisuje vrstvu mezi objekty a třídami, jejich asociace a možné komunikační kanály.
- **Datový** - Popisuje stavy systémových komponent a jejich vazby.
- **Pohled na chování** - Popisuje jak systémové komponenty interagují a charakterizuje reakce na vnější systémové operace.
- **Pohled na rozhraní** - Je zaměřeno na zapouzdření systémových částí a jejich potenciální použití okolím systému.

35. Konceptuální modelování a návrh relační databáze.

Konceptuální modelování

Konceptuální modelování je první krok při **návrhu uložení dat v databázi** (návrhu databáze). Patří do etapy **analýzy požadavků**. Jeho cílem je analyzovat požadavky **na data**, která budou **uložena v databázi**. Zabývá se **modelováním reality** (objektů a jejich vlastností, které potřebujeme ukládat). Snaží se **nebýt** ovlivněno budoucími prostředky řešení. Umožňuje **sjednotit chápání** aplikace mezi uživateli (zákazníky), analytiky a programátory. Výsledek je **použit při logickém návrhu databáze** (jednotlivých tabulek) a slouží také jako **dokumentace**. Obvykle se provádí graficky pomocí:

- **ER modelů** popsaných ER diagramy: jedná se o **strukturovaný** přístup. Při strukturovaném návrhu lze však také použít OOP při implementaci IS.
- **diagramu tříd**: jedná se o objektově orientovaný přístup.



Logický návrh

Cílem je navrhnut **strukturu databáze** (strukturu jednotlivých tabulek). Popisuje, jak jsou **data uložena** v databázi, **vztahy** mezi nimi a **integritní omezení**, tak aby neexistovala redundance a struktura dat v databázi byla co nejjednodušší.

Fyzický návrh

Jde o **fyzické uložení dat** (sekvenční soubor, indexy, clustery, ...). Musí se navrhnut vzhledem k **SŘBD** pro efektivní přístup.

Entity Relation model/Entity Relation diagram

ER model je založen na chápání světa jako množiny základních objektů - **entit** (Entity) a vztahů (Relations) mezi nimi. Popisuje data **staticky** ("v klidu"),

neukazuje, jaké **operace** s daty budou probíhat. Někdy se označuje také jako ERA – třetím základním prvkem modelu jsou **atributy** (Attributes) jednotlivých entit nebo vztahů.

Entita

Objekt reálného světa rozlišitelný od jiných objektů, o níž chceme mít informace v DB. Jedná se o **konkrétní objekt**, např. klient s ID 25.

Entitní množina (typ entity)

Definuje typ/**množinu entit**, které **sdílí tytéž vlastnosti** neboli atributy. Jedná se skupiny stejných objektů, např. klient, bankovní účet, ...

Atribut

Vlastnost entity nebo vztahu, která nás v kontextu daného problému **zajímá a jejíž hodnotu chceme mít v DB** uloženu. Atributy mohou být:

- **jednoduché** (PSČ) a **složené** (celá adresa),
- **jednohodnotové** (popis - objekt má obvykle pouze jeden popis) a **vícehodnotové** (telefon - můžeme požadovat uložení více tel. čísel),
- **povolující prázdnou hodnotu** - NULL (hodnota neexistuje nebo může existovat, ale mi jí zatím neznáme, např. popis) a **nepovolující prázdnou hodnotu** (objekt nemůže existovat bez této hodnoty - např. název),
- **odvozené** (věk je odvozený od data narození) v **OLTP** ne, v **OLAP** ano.

Doména atributu

Obor hodnot atributu.

Vztah

Asociace (**spojení**) mezi dvěma nebo více entitami. Např. klient s číslem klienta K999 vlastní účet s číslem účtu U100.

Primární klíč

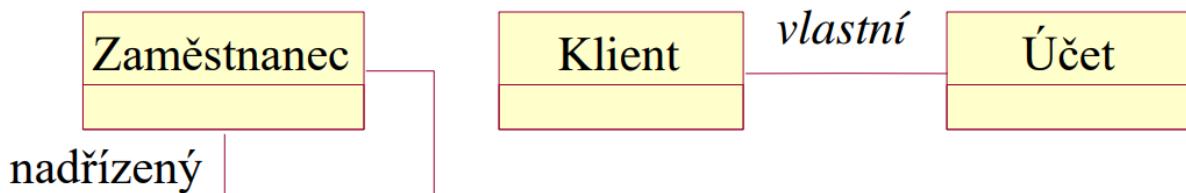
Jedná se o jeden (můžeme vybrat) z **kandidátních klíčů**, což je atribut nebo množina atributů, který/která je v dané množině entit unikátní. Kandidátní klíč musí být minimální (nemůže existovat kandidátní klíč, který má méně atributů).

Vztahy

Spojují **entitní množiny** a vyjadřují vztahy mezi nimi.

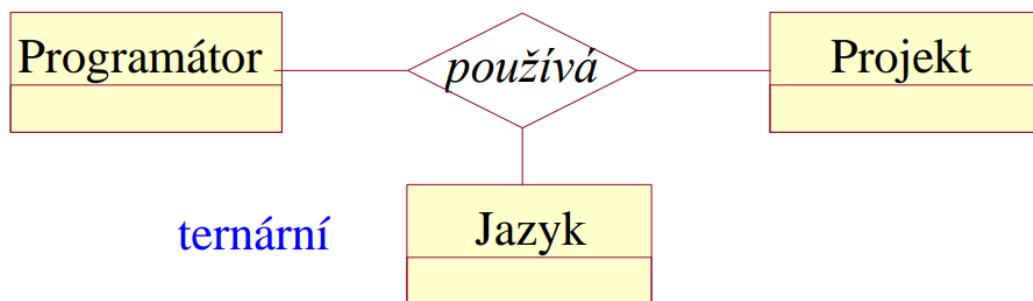
Stupeň

Určuje kolik je vztahem **propojeno entitních množin**.



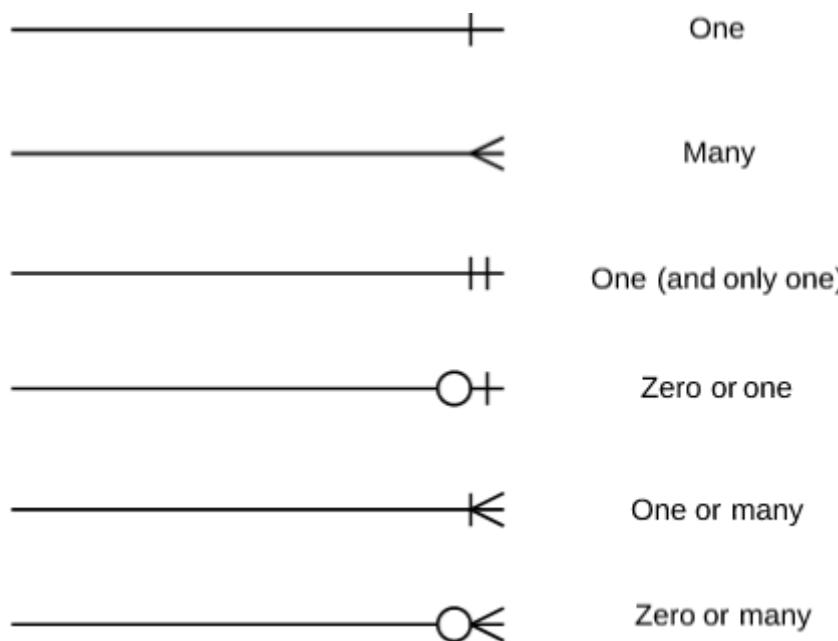
unärní (reflexivní)

binární



Kardinalita

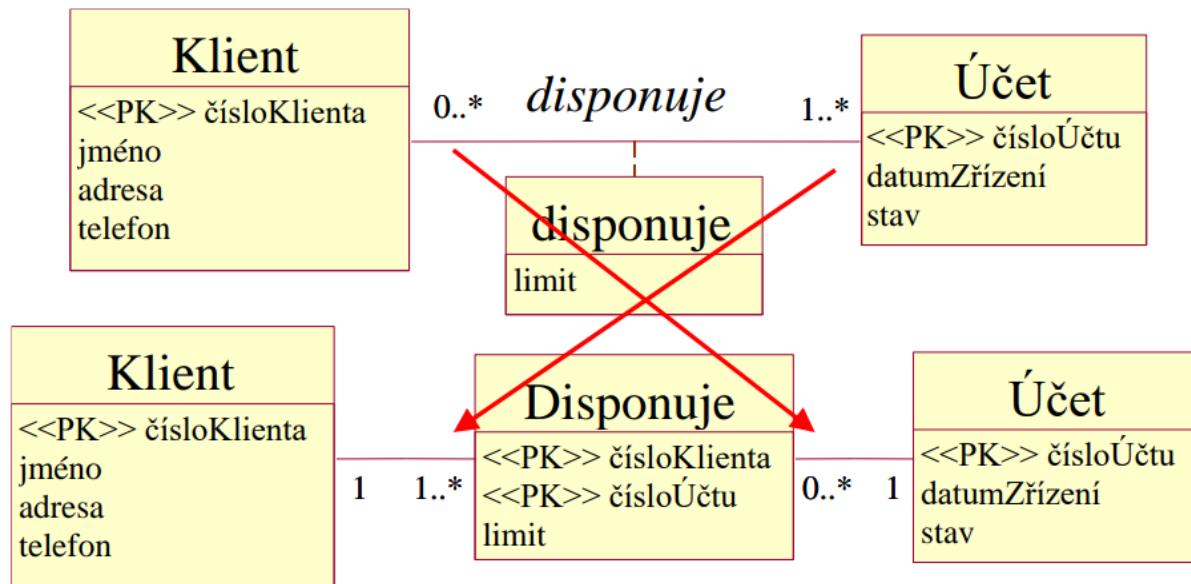
Udává **maximální počet vztahů** daného typu, ve kterých může **participovat jedna entita** (1, M, případně přesněji) z entitní množiny. Jedná se o vlastnost **každého "konce"** vztahu. Pro návrh schématu databáze je rozhodující správné určení **maximální kardinality**. Kardinalitu značíme znaky (0, 1, *) nebo pomocí symbolů na obrázku.



Atributy vztahu

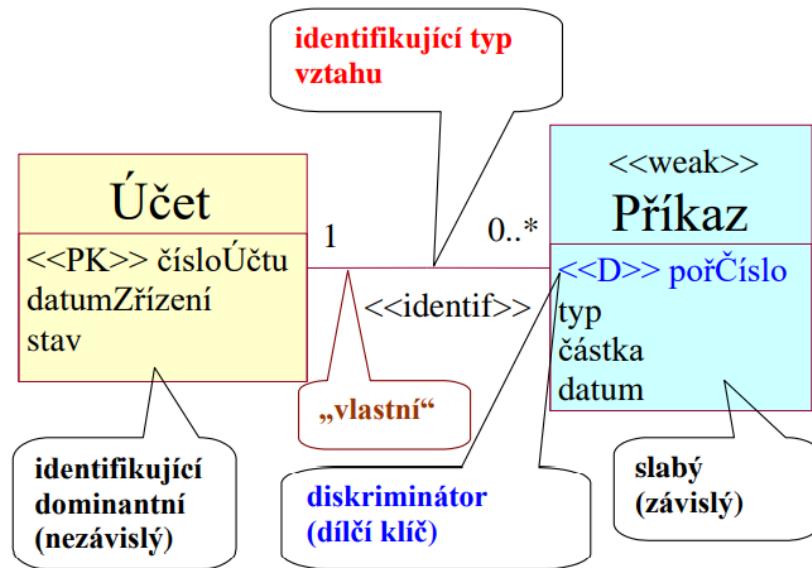
Rozvíjí vztah, do databáze ale musí být uloženy ve zvláštní (vazební) tabulce.

Převádíme je na tuto tabulku viz druhý obrázek.



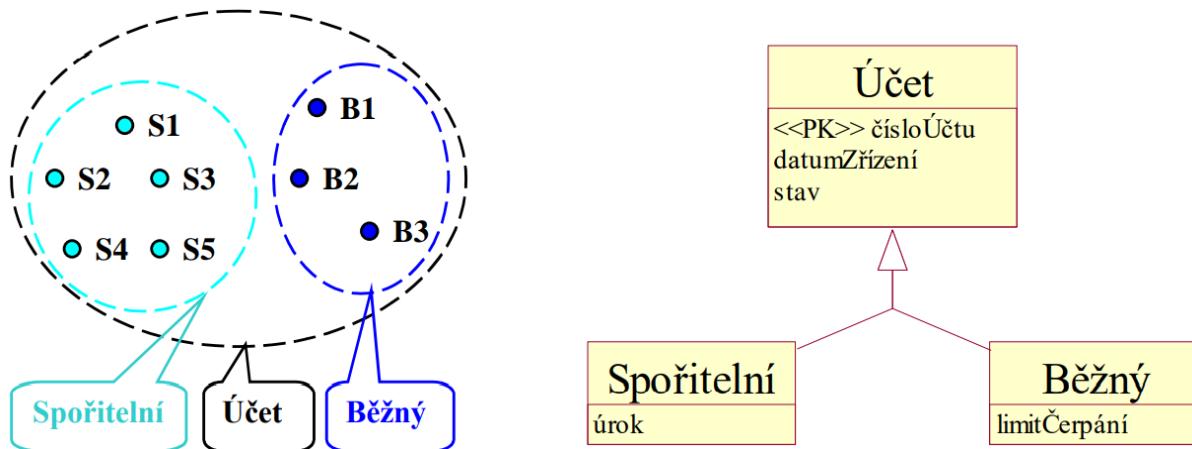
Slabé a silné entity

- **silná entita** může existovat nezávisle na ostatních,
- **slabá entita** je závislá na jiném **jednom dominantním** typu entity. Slabé entity jsou identifikovány **primárním klíčem silné entity a diskriminátorem - dílčím klíčem**. Primární klíč silné entity slouží jako **cizí klíč**. Nemohou existovat sami o sobě a při zániku dominantní entity také zanikají.



Generalizace a specializace

Vychází z principů OOP. Umožňuje rozšiřovat entity o **další atributy** (specializované entity) a současně **sdílet jiné** (obecná entita). Existovat sama o sobě může i obecná entita. Primární klíč odvozené (specializované) entity je stejný jako ten obecné.



Jména/Názvy

- **srozumitelná**, musí vyjadřovat význam typů entit a vztahů,
- **typ entit**: podstatná jména,
- **typ vztahů**: slovesa, předložky,
- je-li jméno typu vztahu jasné ze jmen typů entit, není nutné ho uvádět,
- při několika **různých typech vztahů** mezi **stejnými** entitními množinami je **nutné** použít jméno vztahu.

Rozdíl ER diagramu a diagramu tříd

ER diagram	Diagram tříd
Typ entity	Třída
Entita	Objekt
Atribut	Proměnná/atribut
–	Operace/metoda
Typ vztahu	Asociace
Vztah	Vazba (link)
Kardinalita	Násobnost
Typ slabé entity	Kvalifikovaná asociace
Generalizace/specializace	Generalizace/specializace

Návrh relační databáze

Schéma relační databáze lze získat jedním z následujících dvou způsobů:

- vytvořením **konceptuálního modelu** a jeho transformací (transformací ER diagramu),
- **použitím normalizace** s tím, že na počátku předpokládáme, že **všechny informace** budou uloženy **v jedné tabulce** a tu normalizujeme.

Transformace ER diagramu na tabulky relační databáze

Jedná se o další krok při návrhu databáze. Jde o převod z **konceptuálního návrhu na návrh logický**. Tento způsob se používá častěji než normalizace, ale současně dbáme na to, aby data byla v požadované normální formě.

- **Vztahy s kardinalitou 1:1** ujistíme se, že je opravdu vhodné vytvářet tento vztah a není lepší tabulky **spojit do jedné** (neplatí při vazbě 0..1). Jinak rozšíříme jednu z tabulek o **cizí klíč** do druhé a přidáme do ní **atributy vztahu**.
- **Vztahy s kardinalitou 1:M** záznamy tabulky, které jsou ve vztahu **pouze s jedním záznamem** (vystupují s kardinalitou 1) druhé tabulky musí obsahovat **cizí klíč** do této tabulky. Tyto záznamy musí být také **rozšířeny o atributy vazby**, protože na libovolný záznam druhé tabulky může takto **odkazovat M záznamů první tabulky**.
- **Vztahy s kardinalitou M:M** reprezentujeme přidáním do databáze (vazební) tabulky. Primární klíč se obvykle volí jako **složený klíč z primárních klíčů původních tabulek**. Do vazební tabulky se také přidávají atributy vazby.
- **Vztah vyššího stupně** z naprosté většiny vyžaduje tvorbu vazební tabulky, která jako.
- **Vztah slabé entitní množiny**: tato vazba je **vždy 1:M** a provádí se stejně s tím, že tabulka reprezentující slabou entitní množinu má složený primární klíč z cizího klíče do **dominantní tabulky a diskriminátora**.
- **Generalizace a specializace** lze transformací do tabulek databáze řešit třemi způsoby
 - a. Vytvoření obecné tabulky se sdílenými atributy a tabulek specializovaných, které obsahují pouze specifické atributy. Mají **stejný primární klíč**, který je **současně cizím klíčem** do obecné tabulky.
 - **výhody**: Tato varianta je vhodná pro **disjunktní i překrývající se** specializace a pro **specializaci úplnou i částečnou**. Vhodné také při očekávání **nových specializací**.
 - **nevýhody**: nutnost provádět operaci **spojovalní**.
 - b. Vytvoření dvou nezávislých tabulek, které budou obsahovat atributy obecné entitní množiny a poté každá atributy své specializované entitní množiny. Tento způsob **neumožňuje vytváření pouze obecných entit** (musely by se ukládat do jedné ze specializovaných tabulek, což není vhodné).
 - **výhody**: Není potřeba provádět operace spojování - join.
 - **nevýhody**: specializace musí být **úplná** (nemůže existovat obecný záznam) a **disjunktní**, **nelze** provádět **společné** operace nad obecnými daty.
 - c. Vytvoří se pouze **jedna tabulka**, která obsahuje **obecné atributy i atributy všech specializací**. Jednotlivé specializace pak můžeme rozlišovat na základě **testu prázdné hodnoty** ve specializovaných sloupcích (pokud je to možné a některý ze sloupců specializace

nemůže být prázdný) nebo přidáním speciálního sloupce - **diskriminátoru**.

- **výhody**: není potřeba provádět spojování,
- **nevýhody**: může vést na velké tabulky a mnoho prázdných hodnot, zejména při větším počtu disjunktních specializací. V tomto případě je také vhodné přidat další sloupec identifikující jednotlivé specializace, aby se nemusel provádět složitý test na prázdné hodnoty.

d. Vytvoření tabulky pro **obecnou entitní množinu** a **jedné tabulky pro všechny specializace**, v této tabulce lze opět zavést **diskriminátor**.

Primární klíč v obou tabulkách bude stejný a slouží i jako cizí klíč.

- **výhody**: Lze rychle pracovat s daty obecné entitní množiny.
- **nevýhody**: spojování, prázdné hodnoty, test na prázdnost sloupců.

Normalizace

Normalizace je druhý způsob, jak můžeme postupovat při návrhu relační databáze pro uložení požadovaných dat. Tento způsob (pokud provedený správně) **zajišťuje, že databáze nebude trpět nedostatky špatného návrhu**. Při použití normalizace se vychází z **jedné tabulky**, ve které jsou uložena všechna data. Pokud není tabulka navržena správně, je **nutné ji rozdělit** na dvě nebo více tabulek jednodušších. To, jestli je tabulka navržena správně nebo není určuje normální forma, podle které tabulku konstruujeme. Pro **OLTP** systémy požadujeme alespoň **3. NF**.

1. Normální forma

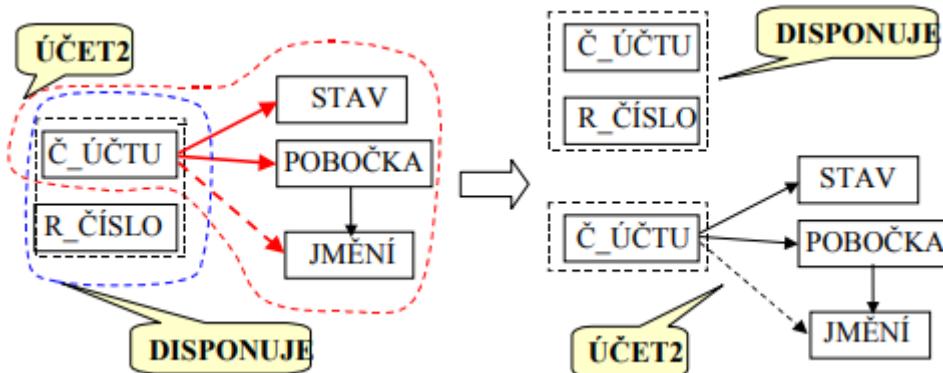
Schéma relace je v 1. normální formě, pokud její atributy obsahují pouze **atomické** (skalární) hodnoty, které nelze dál dělit. Typickým příklad převodu tabulky do 1. normální formy je **převod atributu adresa** na několik atributů jako: **země, město, PSČ, ulice, číslo popisné, ...** Na obrázku u příkladu s telefony je také možné řešení **vytvořit vazbu 1:M**.

Osoba			
Jméno	Příjmení	Adresa	Telefony
Jan	Novák	Havlíčkova 2 Praha 3	125789654;601258987;789456123
Petr	Kovář	Svatoplukova 15 Brno	369852147;357951456;963852741
Pavel	Pavel	Papalášova 25 Kocourkov	546789123;123456789;987456123

2. Normální forma

Schéma relace (tabulka) je v druhé normální formě, pokud je v **1. NF** a každý její **neklíčový atribut**, je **plně funkčně závislý** na každém kandidátním klíči relace (musí být závislý na všech attributech složeného klíče). Převod do 2. NF zajišťujeme

rozdělením tabulky na více tabulek a propojením pomocí cizích klíčů. Na obrázku tímto **cizím klíčem** bude **Č_UČTU**.



3. Normální forma

Schéma relace je ve třetí normální formě, právě když je ve 2NF a neexistuje žádný **neklíčový atribut**, který je **tranzitivně závislý** na některém **kandidátním klíči** relace. Tranzitivní závislost je závislost mezi minimálně dvěma atributy a klíčem, kde **jeden atribut je funkčně závislý na klíči a druhý atribut je funkčně závislý na prvním atributu**. Řešíme tak, že vytvoříme novou tabulku s těmito atributy, kde první atribut je klíčem a v původní tabulce zůstane pouze první atribut jako cizí klíč do nové tabulky.

Zaměstnanec										
Os. č	Jméno	Příjmení	Ulice	C. pop.	C. or.	Město	PSC	Funkce	Plat	
1	Jack	Smith	Studentská	1151	12	Jihlava	58601	CEO	150000	
2	Franta	Vomáčka	Prášilova	1235	25	Praha10	10000	Senior Software Architect	80000	
3	Pepa	František	E. Beneše	12	57a	Plzeň	12345	Senior Software Architect	80000	
4	Pavel	Novák	Mánesova	157	14	Kocourkov	99999	Junior Developer	30000	
5	Petr	Koukal	Kluzká	1855	28	Praha10	10000	Database Designer	75000	
6	Honza	Novák	U Klausů	28	15a	Plzeň	12345	Junior Developer	30000	

transformujeme na:

Zaměstnanec

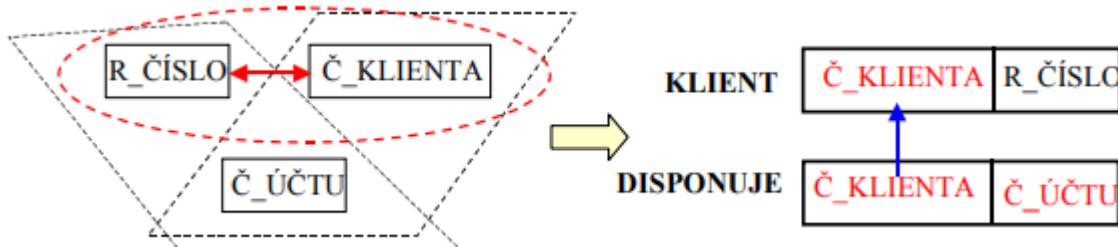
Os. č	Jméno	Příjmení	Ulice	C. pop.	C. or.	Město	PSC	Funkce
1	Jack	Smith	Studentská	1151	12	Jihlava	58601	1
2	Franta	Vomáčka	Prášilova	1235	25	Praha10	10000	2
3	Pepa	František	E. Beneše	12	57a	Plzeň	12345	2
4	Pavel	Novák	Mánesova	157	14	Kocourkov	99999	3
5	Petr	Koukal	Kluzká	1855	28	Praha10	10000	4
6	Honza	Novák	U Klausů	28	15a	Plzeň	12345	3

Zaměstnanec

ID_Fun	Funkce	Plat
1	CEO	150000
2	Senior Software Architect	80000
3	Junior Developer	30000
4	Database Designer	75000

Boyce-Coddova normální forma

Schéma relace je v Boyce-Coddově normální formě, pokud je ve 3. normální formě a v relaci existuje **pouze jeden** kandidátní klíč, nebo jich existuje více, ale jsou **disjunktní** (nemají společný atribut). Jinak řečeno, pokud v relaci **existují dva** (nebo více) **složené** kandidátní klíče, které **sdílí nějaký atribut**, není relace v **BCNF**. Opět řešíme tak, že relaci rozdělíme na dvě (2 tabulky) a sdílený atribut použijeme jako cizí klíč.



Funkční závislost

Pokud je **Y** funkčně závislé na **X** ($X \rightarrow Y$), pak se **nemůže stát** aby **2 řádky** mající stejnou hodnotu **X** měly různou hodnotu **Y**. (nepřesný příklad: **Datum narození** je funkčně závislé na **rodném čísle** - nemůže se stát že u 2 záznamů **se stejným rodným číslem** bude rozdílné datum narození. Nepřesný protože nelze mít 2 záznamy se stejným primárním klíčem).

36. Reprezentace a uložení strukturovaných dat, serializace a deserializace, relační datový model, jazyk SQL, transakce (DB a business).

Reprezentace a uložení strukturovaných dat

Nestrukturovaná data (celočíselná, reálná, znaky, řetězce, datum - záleží jak uložené, čas, výčtové typy) mají často význam jen pokud je ukládáme strukturovaně. Existují základní dva způsoby, jak strukturované datové typy vytvářet. Jedná se o **strukturu a kolekci**.

Struktura (prostá struktura)

Struktura je tvořena **pevným počtem pojmenovaných dílčích hodnot** (dvojice jméno a hodnota) obecně **různých datových typů**. Jedná se o **uspořádanou n-tici**, jejíž prvky jsou prvky kartézského součinu více množin. Často **strukturu** nazýváme také jako **záznam**. Při ukládání dat v (operační) paměti pomocí struktury se názvy jednotlivých dílčích hodnot neukládají (pracuje se s offsetem), názvy se připojují až při serializaci.

Kolekce

Kolekce je tvořena **předem neomezeným** (tj. proměnným) počtem hodnot **stejného datového typu**. Matematicky se jedná o **množinu**, respektive **multimnožinu**, protože obvykle chceme umožnit vícenásobné uložení stejného prvku. **Kolekci** také často nazýváme jako **seznam, posloupnost** nebo **řetězec**. Do kolekce můžeme **vkládat** prvky, **získávat** jejich hodnoty a **mazat** je. K tomu používáme **iterátor** (ukazovátko do kolekce). Dále nad kolekcí můžeme provádět souhrnné operace jako získání počtu prvků, průměrné hodnoty, největší, nejmenší, ... případně můžeme iterovat přes všechny prvky. Nad kolekcí může existovat jedno nebo více definovaných uspořádání podle klíčů jejich prvků.

Objekt

Objekt je struktura, kterou lze jednoznačně identifikovat - je mu přiřazena jednoznačná identifikace (**OID** - object identification). Tuto skutečnost využíváme při ukládání do databází, OID většinou **generuje SŘBD**. Díky OID může objekt vystupovat ve vztazích.

Ukládání strukturovaných dat

Strukturovaná data jsou obvykle tvořena různě **zanořenými strukturami** a

kolekcemi. Mohli bychom je do souboru ukládat binárně, tak jak jsou uložené v operační paměti a informace, co a jak (tj. **metadata** o datech) je v tomto souboru uložené nějak jinak např. do dalšího souboru. Tento způsob je ale dost nepraktický, proto používáme **standardizované formáty** uložení (JSON, XML, YAML, ...), ve kterých jsou uloženy i (některá) metadata o ukládaných datech (jejich **názvy**, jiná metadata jsou ukládána např. pomocí **Document Type Definition** (DTD) a **XML Schema Definition** (XSD) v případě XML a **JSON-LD** v případě JSON). A formát těchto souborů (**meta2data**) jsou právě popsána standardem a již se nemusí s daty předávat.

Serializace

Serializace (marshalling) je proces **konvertování** datových **struktur** nebo **stavů objektů** do formátu, který je čitelný člověkem i strojově a může být **uložen na disk** nebo **přenášen po síti**.

Deserializace

Při deserializaci provádíme **rekonstrukci** serializované hodnoty na **tentýž nebo i jiný formát** (např. serializovaný objekt s určitým pojmenováním atributů můžeme deserializovat do jiného objektu, který má ale stejně pojmenované atributy, ale může mít nějaké další nebo nějaké mohou chybět).

Relační datový model

Relační databáze jsou založené na definicích **množin**, **kartézského součinu** a **relací**, jak je známe z matematiky. **Relace je podmnožinou kartézského součinu** a v případě relačních databází ji chápeme jako **tabulkou**. Konkrétně tabulku tvoří **schéma relace** (záhlaví tabulky - názvy sloupců) a **tělo relace** (představuje uložená data v tabulce po řádcích). Počet atributů relace označujeme jako **stupeň** (řád) relace, kardinalita těla relace (počet řádků) označujeme jako **kardinalitu relace**. Postup odvození tabulky z matematických definicí:

$$D_{\text{LOGIN}} = \{\text{xnovak00}, \text{xnovak01}, \text{xcerny00}, \text{xzelen05}, \text{xmodry02}, \dots\}$$

$$D_{\text{JMÉNO}} = \{\text{Eva}, \text{Jan}, \text{Pavel}, \text{Petr}, \text{Zdeněk}, \dots\}$$

$$D_{\text{PŘÍJMENÍ}} = \{\text{Adam}, \text{Černý}, \text{Novák}, \text{Modrý}, \text{Zelená}, \dots\}$$

$$D_{\text{ADRESA}} = \{\text{Cejl 9 Brno}, \text{Purkyňova 99 Brno}, \text{Brněnská 15 Vyškov}, \dots\}$$

Relace R_{STUDENT} by potom mohla vypadat například následovně:

$$R_{\text{STUDENT}} \subseteq D_{\text{LOGIN}} \times D_{\text{JMÉNO}} \times D_{\text{PŘÍJMENÍ}} \times D_{\text{ADRESA}}$$

$$\begin{aligned} R_{\text{STUDENT}} = & \{(\text{xcerny00}, \text{Petr}, \text{Černý}, \text{Brněnská 15 Vyškov}), \\ & (\text{xnovak00}, \text{Jan}, \text{Novák}, \text{Cejl 9 Brno}), \\ & (\text{xnovak01}, \text{Pavel}, \text{Novák}, \text{Cejl 9 Brno})\} \end{aligned}$$

$R_{STUDENT} : \{ ($

xcerny00,	Petr,	Černý,	Brněnská 15 Vyškov
xnovak00,	Jan,	Novák,	Cejl 9 Brno
xnovak01,	Pavel,	Novák,	Cejl 9 Brno

$)$,
 $)$,
 $\} \}.$



$R_{STUDENT} :$

xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

Přidání schématu (záhlaví) relace

login: D_{LOGIN}	jméno: $D_{JMÉNO}$	příjmení: $D_{PŘÍJMENÍ}$	adresa: D_{ADRESA}
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

Atribut

Doména

Schéma
relace

Tělo
relace

N-tice

- **Doména** - pojmenovaná množina skalárních hodnot téhož typu.
Př) Doména křestních jmen, doména příjmení
- **Skalární hodnota** - nejmenší sémantická jednotka dat, atomická (vnitřně nestrukturovaná).
Př) Josef – z domény křestních jmen; Novák – z domény příjmení
- **Složená doména** – doména složená z několika jednoduchých domén.
Př) složení domén křestních jmen a příjmení, hodnoty jsou dvojice, např. (Josef, Novák)
 - **Atribut relace** považujeme za sloupec tabulky,
 - **N-tici relace** považujeme za řádek tabulky.
 - **Doména** - Množina hodnot, kterých může atribut nabývat. Hodnoty mohou být pouze **skalární**.

Název **relační model** a **relační databáze** je odvozen od faktu, že relace je základním abstraktním pojmem modelu a jedinou strukturou databáze na logické úrovni.

Schéma relační databáze

Schématem relační databáze nazýváme **dvojici** (R, I), kde:

- $R = \{R_1, R_2, \dots, R_n\}$ je množina schémat relací,
- $I = \{I_1, I_2, \dots, I_m\}$ je množina integritních omezení.

Integrní omezení

Omezení ukládaných dat do databáze plynoucí z reality.

- **specifická**: pro konkrétní aplikaci (pole může nabývat určitých hodnot, může

nabývat pouze určitého počtu znaků, nesmí být NULL, ...)

- **obecná:** platí v **každé databázi** (primární a cizí klíč).

Kandidátní klíč

Atribut nebo **složený atribut** pro který platí:

- je **jednoznačný**,
- je **minimální** (nelze už redukovat).

Každá relace v teorii relačního modelu má alespoň jeden kandidátní klíč.

Primární klíč

Primární klíč je jeden z kandidátních klíčů. Primární klíč je základním prostředkem pro adresování n-tice v relačním modelu (řádek tabulky je jednoznačně identifikován primárním klíčem). **Žádné komponenta primárního klíče nesmí být prázdná NULL.**

Cizí klíč

Atribut nebo **složený atribut** relace **R2**, pro který platí:

- každá jeho hodnota je buď **zadaná**, nebo je každá **prázdná**,
- V FK je uložena hodnota, která je **shodná** s nějakou hodnotou primárního klíče **jiné relace R1**.

Relační algebra

Relační algebra je dvojice $RA = (R, O)$, kde:

- **R** je množina relací,
- **O** je množina operací (s relacemi)

Množina operací zahrnuje:

- **tradiční množinové operace:** sjednocení, průnik, rozdíl, kartézský součin,
- **speciální relační operace:** projekce, selekce (restrikce), spojení a dělení.

Tradiční množinové operace mají stejné výsledky jako u množin, lze ale provádět (až na kartézský součin) s relacemi, které mají stejné schéma (stejně záhlaví tabulky).

R1		
A	B	C
0	a	d
1	a	e
2	b	f

R2		
A	B	C
2	c	d
0	a	d
0	a	e

R1 union R2		
A	B	C
0	a	d
1	a	e
2	b	f
2	c	d
0	a	e

R1 intersect R2		
A	B	C
0	a	D

R1 minus R2		
A	B	C
1	a	e
2	b	f

R1 times R2

AR1	BR1	CR1	AR2	BR2	CR2
0	a	d	2	c	d
0	a	d	0	a	d
0	a	d	0	a	e
1	a	e	2	c	d
...

Projekce

Projekce je operace při které **vybíráme jen některé sloupce** z původní tabulky. Z relace **R** tak vytváříme relaci **R[X, Y, ..., Z]** se schématem **(X, Y, ..., Z)** a tělem obsahující patřičné (redukované) **n-tice** odpovídající novému schématu z původní relace **R**. V SQL tomu odpovídá zápis **SELECT name, surname FROM users**, kde users je tabulka obsahující např. (name, surname, title, email, phone, ...). Součástí operace projekce je i **odebrání duplicitních řádků**, které projekcí vzniknou (při SELECT tomu tak ale není).

R

A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	2

R [B,C,E]

B	C	E
a	x	1
b	y	2

Selekce (restrikce)

Restrikce je operace, při které je **zachováno původní schéma** realce (záhlaví tabulky), ale jsou vybrány pouze **některé n-tice** relace (řádky tabulky), které odpovídají určité podmínce. V SQL tomu odpovídá zápis **WHERE id = 42...** (s různými operátory pro porovnání).

R

A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	2

R where A=0

A	B	C	D	E
0	a	x	3	1
0	b	y	4	2

Spojení

Spojení je operace, při které je dochází ke **sloučení dvou schémat** relace (dvou záhlaví tabulek) tak, že schéma výsledné relace obsahuje **všechny atributy původních relací** a minimálně jeden atribut je mezi původními relacemi sdílen.

Podle tohoto **sdíleného atributu** je prováděno spojení, tak že jsou spojeny n-tice původních relací, které mají stejnou hodnotu tohoto sdíleného atributu. V SQL zapisujeme:

- **INNER JOIN companies ON names.id = companies.id ...** Vrací záznamy z levé tabulky které mají **odpovídající** záznam v pravé tabulce (tj. vynechává řádky, které na sebe nelze navázat).
- **LEFT JOIN companies ON names.id = companies.id ...** Vrací **všechny** záznamy z **levé tabulky**. K nim připojí odpovídající záznamy z pravé tabulky a pokud žádný odpovídající záznam neexistuje jsou sloupce odpovídající pravé tabulce **prázdné** (ve výsledku nemusí být všechny řádky pravé tabulky).
- **RIGHT JOIN companies ON names.id = companies.id ...** Vrací **všechny** záznamy z **pravé tabulky** a připojí k nim odpovídající záznamy z levé tabulky a pokud žádný odpovídající záznam neexistuje jsou sloupce odpovídající levé tabulce **prázdné** (ve výsledku nemusí být všechny řádky levé tabulky).
- **OUTER JOIN companies ON names.id = companies.id ...** Vrací **všechny** záznamy obou tabulek, pokud mezi nimi neexistuje spojení, jsou atributy patřičné tabulky ve výsledné prázdné.

R1

A	B	C
0	a	d
1	a	e
2	b	f

R2

C	D	E
e	1	0
d	1	1
d	0	1

R1 join R2

A	B	C	D	E
0	a	d	1	1
0	a	d	0	1
1	a	e	1	0

Jazyk SQL

Structured Query Language je jazyk pro dotazování **relačních databází**. SQL je standardizovaný, nicméně se některé příkazy mohou lišit napříč výrobci (jedná se ale o nestandardní části). Jazyk je **case insensitive**.

Definice dat

Tvorba databázových objektů, **tabulky, pohledy, indexy**.

CREATE TABLE

```
CREATE TABLE Persons (
    PersonID int GENERATED AS IDENTITY PRIMARY KEY,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

```
CREATE TABLE new_table_name AS
    SELECT column1, column2, ...
    FROM existing_table_name
    WHERE ....;
```

CREATE INDEX

```
CREATE (UNIQUE) INDEX index_name
ON table_name (column1, column2, ...);
```

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

CREATE VIEW

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

ALTER

```
ALTER TABLE table_name
ADD column_name datatype;
```

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype;
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

DROP

```
DROP TABLE table_name;
```

Manipulace s daty

Při manipulaci s daty jsou operandem bázové tabulky nebo pohledy, výsledkem tabulka.

SELECT

```
SELECT [ALL|DISTINCT] položka [[AS] alias_s1], ...
      FROM tabulkový_výraz [[AS] [alias_tab]], ...
      [WHERE podmínka]
      [GROUP BY jm_sloupce_z_FROM|číslo, ...]
      [HAVING podmínka]
      [ORDER BY jm_sloupce_z_SELECT|číslo [ASC|DESC], ...]

SELECT DISTINCT K.*
  FROM Klient K, Ucet U
 WHERE K.r_cislo=U.r_cislo
       AND U.pobocka='Jánská'

SELECT jmeno, r_cislo, COUNT(*) pocet, SUM(stav) celkem
  FROM Klient NATURAL JOIN Ucet
 GROUP BY r_cislo, jmeno

SELECT jmeno, r_cislo, SUM(stav) celkem
  FROM Klient NATURAL JOIN Ucet
 GROUP BY r_cislo, jmeno
 HAVING SUM(stav)>100000

SELECT K.jmeno, K.r_cislo, SUM(stav) celkem
  FROM Klient K, Ucet U
 WHERE K.r_cislo=U.r_cislo AND K.mesto<>'Brno'
 GROUP BY K.jmeno, K.r_cislo
 HAVING SUM(stav) > ALL
       (SELECT SUM(stav)
        FROM Klient K, Ucet U
        WHERE K.r_cislo=U.r_cislo AND K.mesto='Brno'
        GROUP BY K.r_cislo)
```

INSERT

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO table_name
```

```
VALUES (value1, value2, value3, ...);

INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger',
'4006', 'Norway');
```

UPDATE

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

DELETE

```
DELETE FROM table_name
WHERE condition;
```

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```

Pohled (View)

Virtuální DB struktura (v pohledech nejsou uložena data), může zprostředkovávat data z nula a více tabulek. Pracuje se s nimi jako s tabulkami. Mohou z tabulek vybírat jen určité řádky nebo sloupce, mohou obsahovat výrazy, mohou spojovat data z více tabulek, mohou odkazovat na další pohledy. Používá se pro zjednodušení práce (opakované dotazování). Vytváří se pomocí dotazu SELECT.

Kurzor (Cursor)

Prostředek, který umožňuje sekvenčně zpracovávat data - číst i modifikovat. Umožňuje např. upravovat jednotlivé řádky tabulky imperativním způsobem na místo deklarativního způsobu jako u dotazu UPDATE.

Uložená procedura (Stored procedure)

Uložená procedura (rutina) je **sada příkazů SQL**, které jsou uložené na databázovém serveru a vykonává se tak, že je zavolána prostřednictvím dotazu názvem, který jím byl přiřazen (je to určitá obdoba funkce).

Trigger

Trigger je uložený program, který se spustí automaticky jako reakce na určitou akci s danou tabulkou (před nebo po ní). Triggery nastavujeme na dotazy UPDATE, INSERT, DELETE.

DB transakce

Databázová transakce poskytuje mechanismus, který zajišťuje, že při manipulací s daty v databázi, bude databáze setrvávat v **konzistentním stavu**. Databázová transakce splňuje vlastnosti, které jsou známé pod akronymem **ACID**. Jedno příkazové operace nemusí být obaleny transakcí, databázové systémy k ním přistupují jako ke transakcím.

Atomicita (Atomicity)

Databázová transakce je již dále nedělitelná. Tzn. buď se provedou všechny její části (příkazy), nebo se neprovede žádný. Např. při vkládání dat do DB, která mají být uložena do více tabulek se musí vždy uložit všechny, nebo operace selže tak, že se neuloží žádná.

Konzistence (Consistency)

Transakce musí **zachovat konzistenci** databáze, tzn. **nesmí být porušeno** žádné ze specifických ani obecných **integritních omezení**.

Izolovanost (Isolation)

Operace prováděné v rámci nedokončené transakce **nemohou ovlivnit jiné probíhající transakce** (např. pokud dojde k rollback - jiná transakce nemůže použít data, která byla upravena touto transakcí, protože úpravy mohou být anulovány). Případně lze dovolit použití těchto dat, ale anulovat všechny transakce, které je používaly). Musí být zajištěno, že **transakce probíhají jedna po druhé, pokud manipulují se stejnými daty**.

Trvalost (Durability)

Poté co je transakce dokončena (commit), je zajištěno, že provedené změny budou mít **trvalý charakter**, a to i při **výpadku systému**.

Business transakce

Jako business transakce chápeme běžné operace, které provádí uživatelé v informačním systému. Např. **vytvoření faktury, nákup zboží v eshopu, vystavení dokladu, příjem zboží** atd. Vytvoření faktury může znamenat zadání nákupčího, data splatnosti, položek faktury a její odeslání emailem nebo vytisknutí. Tato business transakce je tvořena několika databázovými transakcemi, které v průběhu realizace business transakce zajišťují konzistenci v databázi. Např. **přidání každé položky faktury může představovat novou DB transakci** (stav faktury s 1 nebo 2 položkami je konzistentní, nekonzistentní by bylo, kdyby se nějaká položka uložila pouze částečně). Systémy, které se zabývají transakčním zpracováním se označují jako **On-Line Transaction Processing (OLTP)** systémy.

37. Webová uživatelská a aplikační rozhraní, správa sezení a autentizace.

Webová uživatelská rozhraní

Webová uživatelská rozhraní jsou založená na technologiích HTML, CSS, a JavaScript, se kterými dokáží pracovat webové prohlížeče.

HTML

Hypertext Markup Language je značkovací jazyk používaný pro tvorbu webových stránek, které jsou propojeny **hypertextovými odkazy**.

- MIME: **text/html**,
- koncovka: **.html, .htm**

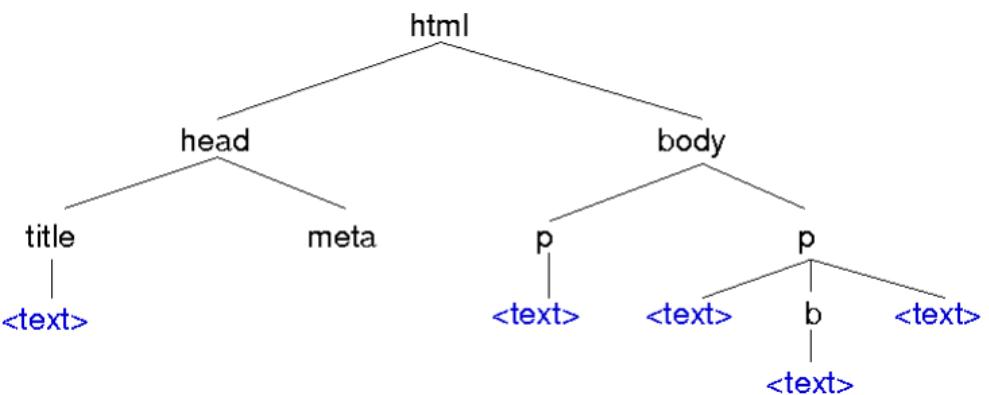
Základem HTML jsou **značky**, které vymezují **úseky** dokumentu. Obvykle jsou značky **párové** (`<div></div>`, `<p></p>`), ale existují i **nepárové** (prázdné - void, `
`, `<hr>`), tyto značky mohou mít pouze atributy, ale nemají obsah. Dále HTML dokument tvoří **komentáře/poznámky**, které nezpracovává prohlížeč (`<!-- Libovolný text -->`). Pro vkládání **speciálních znaků** (`<`, `>`, `&`, ...) používáme escape sekvence (`<`, `>`, `&`, ...). Používáním HTML značek vytváříme **stromovou strukturu** dokumentu.

Samotný html obsah bez stylování pomocí CSS není uživatelsky přívětivý (má dle

```

<!DOCTYPE html>
<html lang="cs">
  <head>
    ...
    ... hlavička ...
  </head>
  <body>
    ...
    ... tělo dokumentu ...
  </body>
</html>

```



mého názoru škaredý vzhled).

Základní ovládací prvky HTML

- **<input>**: slouží pro zadání vstupu uživatelem, který poté lze zaslat na server pomocí formuláře metodou POST nebo GET. Má různé typy (text, submit, phone, hidden, button, email, radio, ...)
- **<textarea></textarea>**: slouží pro zadání rozsáhlého textu, lze nastavit kolik má mít řádků a sloupců a může být zvětšovací.
- **<select><option></option></select>**: umožňuje uživateli výběr jedné z hodnot.
- **<button></button>**: tlačítko například pro odeslání formuláře.

Základní HTML prvky pro zobrazování obsahu

- **<nav></nav>**: element, který by měl obsahovat navigaci na stránce pro SEO,
- ****: očíslovaný seznam,
- ****: neočíslovaný seznam,
- **<table><tr><td></td></tr></table>**: tabulka (zvýraznění buněk pomocí **<th>**), lze nastavit spojování buněk,
- ****: odkaz na jinou stránku,
- **<p></p>**: odstavec,
- **<div></div>**: obecný blokový element,
- ****: obecný řádkový element.

Kaskádové styly CSS

Mechanismus (jazyk), který umožňuje návrhářům oddělit vzhled dokumentu od jeho struktury a obsahu. HTML elementům lze pomocí atributu **class** přidat různé způsoby zobrazení, element lze také stylovat pomocí elementu **style**, to ale není doporučeno. Pomocí kaskádových stylů lze:

- **stylovat text**: barvu, velikost, font, podtržení, kurzívou, tučný text, aj.
- **nastavovat odsazení elementů**,

- nastavovat pozici elementů,
- specifikovat barvu: pozadí, okraje, stínu,
- vytvářet animace,
- specifikovat různé zobrazení pro různě velké obrazovky,
- specifikovat zobrazení pro tisk.

JavaScript JS

JS je skriptovací objektově orientovaný jazyk, který dokáži interpretovat (snad) všechny webové prohlížeče a umožňuje tak vytvářet dynamické webové stránky. Lze pomocí něj manipulovat s **DOM** (Document Object Model), což je objektově orientovaná reprezentace HTML (a XML). Lze tak měnit vzhled elementů a přidávat nebo odebírat je bez znovunačtení stránky. Současně lze pomocí JS **asynchronně** (tj. bez toho aby se zaseklo uživatelské rozhraní) dotazovat server pomocí **AJAX** (Asynchronous JavaScript And XML) a umožnit tak například zapsání změn bez nutnosti načtení stránky, jinak řečeno umožňuje tvořit jedno stránkové aplikace. JS na HTML elementy napojujeme pomocí **event listeners**.

Applikační rozhraní

Applikační rozhraní nabízí způsob, jak spolu mohou **komunikovat** dvě (a více) aplikací, respektive jedna aplikace se může **dotazovat** na informace, které poskytuje ta druhá. Typickým příkladem využití webového API je když vytvářím aplikaci kde musí uživatel vyplnit adresu (např. nějaká dovážková služba), tak použiji API, které za prvé validuje adresu a umí na základě ulice vyhledat město, zemi, PSČ atd. Uživateli lze nabídnout tento výsledek dotazu na API a on se nemusí namáhat s vyplňováním a neudělá omylem chybu. Existuje několik způsobů, jak lze API programovat.

SOAP – Simple Object Access Protocol

Standardizovaný protokol, který specifikuje, strukturu dotazu i odpovědi na dotaz. Je nezávislý na HTML, lze jej použít i např. s SMTP protokolem. Každá zpráva je zabalena do **obálky (envelope)** a je tvořena **hlavičkou (header)**, která však není povinná a ve zprávě může chybět, a **tělem (body)** zprávy, které obsahuje dotaz, respektive odpověď na dotaz. Pro serializaci dat používá formát XML.

- **výhody:** jedná se o standardizovaný protokol (vhodnější pro veřejná api, např. vládní)
- **nevýhody:** objemnost přenášených dat, která je dána použitím serializačního formátu XML. S tím také souvisí náročné zpracování a validace příchozích dat.

REST – Representational State Transfer

REST není protokol, jedná se pouze o návrh architektury, který by měly splňovat systémy, které chtějí implementovat tak zvané RESTful aplikační rozhraní. Je závislý

na HTTP protokolu a využívá jeho metody a stavové kódy. Nad každým zdrojem jsou definovány CRUD operace a využití metod je následující:

- **C** zastává operaci **Create** – vytvoření objektu/objektů daného typu, operace se váže na HTTP dotaz typu **POST**,
- **R** zastává operaci **Read** – čtení objektu/objektů, operace se váže na HTTP dotaz typu **GET**,
- **U** zastává operaci **Update** – aktualizaci objektu/objektů, operace se váže na HTTP dotaz typu **PUT** nebo **PATCH**,
- **D** zastává operaci **Delete** – odstranění objektu/objektů, operace se váže na HTTP dotaz typu **DELETE**.

Jako serializační formát REST používá nejčastěji JSON, ale lze použít i XML či jiný. Zhodnocení REST:

- **výhody:** jednoduchost implementace a široká podpora různými knihovnami a frameworky. Integrace s protokolem HTTP a z toho plynoucí jednoduší zpracování dotazů a také flexibilita při výběru formátu přenášených dat.
- **nevýhody:** nejedná o standard. To způsobuje odlišnosti u jednotlivých implementací, ať už jde o použitý formát dat, nebo i způsob sestavování URL koncových bodů.

GraphQL – Graph Query Language

GraphQL také není protokol, jedná se o silně typovaný jazyk definující syntax, pomocí něhož lze vytvořit dotaz žádající přesně konkrétní data. Také není závislý na protokolu a kromě HTTP lze využít i s např. MQTT. Pro serializaci dat používá GQL výhradně formát JSON.

- **výhody:** Hlavní výhodou aplikativního rozhraní implementovaného pomocí GraphQL je, že server odesílá v odpovědi pouze ta data, o která si klient v dotazu žádá.
- **nevýhody:** malá rozšířenost, komplexní dotazy mohou představovat příliš velkou zátěž na server. Dotazy nelze cachovat, protože se mění.

Správa sezení

Posloupnost dotazů (a odpovědí na ně) jednoho uživatele od okamžiku navštívení stránky, ža po její opuštění. Řeší se přiřazením **unikátního identifikátoru** (Session ID) uživateli při prvním navštívení stránky. Sezení **neřeší autentizaci**, pouze rozlišuje dotazy jednotlivých uživatelů (např. uživatel si v rámci sezení může nějak upravit chování stránky, např. dark mode, a stránka se tak chová do konce sezení). Protože je protokol **HTTP bezstavový** (není zachována žádná informace mezi jednotlivými dotazy) musí klient Session ID pokaždé zasílat při dotazu. To lze realizovat zasíláním v rámci dotazu, např jako součást URL, což je dost nepraktické. Proto se identifikátor sezení **ukládá do Cookies**. Cookie je malý objem dat, který si server může uložit na klientovi a ten je poté zasílá při každém dotazu. Odcizení Session ID může být potenciálně nebezpečné.

Autentizace

Autentizace je proces ověření identity uživatele (nebo jiného objektu). Autentizace vůči webovým službám je typicky řešena nějakou **tajnou informací**, kterou může znát pouze uživatel, který se autentizuje (typicky uživatelské jméno a heslo).

Možnosti autentizace u webových aplikací:

- **Authorization hlavička protokolu HTTP:** Jedná se o mechanismus vestavěný pro **autentizaci** (i když název hlavičky tomu **neodpovídá**) do protokolu HTTP. **Vyžaduje použití HTTPS.** Používá se spíše při autentizaci vůči API, protože je nutné, aby každý dotaz obsahoval tuto hlavičku. Předávaná data mohou mít více formátů např. **Basic a Bearer** (údaje se předávají kódované v Base64). Při GET dotazech lze zadat údaje v url: <http://username:password@example.com/>
- **Autentizace pomocí HTML formuláře a Cookies:** Další možností autentizace je odeslat autentizační údaje v rámci POST metody na server, ten je zpracuje a vyhodnotí, vytvoří uživateli Cookie indikující, že je autentizován (Cookie je obvykle nečitelná - nějakým způsobem šifrovaná) a klient pak při každém dotazu tuto Cookie odesílá na server.
- **JSON Web Token (JWT):** Složen ze tří částí
 - Header (hlavička) – obsahující účel a použité algoritmy,
 - Payload (obsah) – JSON data obsahující informace o uživateli,
 - Signature (podpis) – informace pro ověření, že token nebyl podvržen nebo změněn cestou.

Tyto 3 části se spojí (xxxxx.yyyyy.zzzzz) a kódují se pomocí **base64**.

Většinou se posílají v hlavičce **Authorization: Bearer base64(xxxx.yyyyy.zzzzz)**.

Postup autentizace:

1. Klient kontaktuje **autentizační server a dodá autentizační údaje** (autentizační server může být server používané aplikace nebo nějaký jiný, např. Twitter).
 2. Autentizační server vygeneruje **podepsaný JWT token** a vrátí jej klientovi.
 3. Klient jej **zasílá při každém dotazu** na server.
- **OAuth:** OAuth řeší problém, kdy chce uživatel **umožnit aplikaci přístup k nějaké webové službě** s API, u které má účet, ale **nechce aplikaci sdělit své přihlašovací údaje** k této službě. (Příkladem je např. souhlas, že Draw.io může přistupovat ke Google Disk). Realizuje se tak, že daná aplikace **přesměruje** uživatele na **stránky webové služby**, kterou požaduje. Tam uživatel **potvrdí, že důvěruje** této aplikaci a daná webová služba vygeneruje tzv. **Access Token**, pomocí kterého poté provádí na API webové služby dotazy.
 - **Single Sign On (SSO):** autentizační mechanismus, který umožňuje uživateli přihlásit se pomocí jediného ID a hesla do několika souvisejících, ale nezávislých systémů. Využívá k tomu **adresářové služby** (LDAP, Active Directory) a protokoly jako **NTLM** a **Kerberos**. Uživateli se pak stačí přihlásit

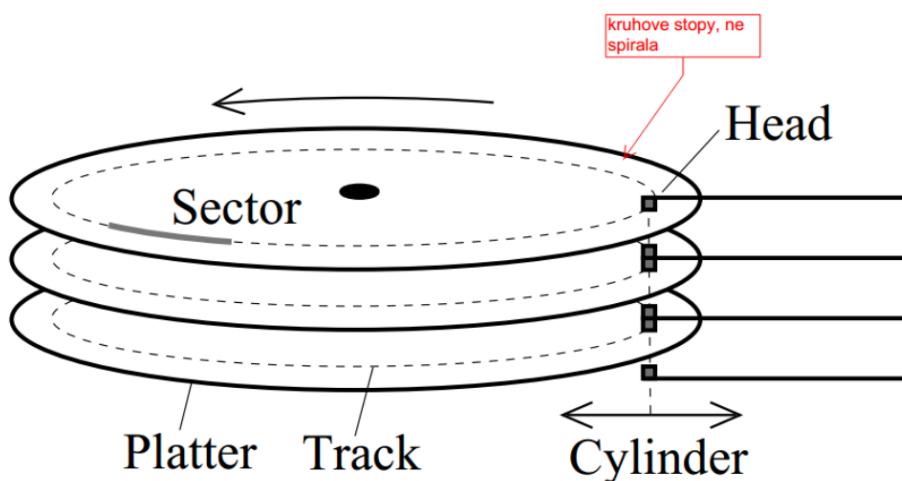
do adresářové služby (při přihlášení na PC např. do Windows) a pomocí jmenovaných protokolů se poté provádí autentizace vůči daným webovým službám za přihlášeného uživatele na PC.

38. Principy a struktury správy souborů a správy paměti.

Organizace a ukládání souborů

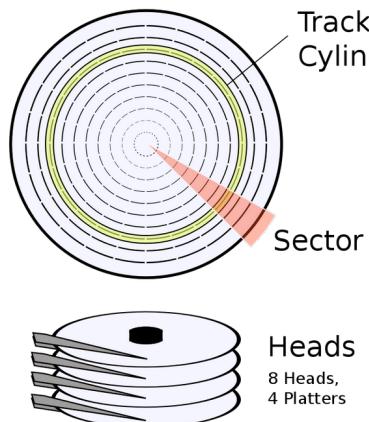
- **Soubor** - Základní organizační jednotka pro **uchovávání dat** na vnějších paměťových médiích.
- **Souborový systém** - Souhrn pravidel definujících chování a vlastnosti jednotlivých souborů a možnosti jejich další logické organizace. Také definuje způsob uložení požadovaných dat a informací k nim.
 - **FAT** - Univerzální mezi OS, je totiž skoro všude podporovaný.
 - **EXT (2,3,4)** - Pro linux.
 - **NTFS** - Pro Windows.

Uložení dat na HDD



Disk je rozdělen na sektory, což jsou nejmenší jednotky, které lze číst/zapsat (dříve 512B dnes 4096B). Adresace sektorů:

- **Cylindr-Hlava-Sektor (CHS)** - Adresace systémem souřadnic, které jsou specifikované číslem válce (C, válec je tvořen z track na více platters), hlavou (H) a sektorem (S), který se nachází na dráze (track). Vhodné pro menší disky, jelikož větší vyžadují **proměnný počet sektorů** na stopě v závislosti na čísle válce (vzdálenosti od středu) a adresace se děje v OS.

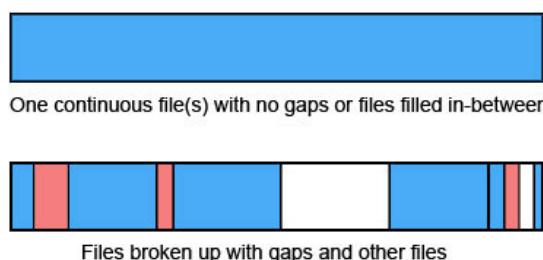


- **Logical Block Addressing (LBA)** - Adresace pomocí lineárního logického čísla sektoru (od 1 do N). Používá se v dnešní době, samotné vyhledání bloku na disku řeší nějaký obvod v disku, nikoliv OS.

Pojmy

- **Sektor disku** - Jeho nejmenší adresovatelná jednotka (má pevnou délku).
- **Alokační blok - 2^n sektorů** - Nejmenší jednotka diskového prostoru, se kterou dovoluje OS pracovat.
- **Fragmentace** - Data jsou uložena nesouvisle po částech. Zpomaluje operace prováděné s diskem. Disk lze defragmentovat - přeuspořádat uložená data, aby byla souvislá.
 - **Interní** - Fragmentace uvnitř alokovaných oblastí. Souborový systém **vyhradí pro soubor větší prostor** než je jeho velikost.
 - **Externí** - Fragmentace mezi alokovanými oblastmi, vzniká mazáním souborů - vytvořením nesouvislého uložení. Při nedostatku místa musí být soubor rozdělen (fragmentován) na části, které jsou uloženy do volných míst. Stejný problém vzniká při zvětšování souborů (za zvětšeným souborem může být uložen jiný a je nutné jej rozdělit). Může zapříčinit, že na disk soubor nelze uložit, i když je tam dostatek místa (soubor nelze ukládat po částech, nebo volná místa jsou natolik malé, že zde není možné současně uložit metadata a data - je nevhodně zvolená velikost alokačního bloku)

Example of File Fragmentation

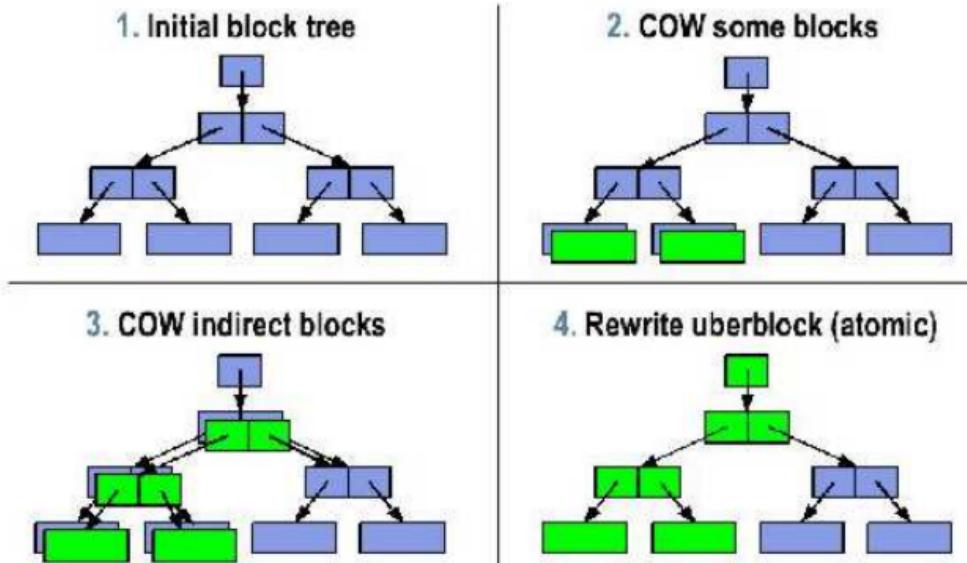


ComputerHope.com

- **Přístup na disk (čtení a zápis)** - Přístup se musí plánovat v závislosti na aktuální pozici hlav, jednotlivé požadavky lze přeuspořádat, tak aby se hlavy musely mezi zápisy pohybovat co nejméně. Různé postupy při pohybu hlav (od středu k okraji a zpět, pouze v mezikruží, kde jsou požadovaná data, operace prováděné pouze při pohybu v jednom směru, ...) Dokončení operace (včetně chyb) je oznamováno pomocí HW přerušení)
- **Logický disk** - Dělení fyzického disku na diskové oddíly (**partition**), se kterými je možné nezávisle manipulovat (jeví se jako více fyzických disků). Tabulka MBR (**Master Boot Record**) nebo novější GPT (**GUID Partition Table**) obsahuje informace o diskových oblastech.
- **Žurnálování** - stejně jako u DB zajišťuje zachování konzistence dat na disku

při zápisu. Na žurnál se zapisují operace, které budou prováděny, a po úspěšném zapsání se odstraní. Při chybě lze použít jednu ze dvou metod REDO a UNDO.

- **Copy-on-write** - nejprve zapisuje nová data a metadata na disk, poté je až zpřístupní. Zápis dat vychází z uložení v B+ stromech. Data se zapisuje postupně od listového uzlu, kde jsou uložena až po kořen (vnitřní uzly představují metadata)



Typické parametry disku

- kapacita do 20 TB,
- doba přístupu od nízkých jednotek ms,
- přenosová rychlosť v desítkách až stovkách MB/s

RAID (Redundant Array of Independent Disks):

Metoda pro zabezpečení dat proti selhání pevného disku.

- **RAID 0:** Disk striping - následné bloky dat jsou rozmištěny na více discích, což znamená vyšší výkonnost, ale žádnou redundanci → není to vlastně raid, nechrání před poruchou.
- **RAID 1:** Disk mirroring - data se ukládají na **2 disky**, velká redundance a výkonnost stejná jako u 1 disku.
- **RAID 2** - Data jsou na discích rozdělena po bitech a zabezpečena **Hammingovým kódem**. **Menší redundancy** dat než u RAID1.
- **RAID 3-5** - Používají paritu, dokáží se vyrovnat se **ztrátou 1 disku**.
- **RAID 6** - Používá 2 paritní bloky oproti RAID 5. dokáže se vyrovnat se **ztrátou 2 disků**.

Parita se používá se k jednoduché detekci chyb

- **Lichá** - Lichý počet jedniček.
- **Sudá** - Sudý počet jedniček.

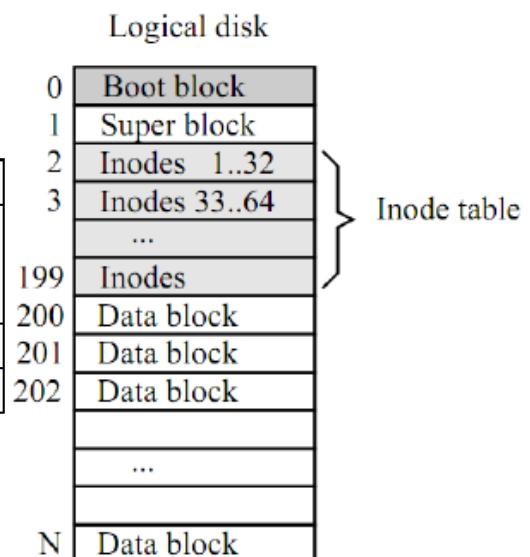
Uložení dat na SSD

SSD jsou organizovány do **stránek** (4096 B - 4 KiB) a ty do **bloků** typicky po 128 stránkách (tj. 512 KiB). Prázdné stránky lze přepisovat jednotlivě, pro přepis stránky (editace, mazání) je nutné načíst celý **blok** (512 KiB) do VP, provést požadované změny, z disku tento blok dat **smažat a zapsat jej znova**. Problém je řešen řadičem SSD, který může sám stánky přesouvat a uvolňovat celé bloky. SSD také může obsahovat nějaké stránky nad udávanou kapacitu, které jsou použity pro přepis.

Unixový systém souborů FS

Nejjednodušší rozdělení disku:

boot blok	pro zavedení systému při startu
super blok	informace o souborovém systému (typ, velikost, počet i-uzlů, volné místo, volné i-uzly, kořenový adresář, UUID, ...)
tabulka i-uzlů	tabulka s popisy souborů
datové bloky	data souborů a bloky pro nepřímé odkazy



vylepšení:

- disk je rozdělen do skupiny **bloků**, každá skupina má **své i-uzly** a datové bloky a volné bloky. Zajišťuje lepší lokalitu dat.
- super blok s informacemi o FS je ukládán vícekrát (při poškození tohoto bloku je problematické číst z disku obnovit data, proto redundancy)

i-uzel (inode)

Základní **datová struktura popisující soubor** v UNIX-ových souborových systémech. Obsahuje metadata, ve speciálních případech i data (např. symbolický odkaz - pokud je cesta krátká je uložena přímo v i-uzlu). Metadata tvoří:

- **stav i-uzlu** (alokovaný, volný, ..),
- **typ souboru** (obyčejný, adresář, **zařízení**, např. **dev/tty** - linux používá 2

abstrakce, a to soubory a procesy, simbolický odkaz, ...)

- **délka souboru v bajtech**,
- **mtime** = čas poslední modifikace dat,
- **atime** = čas posledního přístupu,
- **ctime** = čas poslední modifikace i-uzlu,
- **UID** = identifikace vlastníka (číslo),
- **GID** = identifikace skupiny (číslo),
- **přístupová práva** (číslo, například **0644** znamená **rw-r-r-**).

Samotná data jsou odkazována pomocí:

- **10 přímých** odkazů na data (rychlé, ale jen pro malé soubory),
- **1 nepřímý odkaz první úrovně**,
- **1 nepřímý odkaz druhé úrovně** (tyto bloky obsahují pouze odkazy na bloky první úrovně),
- **1 nepřímý odkaz třetí úrovně** (tyto bloky obsahují pouze odkazy na bloky druhé úrovně),

Na základě **velikosti jednotlivých bloků** a **velikosti odkazů na bloky** lze poté vypočítat maximální velikost souboru v FS:

$$10 * D + N * D + N^2 * D + N^3 * D,$$

kde:

- $N = D/M$ je počet odkazů v bloku, je-li M velikost odkazu v bajtech (běžně 4B),
- D je velikost bloku v bajtech (běžně 4096B).

Velikost souboru je také ovlivněna OS (např. na 32 bit je největší reprezentovatelné číslo $2^{32}-1$, což odpovídá 4 GiB). **i-uzel NEOBSAHUJE název souboru**.

Jiné způsoby organizace souborů

Organizace souborů a popis uložení je implementován tak, aby byla **minimalizována režie** při práci s ním a **bylo snadné**:

- **průchod souboru**, což je spojené s rychlostí **vyhledání následujícího bloku**,
- **přesun v souboru** (seek), což je spojené s **rychlostí vyhledání prvního a určitého bloku** souboru.
- **zvětšování a zmenšování souboru**, což je spojené s **přidáním bloků** (alokací prostoru) a **odebráním bloků** (dealokací prostoru).

Kontinuální uložení

Jednoduchá Data jsou uložena na disku jako jedna **spojitá posloupnost**.

Problematické je **zvětšování souborů**, pokud se za souborem nachází další.

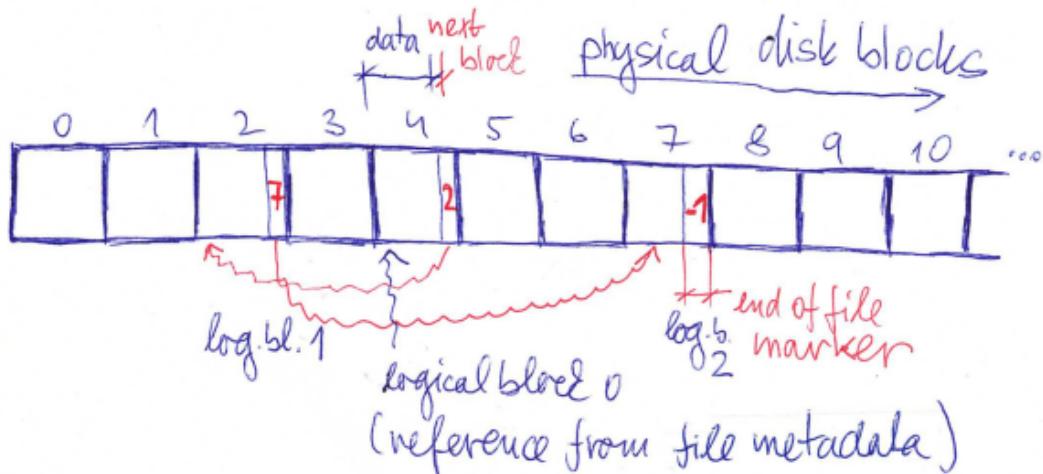
Problém ukládání nových souborů při **externí fragmentaci**.

Zřetězené seznamy bloků

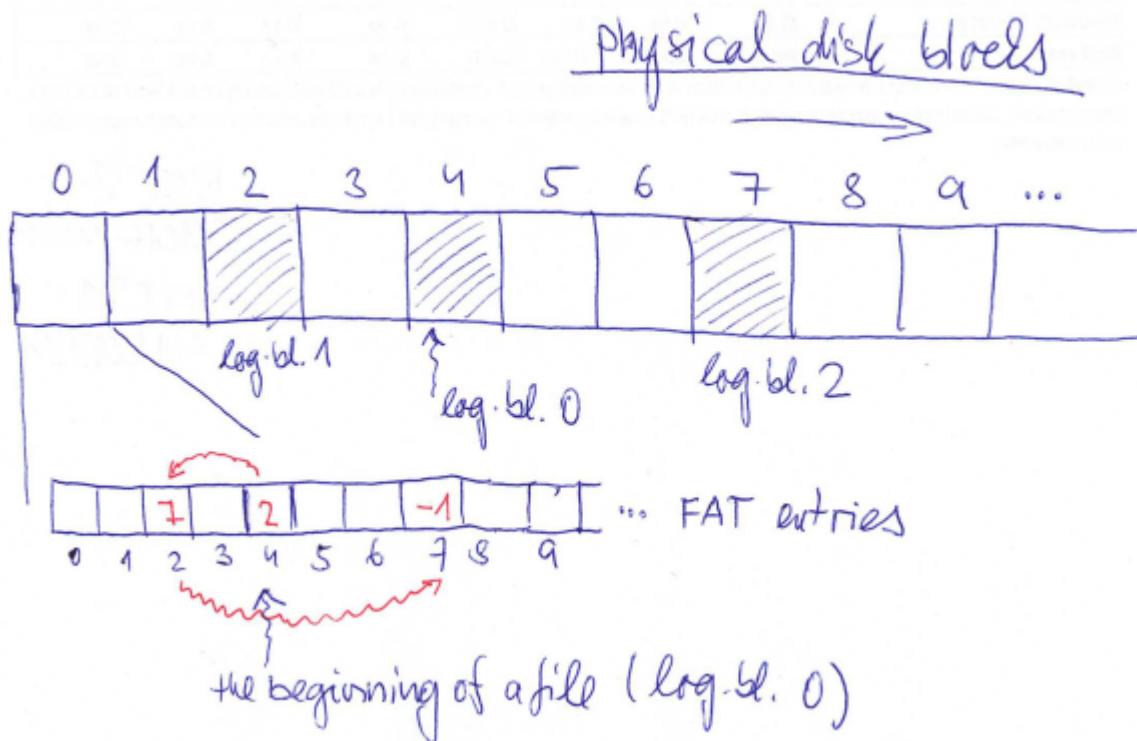
Každý datový blok obsahuje kromě dat **odkaz na další blok** (nebo příznak konce souboru). Problematický je **náhodný přístup** do určité části souboru, vždy se musí

procházet sekvenčně od začátku, stejný problém je při zápisu. Chyba v provázání kdekoliv na disku vede na ztrátu zbytku souboru. Řeší ale problém s fragmentací (bloku lze ukládat libovolně nespojité na disku).

File Allocation Table (FAT)



Podobný princip zřetězeným seznamům bloků. Tentokrát ale **není odkaz** na následující blok uložen **na konci datového bloku**, ale ve speciální **tabulce** (FAT - ta je uložena na začátku disku a pro jistotu 2x). Buňka tabulky obsahuje **odkaz** (index) **na jinou buňku** v tabulce a **odkaz** (index) **na blok s daty** na disku. Částečně řeší problém s náhodným přístupem, není nutné načítat do paměti blok po bloku (pro nalezení odkazu na další), ale samotný průchod ve FAT je **pořad sekvenční** (bude ale rychlejší).

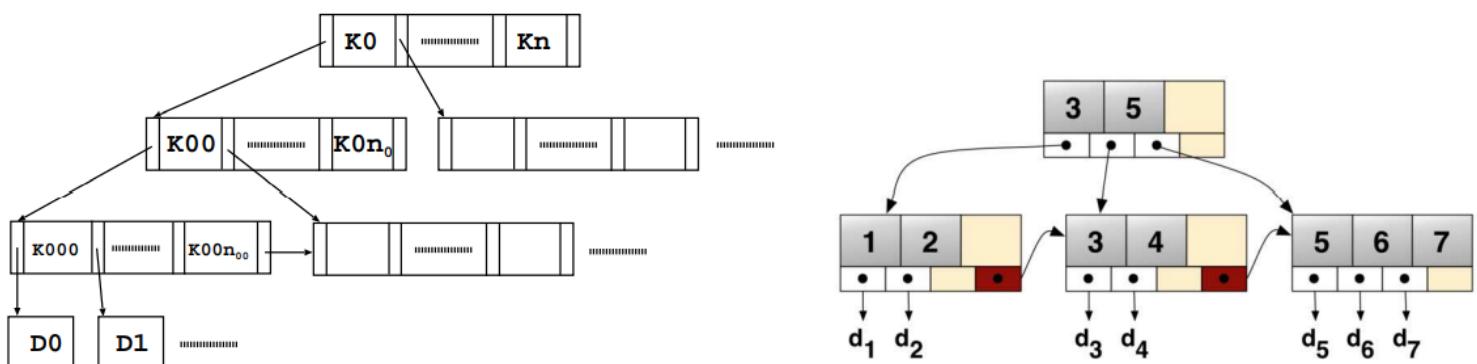


B+ stromy

Dnes nejběžnější způsob ukládání souborů na disk (s případnými modifikacemi).

Řeší problém sekvenčního i náhodného přístupu k datům souboru. Efektivnost B+ stromů je založena na **logaritmické složitosti** vyhledávání ve stromě. Struktura stromu:

- **vnitřní uzel**: je tvořen k klíci (identifikátory bloků - jejich sekvenční číslo) a $k+1$ odkazy na bloky nižší úrovně (tedy i listové). Klíče jsou v uzlu uspořádané **od nejmenšího k největšímu** (obecně mezi nimi musí existovat nějaká relace uspořádání). Po sobě následující klíče vymezují intervaly klíčů, které jsou získatelné, pokud bude následován odkaz mezi těmito klíči. (na začátku uzlu si lze představit imaginární klíč jehož hodnota je rovna min a na konci klíč s hodnotou max+1). Tyto **intervaly jsou zleva uzavřené a zprava otevřené**, tzn. že podmínka jestli je klíč v daném intervalu vypadá následovně: **if (key_i <= key_hledany < key_i+1)** tak následuj daný odkaz, jinak posuň index porovnávaných klíčů. Protože jsou klíče seřazeny, šlo by použít i vyhledávání založené na půlení intervalů.
- **listový uzel**: mají stejnou strukturu, odkazy ale vedou přímo na datové bloky (data souboru). **Poslední odkaz listového uzlu odkazuje na následující listový uzel**, což řeší sekvenční průchod souboru. Posloupnost hodnot klíčů na listové úrovni ale **nemusí být spojitá** (fragmentace), musí se zde nacházet **všechny klíče** (sekvenční čísla bloků souborů). Výběr odkazu na datový blok pak proběhne tak, že **porovnáváme** hodnotu hledaného klíče a hodnoty klíčů uložených v listovém uzlu na **rovnost**, pokud jsou si rovny, použijeme **odkaz na levo** od klíče, na kterém došlo ke shodě: **if (key_i == key_hledany) pak použij link_i**, jinak porovnávej s dalším klíčem.



B+ strom je **výškově vyvážený** a dále pro něj platí:

- **Limity zaplnění**: pro uzly s m odkazy (tedy klíči key_1 až key_{m-1})
 - sólo kořen 1 až $m-1$,
 - kořen 2 až m ,
 - vnitřní uzel $\lceil m/2 \rceil$ až m ,
 - list $\lceil m/2 \rceil - 1$ až $m-1$.
- **Vkládání**: vkládá se na **listové úrovni** při **přeplnění** se uzel **rozštěpí** a případně se rozštěpí i uzly ve vyšších úrovních až nakonec se může rozštěpit kořen, což vede na přidání nového kořene a **zvýšení stromu** o jedna.

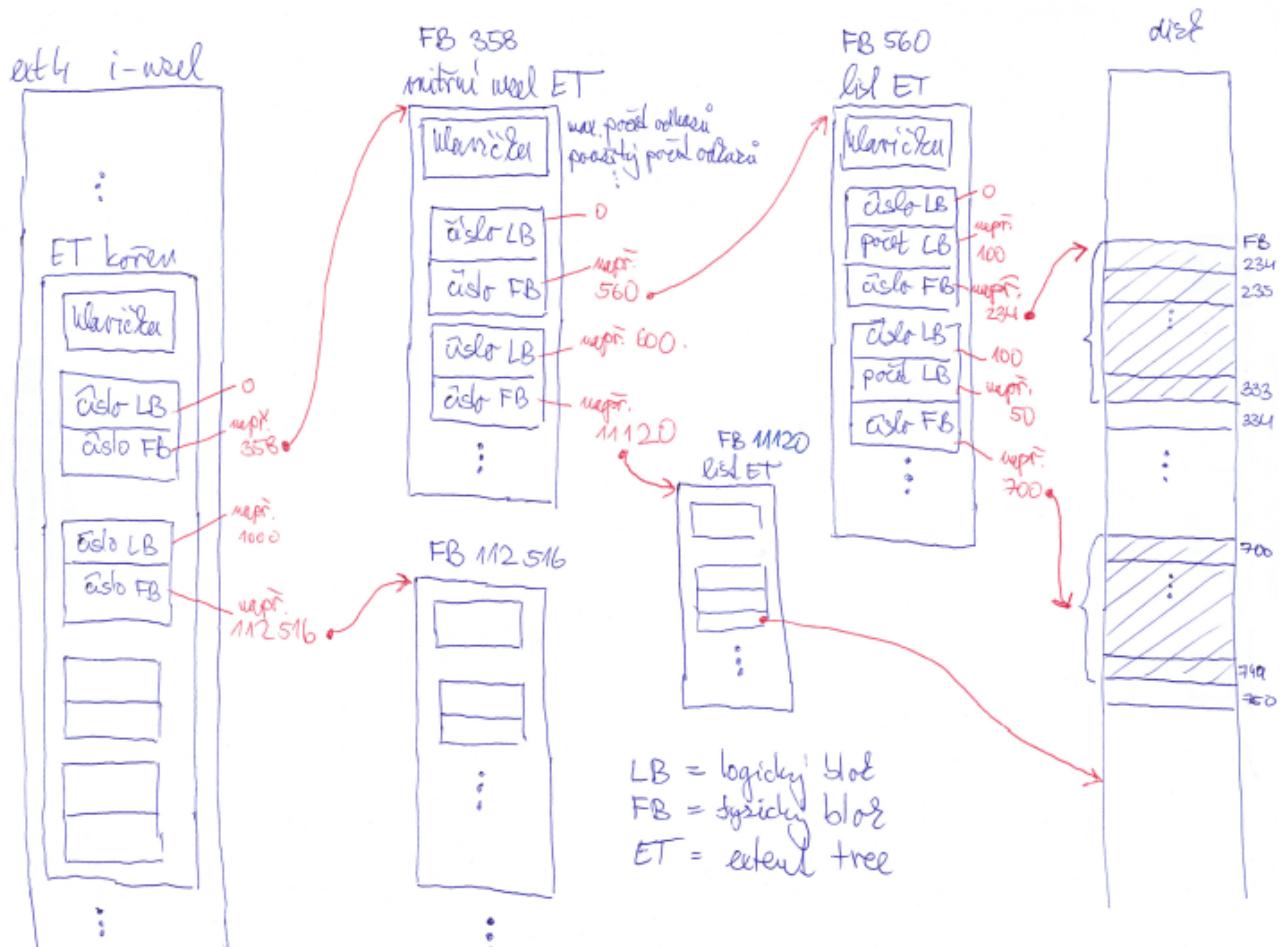
- **Rušení:** ruší se od listové úrovně a pokud je **porušena minimální naplněnost**, buď k **přerozdělení** odkazů **mezi sourozenci** (pokud je to možné), nebo ke **sloučení** sousedních uzlů, a případně ke sloučení uzlů ve vyšších úrovních, až nakonec může dojít ke sloučení druhé úrovně a zrušení původního kořene. Výška stromu se tak **sníží** o jedna.

Extenty

Extenty odkazují na proměnný počet bloku, které jsou **za sebou** uloženy **logicky** (tj. v souboru) a také **fyzicky** (tj. na disku). Zrychluje práci s velkými soubory a snižuje množství metadat. Extenty se **kombinují s B+** stromy, **nelze** je ale použít v **Unixovém stromu**, protože zde není mechanismus, jak ukládat informaci o jejich proměnné délce.

Strom extentů

Analogie B+ stromu **bez vyvažování a bez zřetězení listů**. Má omezený počet úrovní **max 5 úrovní**. Kořen v je umístěn **do i-uzlu** a má **max 4 odkazy** (stačí pro malé soubory nebo při málo zaplněném disku), vnitřní uzly mohou mít **i více**. Vnitřní **indexové** uzly se vytváří až při naplnění odkazů v kořeni, poté se ale do nově vytvořeného indexového uzlu přesunou všechny a do kořene se umístí odkaz na tento indexový uzel. Indexové uzly obsahují hlavičku, ve které je **aktuální počet odkazů** a **maximální počet odkazů** a dvojice **číslo logického boku** a **číslo fyzického bloku** (ten může být již datový nebo opět indexový).



NTFS (New Technology File System)

Souborový systém vyvinut pro Windows NT. Využívá **Master File Table (MFT)**, ve které je aspoň jeden řádek pro každý soubor. Obsah souboru poté může být:

- uložen **přímo** v MFT daného souboru,
- odkazován z MFT pomocí **extentů**,
- odkazován z MFT pomocí **B+ stromu**, který je tvořen jinými MFT záznamy odkazovanými z **primárního MFT** záznamu.

Organizace volného prostoru

- bitové pole** (bitová mapa) s jedním **bitem pro každý blok**, vyhledávání pomocí bitového maskování.
- zřetězení** volných bloků,
- zřetězení odkazů** ve FAT, které odkazují na volné bloky,
- B+ stromem** (adresace velikostí nebo offsetem),
- Po **extentech**.

Deduplikace

Zamezení opětovnému ukládání stejných dat (na úrovni souboru, extentů, bloků,

bytů). Může (výrazně) **ušetřit místo** např. na mail serverech (stejná příloha u několika emailů), git repozitářích (malé změny v souborech). Lze implementovat při **zápisu** nebo **dodatečně**, využívá se **kryptografického hashování** pro hledání shody (a poté případného porovnání). Problém nastává při změně jednoho ze stejných souborů, nutno rozdvojit atd. Při **malé duplikaci** může způsobovat navýšení spotřeby CPU času i místa na disku.

Typy souborů

-	obyčejný soubor
d	adresář
b	blokový speciální soubor
c	znakový speciální soubor
l	symbolický odkaz (symlink)
p	pojmenovaná roura
s	socket

- **Adresáře** obsahuje dvojice **jméno souboru** (dříve 14 znaků, dnes až 255, nemůže obsahovat / a \0) a **číslo souboru** (typicky se jedná o **číslo i-uzlu**, případně číslo pro hledání v B+ stromu). Adresář vždy odkazuje odkazy na sebe (.) a rodiče (...). Implementace pomocí seznamů, B+ stromů, hash table.
- **Symbolický odkaz** obsahuje jako data jméno odkazovaného souboru, při otevření se vícekrát musí zpracovávat cesta. Po smazání odkazovaného souboru **symlink** zůstává ale je neplatný (jeho otevření způsobí chybu). Cykly se řeší omezení úrovně zanoření, cesta a jméno odkazovaného uzlu může být uložena přímo v i-uzlu.
- **Terminály** jsou fyzická nebo virtuální rozhraní umožňující primárně textový výstup, editaci na vstupním řádku a speciální znaky. V UNIX jsou reprezentovány soubory (/dev/tty, /dev/tty1, ...). Více režimů zpracování znaků (**raw**, **cbreak**, **cooked**).
- **Roury** (pipes) umožňují **komunikaci mezi procesy**, mají **dva deskriptory** (čtecí a zápisový) V terminálu se vytváří pomocí znaku |. Jsou implementovány pomocí kruhového bufferu s omezenou kapacitou. Dělíme je na **pojmenované roury** a **nepojmenované roury**.
- **Sockety** Umožňují síťovou i lokální komunikaci mezi procesy (pojmenované a uložené v FS). Komunikace může být **blokující** i **neblokující**.

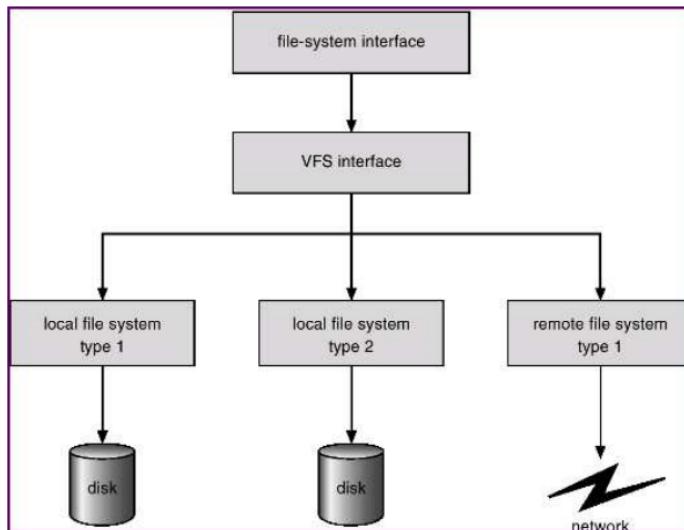
Montování disků

Nový disk (a jiné zařízení) lze namontovat do libovolného adresáře již existujícího adresářového stromu. Dnes většina OS automaticky montuje disky při připojení do

PC.

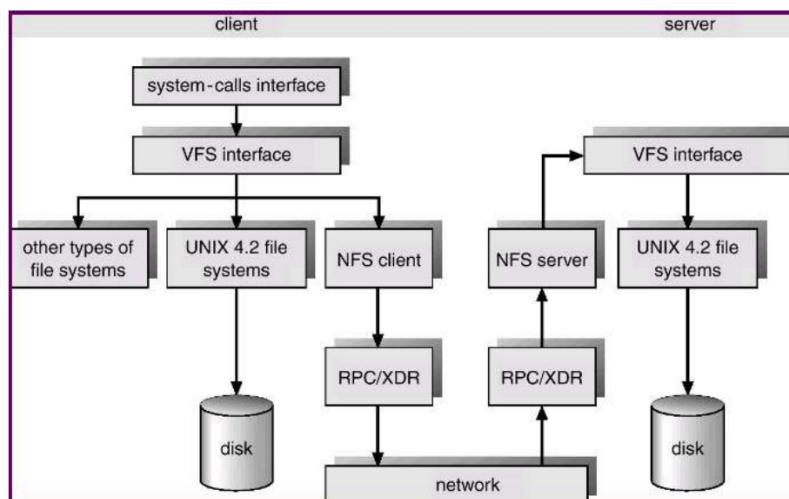
Virtual File System (VFS)

Vytváří **jednotné rozhraní** pro práci s **různými souborovými systémy**, odděluje vyšší vrstvy OS od konkrétní implementace jednotlivých souborových operací na jednotlivých souborových systémech (FAT, ext4, NFS, ...).



Network File System (NFS)

Zpřístupňuje soubory uložené na vzdálených systémech, zapojuje se do **VFS** a práce s ním je pak pro OS i uživatele stejná jako se soubory na disku.



Spooling (simultaneous peripheral operations on-line)

Umožňuje **zrychlení** u pomalých **výstupních zařízení**. Výstup je proveden do **souboru** a proces, který ho zadal (nejčastěji se jedná o **tisk**), může pokračovat. Vytvořený soubor je uložen do **fronty požadavků** na výstupní zařízení (tiskárnu) a čeká až na něj přijde řada.

Přístupová práva

Různá přístupová práva pro **vlastníka, skupinu a ostatní**.

např. **-rwx---r--** (číselně 0704), první znak znamená typ souboru (zde obyčejný soubor)

obyčejné soubory	
r	právo číst obsah souboru
w	právo zapisovat do souboru
x	právo spustit soubor jako program
adresáře	
r	právo číst obsah (ls adresář)
w	právo zapisovat = vytváření a rušení souborů
x	právo přistupovat k souborům v adresáři (cd adresář, ls -l adresář/soubor)

- vlastník: čtení, zápis, provedení,
- skupina: nemá žádná práva,
- ostatní: pouze čtení.

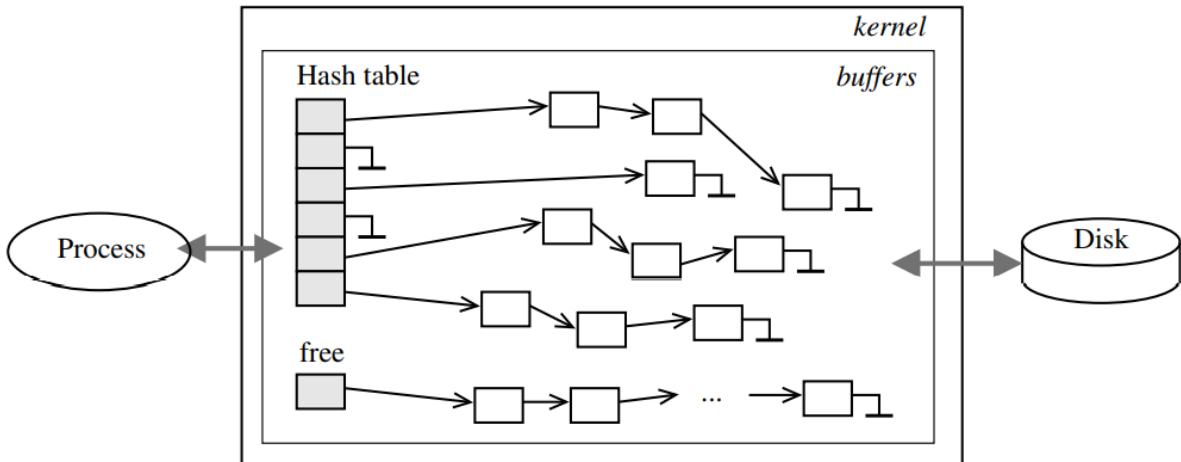
Sticky bit (t) je příznak, který nedovoluje rušit a přejmenovávat cizí soubory v adresáři, i když mají všichni právo zápisu (-rwxrwxrwxt).

Práva pro procesy

- **UID** - ID uživatele co ho spustil.
- **EUID** - Efektivní UID - pro kontrolu přístupových práv.
- **GID** - ID skupiny, ve které je uživatel co proces spustil.
- **EGID** - Efektivní GID - Podobně jako EUID.
- **SUID/SGUID** - Pro propůjčení práv vlastníka uživateli.

Práce se soubory vstup/výstup/mazání

Pro urychlení práce se soubory se používají vyrovnávací paměti (VP), které minimalizují operace s pomalými periferiemi (HDD, SSD, terminál, ...). Dílčí VP mívají **velikost datového bloku** nebo skupiny a jsou sdružovány do kolekcí pevné nebo proměnné délky. Možná implementace pomocí **hash table**.



Čtení

Postup při prvním čtení (soubor ještě není ve VP):

1. Přidělení VP a načtení bloku (prvního nebo požadovaného)
2. Kopie požadovaných dat z VP do adresového prostoru procesu (RAM → RAM).

Dokud jsou data v RAM a dochází pouze k bodu **2**, jakmile čtením proces **překročí do jiného datového** bloku dochází k bodu **1 i 2** (pokud ještě není ve VP).

read()

1. Kontrola platnosti **fd**.
2. Provedení kroků 1 a 2 popsaných výše dle potřeby.
3. Návratovou hodnotou je **počet doopravdy přečtených** bytů nebo -1 při chybě a nastavení **errno**.

Zápis

Postup:

1. Přidělení VP a načtení bloku souboru do VP (pokud se nevytváří nový nebo zcela nepřepisuje).
2. Zápis dat do VP z adresového prostoru procesu (RAM → RAM), **nastavení příznaku modifikace (dirty bit)**.
3. **Zpožděný zápis na disk, nuluje příznak** (např. při uvolnění místa z RAM, ...).

write()

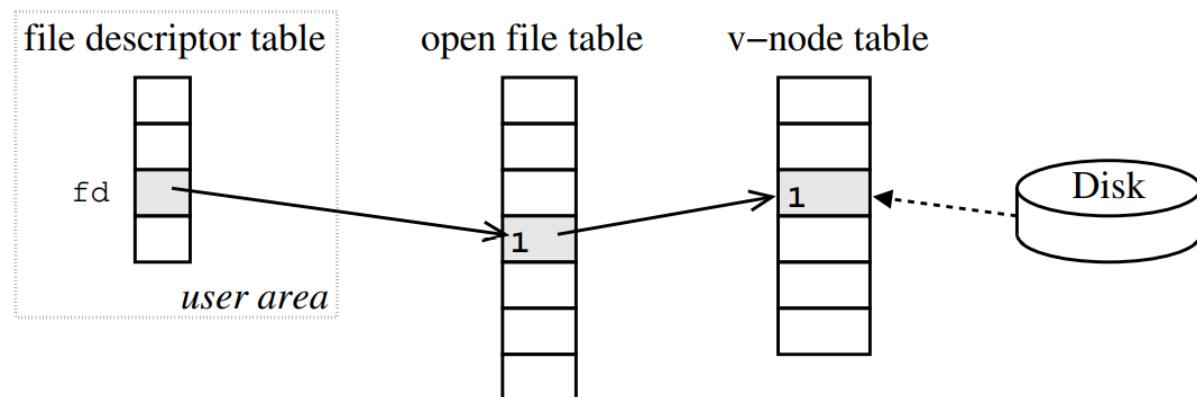
1. Kontrola platnosti **fd**.
2. Provedení kroků 1 a 2 popsaných výše dle potřeby.
3. Před zápisem dojde ke **kontrole dostupnosti dostatečného prostoru** na disk.
4. Návratovou hodnotou je **počet doopravdy zapsaných** bytů nebo -1 při chybě a nastavení **errno**.

Otevření

Nejdříve (v průběhu bodu 1) se provádí kontrola na přístupová práva.

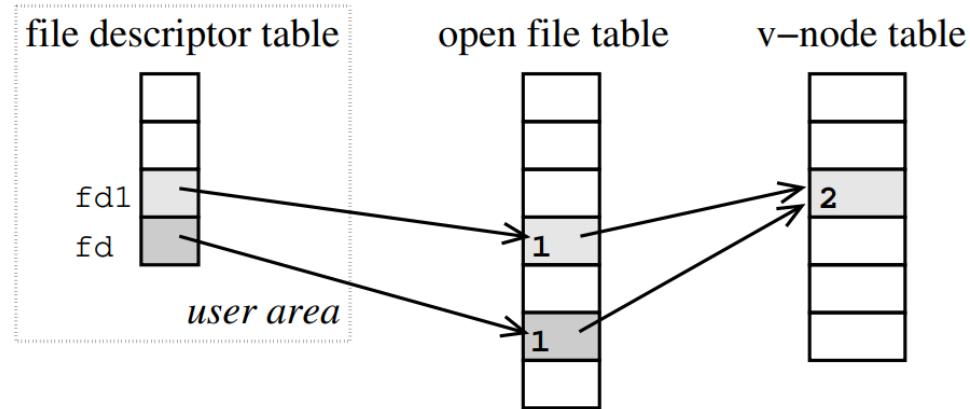
Soubor ještě **nebyl otevřen**:

1. **hledání i-uzlu** souboru: na základě cesty a jména souboru se postupně načítají **i-uzly adresářů na cestě** a datové bloky s obsahem adresářů, až je nakonec vyhledán požadovaný soubor a jeho i-uzel. Některé (časté) i-uzly mohou být už ve VP
2. V **systémové tabulce aktivních i-uzlů (inode table)** se vyhradí nová položka, do které se načte z VP **i-uzel** a stává se z něj tak **v-uzel**, což je jeho **rozšířená kopie**.
3. V **systémové tabulce otevřených souborů (open file table)** vyhradí novou položku a naplní ji:
 - a. **odkazem** na položku tabulky **v-uzlů**,
 - b. **režimem** otevření (čtení, zápis - smaže původní obsah, čtení i zápis),
 - c. **pozici** v souboru (0),
 - d. **čítačem počtu referencí** na tuto položku (1), počet referencí roste pouze při **fork** nebo **dup**, kdy dojde k **duplikaci fd** pro nový proces, procesy si mohou měnit pozici v souboru - nebezpečné...
4. V **poli deskriptorů souborů** (každý proces má vlastní, obsahují informace o **fd**) se vyhradí položka (**fd - int, index do pole**) a naplní se odkazem na položku v tabulce otevřených souborů.
5. Vrácení odkazu na **fd** nebo chyba.



Otevření pro čtení **již otevřeného** souboru (např. jiný proces nebo i ten samý):

1. **Vyhledání čísla i-uzlu**, jako při novém otevření.
2. **Vyhledání v-uzlu** v systémové tabulce (provádí se i při prvním otevření, ale při tom vyhledání selže, tabulka musí být upravena vyhledávání).
3. **Vyhrazení nového záznamu** v **systémové tabulce otevřených souborů** (může jít o jiný způsob otevření a také každé otevření může vyžadovat jinou pozici v souboru, odkaz v této tabulce se sdílí **pouze po fork**) s odkazem na v-uzel a **zvýšení počítadla odkazů v-uzlu**.
4. Vytvoření nového **fd**.
5. Vrácení **fd** nebo chyba.



Přímý přístup

Pomocí lseek, postup:

1. Kontrola platnosti **fd**,
2. nastavení **pozice fd jako offset bytů**,
3. Návratovou hodnotou je **výsledná pozice** nebo -1 při chybě

Při seek za konec souboru a zapsáním vzniká řídký soubor (uprostřed má prázdný prostor, který není uložen na disku), offset může být záporný, nelze ale nastavit pozici před začátek souboru.

Zavření souboru

Zavření souboru **nezpůsobí uložení obsahu jeho VP na disk**. Postup:

1. Kontrola platnosti **fd**.
2. Uvolnění **fd** z tabulky fd, snížení odkazů na záznam v **tabulce otevřených souborů** (open file table).
3. Pokud klesne počet referencí na záznam v tabulce otevřených souborů je záznam odebrán a snížen počet referencí na v-uzel v **tabulce otevřených i-uzlů**.
4. Pokud klesne počet referencí na v-uzel na 0, z v-uzlu se **okopíruje část, která tvoří i-uzel do VP** a v-uzel se uvolní.
5. 0 jako úspěch -1 neúspěch.

Mazání/rušení souboru

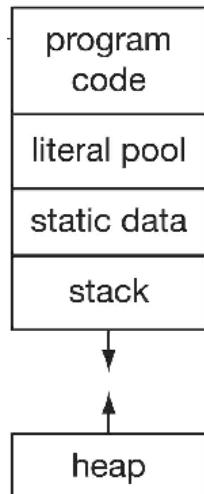
1. **Vyhodnocení cesty** a existence souboru, **kontrola přístupových práv**.
2. **Odstranění pevného odkazu** na i-uzel, v daném adresáři (vyžaduje práva pro zápis do adresáře).
3. **Zmenšení počtu odkazů (jmén)** na i-uzel (symlink v tomto počtu nejsou zahrnuti)
4. Pokud počet jmen **klesne na 0**, může být i-uzel a všechna jeho **data uvolněna**, to se provede, až soubor přestanou všichni používat (bude všemi uzavřen). Tzn. soubor jde smazat vždy (ne jak na windows, kdy kvůli tomu je potřeba pomalu restartovat PC...), to platí i pro spustitelné soubory (program

dále poběží).

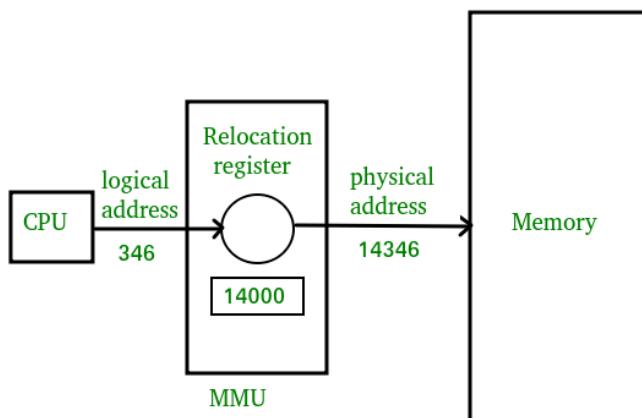
Správa paměti

Rozlišujeme:

- **Logický adresový prostor (LAP):** jedná se o virtuální adresový prostor, se kterým pracuje procesor při provádění kódu, každý proces i jádro jej **má svůj** a jsou **stejné** (více stejných logických adres je mapováno na různé fyzické adresy v paměti.). Pro úsek paměti se používá označení **stránka** (page).



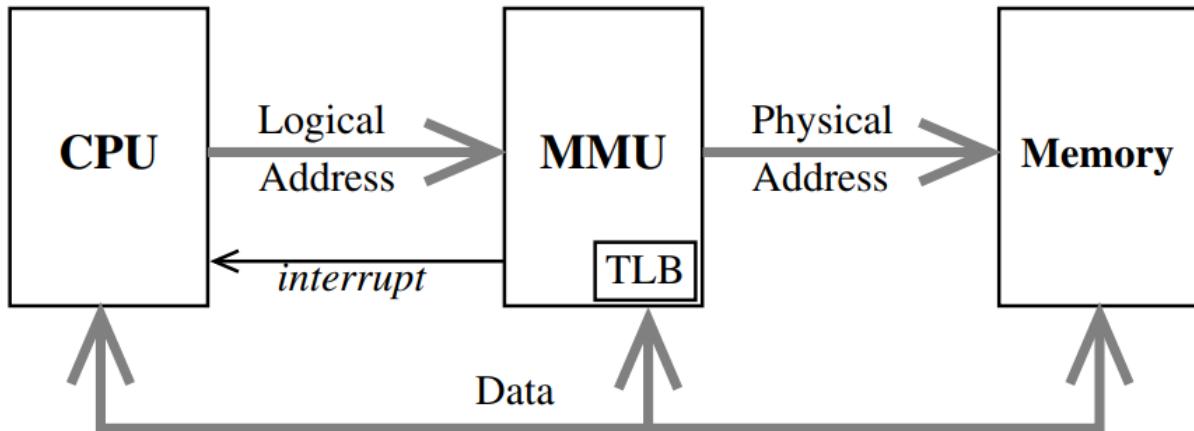
- **Fyzický adresový prostor (FAP):** pouze **jeden pro celé PC** (společný pro všechny procesy i jádro), jedná se o adresový prostor **fyzických adres v paměti** (RAM). Pro úsek paměti se používá označení **rámec** (frame).



Memory management unit (MMU)

HW jednotka, která zajišťuje překlad logických adres na adresy fyzické, běžně je součástí čipu (na obr. ilustrativně mimo). Pro překlad MMU využívá speciálních registrů a i hlavní paměti. Pro urychlení překladu používá různé VP, např.

Translation Lookaside Buffer (TLB).



Přidělování paměti

Přidělování **FAP** pro zmapování do **LAP** se používají úseky paměti o velikosti, které jsou konzistentní se způsobem překladu **LAP** na **FAP**, který je podporován v HW daného systému, což jsou:

- **spojité bloky**,
- **segmenty**,
- **stránky**,
- kombinací přístupů.

Při programování (např. v C malloc) se přidělují již namapované úseky **LAP**.

Contiguous Memory Allocation

Procesům jsou přidělovány **spojité bloky** paměti o určité velikosti. Pokud není aktuálně volný **dostatečně velký** spojitý úsek paměti, proces buď musí čekat, nebo musí dojít k **odložení** celé paměti jiného procesu na HDD.

- **výhody**: velmi jednoduchá implementace v HW i SW.
- **nevýhody**: výrazný projev **externí fragmentace**, a to z důvodu přidělování a uvolňování paměti různé velikosti. Vznikají úseky, které mohou být příliš malé, na to aby byly využity. Dalším problémem je **zvětšování přiděleného prostoru** pokud již za úsek aktuálně přidělené paměti není místo, je nutné celý již přidělený úsek přesunout do dostatečně velké oblasti a poté až přidělenou paměť zvětšit. Lze použít nějakou **strategii** pro přidělování paměti (first fit, best fit, worst fit), to ale ubírá na jednoduchosti implementace.

Přidělování paměti po segmentech

LAP je rozdělen na **několik segmentů**. Segmenty přiděluje různým částem procesu (zásobník, hromada, nebo i jednotlivým procedurám, ...) překladač nebo programátor. U segmentů může být dále specifikován způsob přístupu - čtení/zápis (bezpečnější, kód lze umístit do read only segmentu). **Logická adresa** je tvořena z **čísla segmentu a posunu v něm**, každý segment má číslo a velikost.

- **výhody**: poměrně jednoduchá implementace, zlepšuje externí fragmentaci, ale problém vlivem proměnné velikosti segmentů přetravává.

- **nevýhody:** segmentování programu je řízeno při tvorbě programu, což může vést na **problémy při implementaci**. Stále neřeší **problém s externí fragmentací**, příliš velké segmenty nemusí být možné namapovat do **FAP**, nutné odkládat větší úseky paměti na HDD.

Stránkování (přidělování paměti po rámcích)

LAP je rozdělen na **stránky** (pages) o **pevné velikosti** (každá stránka je stejně velká, obvykle 4096 B, což odpovídá velikosti bloku na disku), **FAP** je rozdělen na **rámce**, které jsou ale **stejně velké jako stránky**. Mapuje se vždy **jedna stránka na jeden rámec** (nelze aby se jedna stránka mapovala přes dva rámce) a **mapování je odstíněno** od implementace programu a běžícího procesu. Pro zvýšení efektivity přístupu do paměti je snaha o **mapování spojitéch posloupností rámci**, pokud je to možné.

- **výhody:** eliminuje externí fragmentaci (každý rámec lze využít), neviditelný pro běžící proces. Umožňuje **jemné odkládání** po stránkách/rámcích, proces tím ani nemusí být zpomalen, pokud danou stránku nepoužívá. Další výhodou je řízení přístupu pro každou stránku.
- **nevýhody:** složitější implementace (HW i SW), představuje větší režii (hledání, jestli je stránka namapovaná) a může být pomalejší (zejména při TLB miss). Stránkování způsobuje **interní fragmentaci** (přidělení větší části paměti, než je potřeba, zarovnávání, aby nebyly instrukce/data na rozmezí dvou stránek atd.)

Jednoúrovňové tabulky stránek

Nejjednodušší ale prakticky **nepoužitelný** způsob implementace tabulek stránek. OS si udržuje informace o volných rámcích a pro každý proces včetně jádra si udržuje jeho **tabulku stránek** (page table), **adresa tabulky** je uložena v k tomu určenému **registru**. Tabulka stránek obsahuje **popis mapování** (dvojice první logická adresa stránky a první fyzická adresa rámce) a **různé příznaky** (modifikace, přístupová práva, příznak globality - může používat více procesů, ...). Tabulky stránek jsou stejně jako data **udržovány v hlavní paměti**, z čehož plyne důvod nepraktičnosti jednoúrovňových stránek. Na **32 bit** systému existuje **2^{32}** adres, pokud má stránka velikost **4096 B** (**2^{12} B**) je nutné uchovávat informaci o **2^{20}** stránkách, jeden záznam o mapování stránky na rámec může mít **4 B**, což znamená **4 MiB** pro uložení stránek **pro jeden proces**. Problém exponenciálně roste u 64 bit systémů.

Při provádění příkazu v programu dochází ke **dvoum přístupům do paměti - instrukce a její operandy**. To vede ke dvoum přístupům do tabulky stránek. Pro urychlení tohto procesu se používá **TLB**.

TLB (Translation Look-aside Buffer)

Jedná se o rychlou **asociativní (obsahem adresovatelná paměť)** vyrovnávací paměť, která umožňuje **paralelně** (nebo částečně paralelně) **vyhledat** na základě

čísla stránky (1. log. adresy) odpovídající **číslo rámce** (1. fyz. adresu). **TLB** obsahuje **vybrané** záznamy (zejména **číslo stránky** a **číslo rámce**, příznaky nemusí všechny) z tabulek stránek, **rozhodně neobsahuje** data stránek/rámců. Výběr uložených položek v **TLB** je **zásadní** pro rychlou práci s pamětí, při **TLB hit** lze ihned pomocí vyhledaného čísla rámci přistoupit (**1 přístup**) k fyzické paměti. Naopak při **TLB miss** (požadované číslo stránky není v TLB) je nutno provést **dva přístupy** do paměti, a to vyhledání čísla rámce v tabulce stránek a přístup do vyhledaného rámce (doba pro vyhledání v TLB je řádově menší než doba pro přístup k RAM). V praxi lze naštěstí díky **časové a prostorové lokalitě** zajistit **TLB hit** kolem **98%**, **99%**, procesory **preemptivně** načítají očekávané mapování do TLB. K **TLB miss** může teoreticky dojít až **4x** při provádění **jedné instrukce** (instrukce je na rozmezí 2 stránek, operandy jsou na rozmezí 2 stránek). Pozor, **neplést si TLB miss s výpadkem stránku** (tj. stránce není přidělen rámec). TLB miss lze řešit:

- HW automaticky hledá v tabulkách stránek (rychlejší ale dražší),
- řízení je předáno **SW**, které do TLB doplní požadovanou dvojici a hledání v TLB se opakuje.

Při **přepnutí kontextu** (změna procesu, který je prováděn) je nutné **změnit i položky v TLB** (oba procesy používají stejný LAP → vznik kolizí). Lze řešit označením některých stránek za **globální** (sdílené mezi procesy) nebo **přidáním identifikátoru procesu** do TLB (pak nemusí být záznamy odstraňovány, ale je složitější porovnání a vyžaduje více paměti). Záznamy z TLB musí být odstraněny i při změně mapování.

Efektivní přístupová doba: $(\tau + \varepsilon)\alpha + (2\tau + \varepsilon)(1 - \alpha)$, kde

- τ : vybavovací doba RAM
- ε : vybavovací doba TLB
- α : pravděpodobnost úspěšných vyhledání v TLB (*TLB hit ratio*)

Hierarchické tabulky stránek

Eliminuje problém s **nadměrnou velikostí tabulek stránek**, na moderních procesorech jsou tabulky stránek **4 úrovňové** (tj. strom o výšce 4, v paměti jsou uloženy jen **podstromy**, které jsou potřeba) Údaje o mapování **stránek s daty** jsou obvykle v listové úrovni, mohou být ale i v **některé z vyšších úrovní** (podle příznaku), pak má stránka a odpovídající rámec větší velikost (2 MiB nebo někdy i 1 GiB). U hierarchických tabulek **roste význam TLB**, při **TLB miss** je nutné vyhledávat ve stránkové hierarchii (až 4 přístupy do paměti navíc). Optimalizace např. použitím **různých TLB pro kód a pro data**.

Hashované tabulky stránek

Tento princip je založený na **hashovací tabulce s explicitním řetězením synonym** pomocí seznamu (položky seznamu již nemusí obsahovat celé číslo stránky, pokud hashovací funkce dokáže vždy některé bity rozlišit). **Délka seznamů může být**

omezena, pak je nenalezení řízeno SW (a některá překladová dvojice může být ze seznamu odebrána). Hashovací tabulka může být **pro každý proces** nebo **sdílená** (překladové položky pak musí obsahovat i číslo procesu).

Invertovaná tabulka stránek

Pro **každý rámec** udává, **jaký proces** do něj má namapovanou **jakou stránku** (některé rámců mohou být aktuálně neobsazené). Index v tabulce rámců odpovídá číslu rámce (není ale problém s velikostí, protože tabulka je jen jedna pro všechny procesy). Pro urychlení vyhledávání se používá hashování. **Problémem** je **sdílení paměti mezi procesy** (lze řešit mapováním regionů).

Stránkování a segmentace na žádost

Je nemožné (u 32 bit teoreticky ano, u 64 bit ne), aby byl celý LAP jednoho procesu, natož více procesů, namapován do paměti (RAM má obvykle kolem 8-128 GB, LAP má u 64 bit procesoru 2^{64} adres, což je mnohem víc než 128 GB). I když ale **není celý LAP procesu namapován** do fyzické paměti, **proces o tom neví**.

Stránky/segmenty jsou mapovány na rámců a tím **zaváděny do RAM**, až když je to **potřeba** (proces čte/zapisuje do dané oblasti paměti). Namapování stránky na rámec je ale **časově velmi náročná** operace (může být nutné číst data z disku), proto se tyto operace snažíme eliminovat, procesor tak **preemptivně načítá stránky** do paměti předtím, než z nich proces chce číst/zapisovat. Při využití všech rámců je nutné část namapovaných dat odložit na disk (swap area), což většinou vede na velké zpomalení systému. Informace o načtených a odložených stránkách si vede **jádro ve svých strukturách** (také v paměti, ale zde si jádro zajistí, že tam jsou neustále), **MMU** o nich **nemá informace**. K **výpadku stránky** dochází, když do ní proces **odkazuje**, ale není načtena v tabulkách stránek nebo chybí v hashovací tabulce či invertované tabulce. Při výpadku generuje **MMU přerušení**, a jádro musí výpadek stránky obsloužit (některé stránky jsou chráněny proti výpadku, např. ty, které obsahují tabulky stránek).

Obsluha výpadku stránky

1. Kontrola, zda proces **neodkazuje mimo přidělený prostor** (o který zažádal např. použitím malloc).
2. **Alokace rámce**: buď je nějaký **prázdný** nebo musí být **vybrána oběť** (dle nějaké strategie), pokud je navíc obsah oběti modifikován (dirty bit), musí být uložen na disk - **odložen do swap**.
3. **Inicializace stránky**: ve všech případech musí být data, která v rámci zbyla po předešlém procesu **znehodnocena** aby proces, kterému je rámec nově přiřazen toho **nemohl zneužít**. Pokud se jedná o kód, nějaká statická data nebo byla dřív stránka odložena na swap (byla editována), je rámec **přepsán** tímto, jinak je **nulován**.
4. Aktualizace tabulky stránek aby odpovídala provedené změně.

5. Proces může opakovat provedení instrukce, která způsobila výpadek.

Efektivní doba přístupu do paměti: $(1 - p)T + pD$, kde

- › p : *page fault rate* = pravděpodobnost výpadku stránky,
- › T : doba přístupu bez výpadku,
- › D : doba přístupu s výpadkem.

Může dojít k **výpadku i několika stránek** při provádění jedné instrukce (kód instrukce je na rozhraní dvou stránek, operandy jsou na rozhraní dvou stránek a ještě k tomu může dojít k výpadku v hierarchické struktuře tabulky stránek). Nejvyšší úroveň tabulky stránek musí být ale chráněna proti výpadku, jinak by nebylo možné výpadek obsloužit.

Algoritmy pro výběr oběti

Výběr odkládané stránky může být:

- **lokální** v rámci procesu, u kterého došlo k výpadku,
- **globální** tj. uvolnění libovolného rámců, nehledě na to, kdo jej používá.

Typicky je ale udržován (**page daemon** - spouští se v okamžiku nedostatku) určitý počet volných rámců. Při výpadku stránky se používá **volný rámeček**. Navíc pokud byla vybrána nesprávná oběť (tj. proces požaduje stránku hned po jejím označení za oběť) je mu stránka **vrácena** a vybírá se jiná oběť. Při výběru oběti jsou **upřednostňovány nemodifikované** stránky (není je třeba odkládat na swap).

Proces musí mít vždy přidělen minimální počet rámců pro provedení jedné instrukce, jinak musí být zastaven (docházelo by k neustálým výpadkům). Počet rámců přidělených procesu se určuje na základě heuristik (např. priorita, velikost programu, frekvence výpadků, ...).

FIFO (first in first out)

Odstraňuje stránku, která byla do paměti **zavedena před nejdelší dobou**.

- **výhody**: jednoduchá implementace,
- **nevýhody**: může odstranit stránku, která je namapovaná dlouho, ale také se často používá (eliminace použití heuristiky viz výše). Trpí Beladyho anomalií, tj. při zmenšení počtu rámců se může zvýšit počet výpadků.

LRU (Least Recently Used)

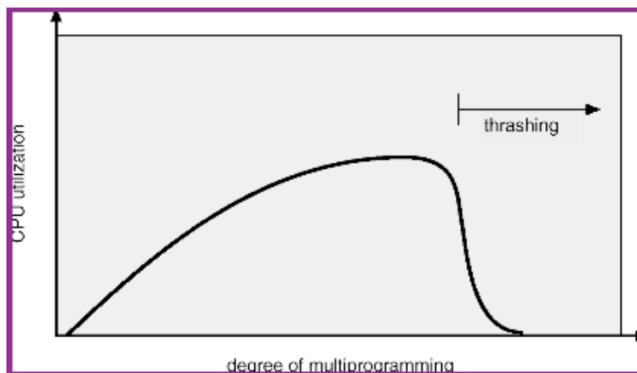
Odkládá **nejdále nepoužitou stránku**. Dobře approximuje ideální algoritmus výběru oběti (ten co vybere stránku, která již nebude použita, nebo použita za nejdelší dobu). Vyžaduje **podporu v HW**, nutné uchovávat **časová razítka** nebo stránky mít **seřazené** podle přístupu. Obvykle se approximují časová razítka několika bity pro každou stránku. **Jádro** periodicky stránky prochází a **nuluje bity** (shiftem doprava), při **přístupech** ke stránce se naopak nastavují **bity na 1 (MSB)**. Oběť je taková,

která má v registru nejmenší hodnotu (nejvíc MSBs je na 0). Lze použít pouze jeden bit (**referenční bit**), který je periodicky nulován jádrem a nastavován při odkazu na stránku, oběť je první stránka s nulovým referenčním bitem.

- **výhody:** způsobuje malý počet výběrů nesprávné oběti.
- **nevýhody:** vyžaduje HW podporu, nefunguje pro cyklické průchody rozsáhlých polí (lze však detekovat a použít **MRU** strategii **Most Recently Used**).

Trashing

Problém, který nastává při nadměrném vytížení RAM a nedostatku rámců. **Řešení výpadků stránek zabírá delší čas, než požadovaný výpočet**, a to mnohonásobně. Řešením je úplné pozastavení některých procesů a přesunu veškeré jejich paměti na swap area.



Sdílení stránek

Používá se zejména pro běžné knihovny **.dll** a **.so**. Ve **FAP** jsou knihovny načteny **pouze jednou**, využívá je současně ale více procesů, více LAP je namapované na stejný FAP. Sdílení stránek také umožňuje Inter Process Communication (IPC), tj. dva procesy používají stejný úsek **FAP**. Sdílení paměťové mapovaných souborů.

Paměťově mapované soubory

Bloky souborů z disku jsou **mapovány do stránek v paměti** a mohou být pak čteny/zapisovány běžným přístupem do paměti na místo `read()/write()`. Umožňují sdílený přístup k souborům.

39. Plánování a synchronizace procesů, transakce.

Správa procesů zahrnuje:

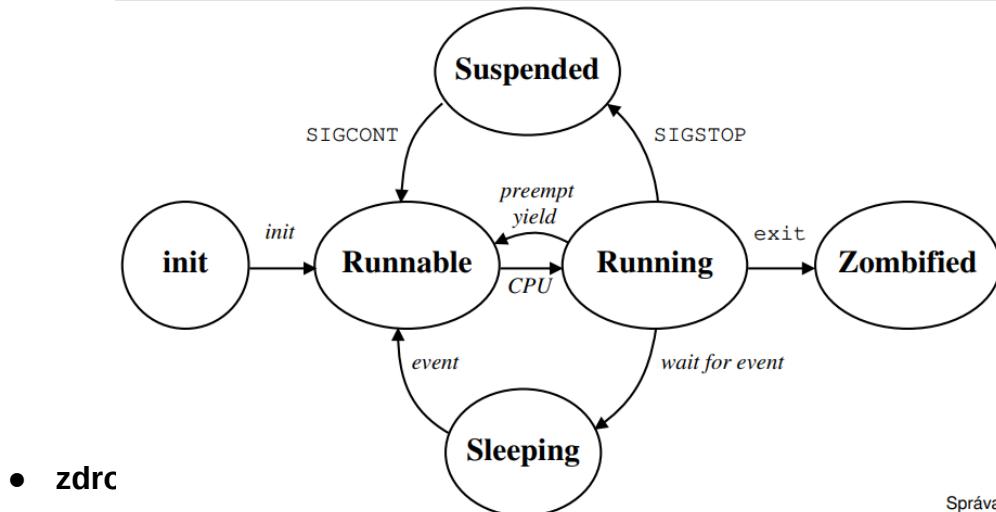
- **přepínání kontextu** (dispatcher): fyzické **odebírání** a **přidělování procesoru** procesům na základě rozhodnutí plánovače,
- **plánovač** (scheduler): rozhoduje, který proces poběží a případně jak dlouho.
- **správu paměti** (memory management): přidělování paměti procesům,
- **meziprocesovou komunikaci** (IPC): signály, sdílená paměť, roury, sockety,
- ...

Proces

Jedná se o běžící program. V OS je identifikován a náleží mu:

- **identifikátor - PID**,
- **stav plánování** (běžící, pozastavený, čekající, připravený běžet, ukončený (zombie), ...),
- **program**, který jej řídí,
- **obsah všech registrů**,
- **zásobník** - rozpracované funkce,
- **data** - statická, hromada,

stav	význam
Vytvořený	ještě neinicializovaný
Připravený	mohl by běžet, ale nemá CPU
Běžící	používá CPU
Mátoha	po exit, rodič ještě neprevzal exit-code
Čekající	čeká na událost (např. dokončení read)
Odložený	"zmrazený"signálem SIGSTOP



Process Control Block (PCB)

Je datová struktura, pomocí které je v OS **reprezentován proces** (někdy také control block nebo task struct). PCB zahrnuje (případně odkazuje na):

- **identifikátory** spojené s procesem (PID),
- **stav plánování** procesu (běžící, pozastavený, ...),
- **obsah registrů** (včetně EIP a ESP),
- **plánovací informace** (priorita),
- informace spojené se **správou paměti** (tabulky stránek, ...),
- Informace spojené s **účtováním** (doba využití procesoru, ...),
- **Využití I/O zdrojů** (otevřené soubory, používaná zařízení...).

Části procesu v paměti v Unix

- **Uživatelský adresový prostor** přístupný procesu (kód, hromada, statická data, ...).
- **Uživatelská oblast** uložená pro každý proces spolu s **daty, kódem a zásobníkem** a **částí PCB** (PID, obsah registrů, fd, obslužné funkce pro signály, účtování, pracovní adresář, ...) v uživatelském adresovém prostoru, je **přístupná pouze jádru**.
- **Záznam v tabulce procesů**, který obsahuje informace o procesu, které jsou důležité, i když zrovna neběží (PID, stav plánování, událost, na kterou čeká, čekající signály, ...). Je **trvale uložen v jádře**.

Kontext procesu

- **uživatelský kontext**: kód, data, zásobník, sdílená data,
- **registrový kontext**: obsah registrů,
- **systémový kontext**: systémové údaje o procesu tj. záznam v tabulce procesů, tabulka stránek, otevřené soubory procesem, ...

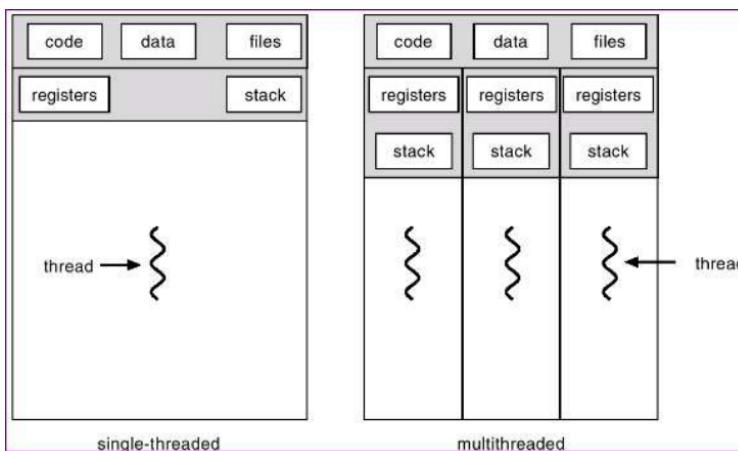
Tvorba procesu

Nový proces vytváříme **duplicací aktuálního** pomocí funkce **fork** (vrací 0 v child procesu a PID potomka v rodičovském procesu). Duplikovaný proces je **takřka totožný**. Uživatelský adresový prostor má **stejný** obsah jako obsah rodiče, sdílí otevřené soubory, obsluhu signálu, ... Pro omezení plýtvání pamětí se používá **copy-on-write** (data se doopravdy duplikují, až když jeden z procesů začne zapisovat). Potomkovský proces se **liší** návratovou hodnotou fork (0), identifikátory, údaje o plánování a účtování, nedědí čekající signály a zámky souborů. Předkem všech uživatelských procesů je proces **init s PID=1**, tento proces přebírá návratový (tím je proces ukončen ze stavu zombie/mátoha) kód i procesů, kterým dřív skončí rodič.

Změna vykonávaného programu se provádí pomocí **exec()**.

Vlákna

Jedná se o odlehčené vlákno, v rámci procesu jich může běžet více. Program je ve vláknech vykonáván paralelně. Vlákna mají své **vlastní registry a zásobník**, sdílí kód, data a další zdroje (otevřené soubory, signály, tabulku stránek). Vlákna se **rychleji vytvářejí** a je **rychlejší** mezi nimi **přepínat**. Komunikace mezi vlákny je také jednodušší, protože sdílí data (nutno používat obezřetně - uzamykat). Pro parallelizaci úloh se běžně používají vlákna na místo procesů.



Plánování procesu

Plánovač procesů (**scheduler**) rozhoduje, **který proces poběží** a případně **jak dlouho**. Existují dva druhy plánování:

- **Nepreemptivní plánování**: změnu procesu lze **provést pouze**, pokud aktuálně běžící proces **předá řízení jádru** (např. I/O operace, exit, yield), tj. dokud chce běžet, nic jej nemůže přerušit.
- **Preemptivní plánování**: změnu procesu může **jádro vynutit**, i když mu nepředá řízení, a to na základě **přerušení** (typicky od časovače, ale třeba i od disku, aj.).

Samotné **přepnutí** procesu **řídí dispečer (dispatcher)**. Plánování závisí na **prioritě** procesu a **dostupnosti zdrojů**, které potřebuje (např. paměti, dokončení IO operace, otevřání souboru, ...). **Spuštění nových procesů** se také plánuje a může být **pozdrženo** (z důvodu nedostatku paměti).

Přepnutí procesu (kontextu)

Přepnutí kontextu provádí dispečer a znamená odebrání procesoru procesu A jeho přidělení procesu B, což zahrnuje:

1. **Úschovu stavů a registrů** procesu A do **jeho PCB**.
2. **Úprava** některých řídících **struktur v jádře** a třeba i zrušení záznamů procesu A v **TLB** (záleží na implementaci TLB).
3. **Obnova registrů** procesu B z **PCB**.
4. **Předání řízení** na adresu **procesu B**, odkud bylo dříve přerušeno.

Neukládá se a neobnovuje se **celý stav** procesu. Např. namapované stránky na rámce v paměti zůstávají, **mění se pouze ukazatel na tabulku stránek**, se kterou se bude aktuálně pracovat.

Přepnutí přesto trvá několik **stovek** nobo **tisíc instrukcí**, proto doba mezi přepínáním musí být dostatečně dlouhá, aby procesor nezávadil nepřiměřeně dlouho přepínáním kontextu.

Plánovací algoritmy

Algoritmy pro plánování mohou upřednostňovat některý z požadavků na plánování procesu:

- Využití CPU
- Propustnost
- Doba běhu
- Doba čekání
- Doba odezvy

First Come, First Served (FCFS)

Procesy čekají na přidělení procesoru ve frontě **FIFO**. Jedná se o **nepreemptivní algoritmus** (tzn. proces se musí sám vzdát procesoru žádostí jádra nebo voláním yield). Procesor je **přidělen** procesu na **začátku frontu** a proces, kterému je procesor **odebrán**, je zařazen **na konec fronty**, stejně jako nový proces.

Round Robin

Algoritmus založený na principu **FIFO** jako FCFS, je ale preemptivní, tzn. procesům je přiděleno **časové kvantum** (dobu kterou mohou strávit výpočtem, než jim bude CPU odebráno), po **vypršení kvantu** nebo pokud se proces **vzdá** procesoru (yield nebo žádostí jádra např. o I/O) je zařazen na konec fronty a procesor je přidělen procesu na začátku fronty.

Shortest Job First (SJF)

Algoritmus přiděluje procesor procesu, který požaduje **nejkratší dobu** pro své další provádění na procesoru **bez I/O operací**. Jedná se o **nepreemptivní** algoritmus. **Minimalizuje** průměrnou **domu čekání** a **zvyšuje propustnost** systému, OS ale musí znát nastávající dobu procesu nebo jí aspoň dobře odhadnout. Používá se proto ve **specializovaných** systémech, kde se **opakovaně provádí podobné úlohy**. Má ale jeden zásadní problém - **stárnutí** (procesy s delším požadavkem na CPU se nemusí nikdy k CPU dostat).

Shortest Remaining Time (SRT)

Jedná se o obdobu algoritmu **SJF**. Algoritmus je **preemptivní**, ale tím, že procesor aktuálně používá proces, který má **nejmenší dobu pro dokončení**, může k preemci dojít pouze když dojde k **vytvoření** nového procesu s **kratším časem**.

dokončení, než je čas dokončení.

Víceúrovňové plánování

Procesy jsou rozděleny do různých **skupin** (typicky dle **priority**). V rámci každé **skupiny** lze použít **jiný plánovací algoritmus**. Dále existuje **další algoritmus**, který **určuje** (nejčastěji na základě priorit), jaké skupině umožní výběr procesu (dle algoritmu dané skupiny). Může docházet ke **hladovění** a **inverzi priorit**.

Víceúrovňové plánování se zpětnou vazbou

Procesy jsou rozděleny do **skupin dle priorit**. Plánování může proběhnout jednou z variant:

- Nově připravené procesy běžet jsou zařazovány do fronty s **největší prioritou** (používá se **Round Robin** - preemptivní). Jejich **priorita postupně klesá** a tím se přesouvají do front nižší úrovně, až nakonec jsou plánovány v nejméně prioritní frontě (tam se dostanou především dlouho trvající procesy).
- Proces je po spuštění zařazen do prioritní fronty dle jeho stanovené počáteční priority (statická priorita). Následně se jeho priorita mění (dynamická priorita):
 - **rosté** pokud proces často **čeká na I/O** (asi jej uživatel hodně používá, takže zajistíme jeho rychlou reakci)
 - **klesá** pokud proces **spotřebovává hodně** procesorového času.

Completely Fair Scheduler (CFS)

Přiděluje procesor každému procesu tak, aby mu bylo přiděleno **odpovídající procento procesorového času dle jeho priority**. U každého procesu si tak musí vést údaj o **stráveném virtuálním** (vliv priority) procesorovém čase a procesor přiděluje procesu s **nejmenším** stráveným virtuálním časem. Procesy organizuje dle **stráveného času** (nejméně v kořeni) do **Red-Black Tree**. Nově vzniklým procesům musí být přidělen nějaký počáteční virtuální čas dle jeho priority (protože běžící procesy již strávili čas na CPU). Algoritmus je **preemptivní** a časové **kvantum** je procesu vypočteno **na základě priority** (respektive méně prioritním procesům ubíhá čas rychleji než prioritním). Po přepnutí kontextu je nově proces, jemuž byl odebrán procesor, zpět vložen na **odpovídající místo v RB-Tree**. Algoritmus také umožňuje plánovat procesy **více uživatelům** nebo z **více terminálů** s různou prioritou.

Plánovače v Linux a Windows

Procesy jsou rozděleny do několika (desítky) úrovní priority. **Nejvyšší** priority mají procesy **reálného času** (plánovány pomocí Round Robin) **střední** prioritu mají **běžné procesy** (plánovány pomocí CFS) **nejnižší** prioritu mají **idle** procesy (většinou zaměstnávají CPU, když není co na práci). Procesy jsou startovány se **statickou prioritou**. Dynamická priorita se poté může měnit na základě chování procesu (zejména na Win):

- **Priorita se zvyšuje**: okno procesu je na **popředí**, do okna procesu **přichází zprávy** (myš, klávesnice, ...), proces je **uvolněn z čekání** na I/O.

- **Priorita klesá:** Proces spotřebovává moc CPU času, po vyčerpání každého kvanta.

Inverze priorit

Je stav, kdy efektivně prioritita nízko prioritního procesu přesáhne prioritu vysoko prioritního procesu. Stane se tak, pokud **nízko prioritní proces obsadí nějaký zdroj**, je mu odebrán CPU, než jej uvolní. Následně některý z **vysoko prioritních procesů potřebují tento zdroj**, ale musí na něj čekat. Nízko prioritní proces mezi tím **předbíhají jiné procesy** s vysokou a střední prioritou. Vysoko prioritní proces se tak dostane na řadu, až za dlouhou dobu (nejdřív musí být uvolněn požadovaný zdroj). Inverze priorit nemusí být problém, zvyšuje však obvykle odevzdu systému a v případě **real-time procesů** pracující např. s HW může být **kritická**. Inverzi priorit lze řešit:

- procesům v **kritické sekci** je přiřazena **největší priorita**,
- procesy, na které **čeká jiný proces dědí jeho prioritu** (pokud je vyšší),
- **zákaz přerušení** pokud je proces v **kritické sekci**.

Komplikace plánování procesů

- **více procesorové systémy** (dnes snad všechny) vyžadují **vyvažování výkonu jednotlivých CPU (jader)**, zde je ale navíc **problém s obsahem cache atd.**
- **hard real-time systémy** u kterých musí plánovač zajistit (**garantovat**) určitou odevzu.

Komunikace mezi procesy

IPC = Inter-Process Communication, provádí se pomocí:
signály (kill, signal, ...),
roury (pipe, mknod p, ...),
zprávy (msgget, msgsnd, msgrcv, msgctl, ...),
sdílená paměť (shmget, shmat, ...),
sockety (socket, ...).

Procesy jsou členěny na:

- **Úloha** - Skupina paralelně běžících procesů spuštěných jedním příkazem a propojených do **pipeline** (|).
- **Skupina procesů** - Množina, které je množné **poslat signál jako jedné jednotce** (může mít vedoucího - tím je implicitně první proces), procesy jsou přidávány do stejné skupiny jako rodič, lze ale změnit.
- **Sezení** - Každá skupina procesů je v **jednom sezení** (může mít vedoucího).

Signály

Jeden z principů komunikace mezi procesy. Jedná se o **číslo (int)**, které je zasláno procesu pomocí **speciálního rozhraní**, které je pro to určeno. Lze zde najít **analogii** s přerušením na SW úrovni. Také se **dělí asynchronně** a také pro ně **definujeme obslužné funkce**. Signály jsou generovány při:

- chybách (chybná práce s pamětí, aritmetická chyba - dělení 0, ...),
- externích událostech (vypršení časovače, dokončení I/O, ...),
- na žádost procesu (např. rodič posílá signál potomkovi, signál kill, ...).

Obsluha signálů může být **složitá a potenciálně obsahovat chyby** (kvůli asynchronnímu vzniku signálů), při obsluze **nelze použít některé funkce**, protože by to mohlo ovlivnit normální běh procesu např. **printf()**. Pokud vzniknout dva signály hned za sebou, může být druhým přerušena obsluha prvního. Příklady: SIGHUP, SIGINT, SIGKILL, SIGALARM, SIGSTOP, SIGCONT, SIGCHILD, SIGUSR1, SIGUSR2, ... Kromě **SIGKILL** a **SIGSTOP** lze signály předefinovat. Implicitní reakce na signály jsou buď ukončení procesu (časté), ignorování signálu, zmrazení procesu, nebo rozmrazení procesu. Signály až na SIGKILL, SIGSTOP, SIGCONT lze blokovat. Nastavení **obsluhy signálů** včetně blokování se **dědí** na potomky **při fork**. Při **exec** se nastaví **implicitní obsluha**. Signály se **zasílají** (funkce **kill**) na základě **znalosti PID** případně lze zaslat všem procesům určité skupiny (např potomkům). Čekání na signály pomocí **pause()** nebo **sigsuspend()**.

Synchronizace procesů

Synchronizaci provádíme, aby při **paralelném přístupu** ke sdíleným zdrojům (sdílená paměť, soubory, I/O zařízení, ...) nedocházelo k chybám způsobených **nesprávným pořadím** provedených dílčích **operací** různými procesy. Např. 2 procesy sdílí paměť:

proces 1 kód: <pre>N++;</pre> assembler: <pre>register = N register = register + 1 N = register</pre>	proces 2 kód: <pre>N--;</pre> assembler: <pre>register = N register = register - 1 N = register</pre>
--	--

Pokud je **N** na začátku **1**, mělo by být **1** i po provedení následujícího kódu (oběma procesy), ale může být také **0** nebo **2**.

Race condition

Časově závislá chyba (souběh) je chyba, která vzniká při **přístupu ke sdíleným zdrojům** kvůli **různému** pořadí provádění jednotlivých **paralelních výpočtů** v systému.

(Sdílená) kritická sekce

Jedná se o **úseky v programech**, které řídí **parallelně** běžící procesy, ve kterých dochází k přístupu ke **sdíleným zdrojům**. Provádění jednoho z **úseku** jedním procesem **vyloučuje** provádění libovolného z těchto úseků ostatními procesy.

Problém kritické sekce

Jedná se o problém zajištění **korektní synchronizace** procesů na množině **sdílených kritických sekcí**. Je nutné **zajistit**:

- **vzájemné vyloučení**: **nanejvýš jeden** proces je v daném **okamžiku** v dané množině kritických sekcí. Případně je tam nanejvýš **k** procesů.
- **dostupnost kritické sekce**: Je-li kritická sekce **volná**, respektive je volná aspoň v určitých okamžicích, proces **nemůže čekat neomezeně dlouho** na **přístup** do ní.

Musíme se **vyhnout**:

- uváznutí (deadlock),
- blokování (blocking),
- stárnutí (hladovění - starvation).

Data race

Časově závislá chyba nad daty nebo také souběh nad daty. K této chybě může dojít, pokud ke zdroji s **výlučným** přístupem přistupuje **více procesů bez synchronizace** (2 a více) a alespoň **jeden** k němu přistupuje **pro zápis**.

Deadlock (uváznutí)

Jedná se o situaci, kdy **každý proces** z určité **neprázdné** množiny procesů je **pozastaven a čeká** na uvolnění nějakého zdroje s **výlučným přístupem**, který je **vlastněn** nějakým procesem z dané množiny, který **jediný jej může uvolnit**, a to až po dokončení práce s ním. (Ale tu dokončit nemůže, protože čeká. Takže každý proces čeká na jiný proces - v **grafu** čekání se vytvořila **kružnice**).

Podmínky uváznutí (Coffmanovy podmínky)

1. **Vzájemné vyloučení** při použití **prostředků** (tj. k prostředku může v jeden čas přistupovat pouze jeden proces),
2. **vlastnictví** alespoň **jednoho** prostředu (zdroje), pozastavení **a čekání** na další (tj. proces má alokovaný nějaký zdroj, potřebuje i jiný, o něj žádá, ten není ale k dispozici, tak čeká),
3. prostředky **vrací proces**, který je **vlastní**, a to až po dokončení jejich používání (tj. pokud má proces alokovaný nějaký zdroj, např. I/O zařízení, pouze tento proces může říct, že už jej nepotřebuje, a to až v moment, kdy jej doopravdy nepotřebuje - dokončil čtení/zápis),
4. **cyklická závislost** na sebe čekajících procesů (tj. proces A čeká na proces B a proces B čeká současně na proces A).

Řešení uváznutí

- **prevence uváznutí:** porušení jedné z **Coffmanových podmínek**,
- **vyhýbání se uváznutí:** kontrola grafu **žádostí a vlastnictví zdrojů**, zamezení vzniku kružnice.
- **detekce a zotavení:** existuje speciální proces (mimo množinu procesů, na které došlo k uváznutí), který detekuje vznik uváznutí a nějakým způsobem jej vyřeší.

Blocking (blokování)

Blokování při přístupu do kritické sekce je situace, kdy proces, který žádá o přístup do kritické sekce **musí čekat**, i když je **kritická sekce volná** (žádný proces se nenachází v žádné z množiny těchto kritických sekcí - jinak řečeno, **žádný proces nepoužívá daný sdílený zdroj s výlučným přístupem**).

Starvation (stárnutí)

Stárnutí je situace, kdy **proces čeká** na splnění podmínky, která **nemusí nikdy nastat**. V případě kritické sekce se jedná o splnění podmínky, která umožňuje vstup do kritické sekce. V případě plánování procesů se jedná o situaci, kdy proces nemusí být nikdy plánovačem naplánován pro běh na procesoru (předbíhají jej jiné procesy). Uváznutí (deadlock) i blokování (blocking) lze zobecnit na stárnutí.

Livelock

Jedná se o zvláštní situaci stárnutí a v kombinaci s uváznutím. Každý proces z určité neprázdné množiny **běží** ale provádí jen **omezený úsek kódu**, ve kterém **opakovaně** žádá o přístup do **kritické sekce** (žádá o zdroj s výlučným přístupem), ve které je jiný proces z dané množiny procesů, který **jediný jí může uvolnit**, pokud by mohl pokračovat.

Zajištění výlučného přístupu do KS - synchronizace

Výlučný přístup do KS se zajišťuje pomocí tzv. **spinlock**. Jedná se o řešení, které umožňuje **aktivní čekání**. Spinlock lze implementovat pomocí:

- **speciálních algoritmů**, které nevyžadují atomické operace. Např.
 - **Petersonův algoritmus:** Proces nastavením příznaku vyjádří svůj zájem o kritickou sekci, ale zároveň dá přednost jinému procesu. Následně čeká na uvolnění kritické sekce, nebo až mu druhý proces vrátí přednost.
 - **Bakery algoritmus L. Lamporta:** Před vstupem do KS získá proces **lístek**, jehož hodnota je větší než hodnota lístků procesů, které již lístek mají a čekají. Pokud je hodnota stejná (kvůli tomu že lístek procesy získaly současně - souběžně četly největší hodnotu), rozhoduje se podle velikosti **PID**. Držitel **nejmenšího čísla lístku může vstoupit** do kritické sekce,

- pomocí **atomických instrukcí**, jejíž atomicitu zajišťuje HW implementace, např. instrukce **SWAP**.

Aktivní čekání lze využít na krátkých neblokujících sekcích bez preemce (tj. zákazem přerušení). Tyto krátké sekce se mohou nacházet např. v **implementaci operací** nad **semaphorem** nebo **monitorem**, nikoli však v uživatelském kódu.

Klasickými synchronizačními problémy jsou např. **producent a konzument** (producent zapisuje do fronty, konzument odebírá), **čtenář a písář** (současně může číst libovolný počet čtenářů, při zápisu však může přistupovat pouze písář).

Semafor

Synchronizační nástroj, který **minimalizuje** (případně úplně odstraňuje) **aktivní čekání**. Pokud proces aktuálně nemůže vstoupit do kritické sekce (tj. uzamknout semafor - uzamčení musí být **atomické**, zde se vyskytuje **spinlock** a **zákaz přerušení**), je **pozastaven** (nečeká aktivně) a umístěn do **fronty čekajících procesů** (pořadí procesů není garantováno) na uvolnění KS. Obecně mohou být semafory implementovány pomocí **celočíselné proměnné** a **fronty**. Proměnná udává, kolik procesů může ještě vstoupit do KS. Pokud je **0** a **menší** sekce je **uzamčena**. Zvláštním (a nejpoužívanějším) případem je poté **binární semafor - mutex**, který umožňuje vstup do KS pouze **jednomu procesu**, který jediný jej může zpět odemknout.

Monitor

Jedná se o vysokoúrovňový synchronizační prostředek. Zapouzdřuje data a poskytuje operace, které zajišťují správné provádění operací nad chráněnými daty.

- **wait**: uspí dané vlákno/process (pasivní čekání),
- **notify**: **probudí jeden** (pokud nějaký existuje, jinak prázdná operace) z čekajících procesů (dříve volaly wait), ale **pokračuje aktuální proces** (vlákno), který zavolalo notify.
- **signal**: **probudí jeden** (pokud nějaký existuje, jinak prázdná operace) z čekajících procesů (dříve volaly wait) a **pokračuje nově probuzený proces**.
- **notifyAll**: **probudí všechny** čekající procesy (pokud nějaké čekají, jinak prázdná operace).

Prevence uváznutí (podrobněji)

Zrušíme platnost některé z nutných **podmínek uváznutí (Coffmanovy podmínky)**. To lze zabudovat přímo do celého implementovaného systému a nějak ověřit (verifikace), nebo pro tento účel využívat systém přidělování zdrojů, který bude vždy vynucovat porušení alespoň jedné podmínky. Příklady řešení pro jednotlivé podmínky:

1. **Nepoužíváme** sdílené prostředky nebo používáme takové, ke kterým **není** nutné přistupovat **výlučně** (může k nim současně přistupovat více procesů).
2. **Statická** (tj. program neběží) kontrola kódu, že proces žádá o zdroj (a tedy

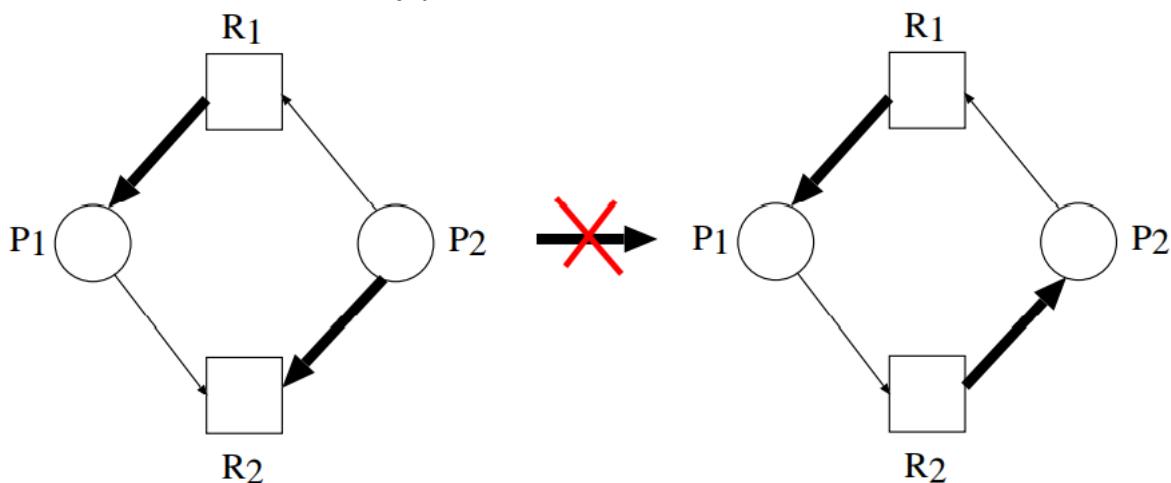
může být pozastaven), jen v případě, že aktuálně **žádný zdroj nevlastní**. Nebo **dynamická** kontrola uvedeného, pak je ale nutné řešit, co s takovým procesem. (Nabízí se jeho ukončení a odebrání jím vlastněných zdrojů, to ale může vést na nekonzistence a nedokončení některých úloh).

3. Při pozastavení procesu (neúspěšné žádosti o další zdroj) jsou mu **odebrány všechny jím vlastněné** zdroje. Ty jsou mu **vráceny** včetně nově žádaného zdroje, až jsou **všechny k dispozici** a proces je znova spuštěn. (Odebrání zdrojů v průběhu práce s nimi může způsobit nekonzistence).
4. **Očíslování prostředků** a jejich přidělování v **pořadí**, které **vylučuje** vznik cyklické závislosti (tj. **vzestupně** nebo **sestupně**).

Vyhýbání se uváznutí (podrobněji)

Obecně lze problém řešit tak, že procesy **deklarují**, jaké zdroje **aktuálně potřebují** a na základě aktuálního **stavu přidělených** zdrojů a těchto **žádostí o zdroje** se rozhodne, jestli **může** být dané žádosti **vyhověno**, nebo proces **musí čekat**. Žádosti o zdroj je vyhověno, pokud aktuálně **nemůže vzniknout cyklická závislost** na sebe čekajících procesů a **cyklická závislost nebude možná ani v budoucnu**, a to při zohlednění té **nejhorší možné** situace.

V praxi se tento problém řeší pomocí grafových algoritmů, konkrétně pomocí **grafu alokace zdrojů**. Zdroj je **přidělen** jen tehdy, pokud **nehrozí vznik cyklu v grafu**, jinak proces musí čekat. (Vznik cyklu v grafu neznamená vznik uváznutí, pouze značí, že k uváznutí může dojít).



Detekce uváznutí (podrobněji)

Na dané **neprázdné množině** procesů **může** dojít k **uváznutí**, ale existuje nějaký **další proces mimo tuto množinu**, který periodicky (nebo nějak jinak) **kontroluje**, zda k uváznutí **došlo**, a pokud ano, **provede zotavení** (zajistí, že uváznutí bude zrušeno).

Detekce uváznutí může být v konkrétním případě řešena pomocí **grafu vlastnictví a čekání na zdroje**. **Cyklus** v tomto grafu znamená vznik **uváznutí**.

Zotavení z uváznutí lze nejjednodušeji řešit **ukončením** (restartem) některého z

procesů - **victim** (děje se tak např u databázi v SŘBD) a případně doplnit ukončení procesu o **rollback** (viz transakce v DB). Dále lze zotavení řešit **odebráním zdrojů** některému procesu a jejich **přidělení jinému** a poté zajištění, že tomuto procesu budou opět přiděleny včetně těch, co potřeboval původně.

Transakce na úrovni OS

Jedná se o druh zpracování, při kterém je **skupina logických operací - transakce**, provedena jako **jeden celek**, tj. buď **výbec nebo všechny**. Pokud se při zpracování v rámci transakce vyskytne jakákoli **chyba** a transakce nemůže být dokončena, všechny dílčí operace musejí být **vráceny do stavu před začátkem transakce**. Jejich zpracování se musí **vypořádat s výskytem poruch a parallelismem**. Definice z databází pomocí **ACID** zde nemusí být úplně přesná. Např. dosažení **trvalosti** (durability) nemusí být možné, pokud za transakci považujeme přepnutí kontextu (asi těžko zajistíme, že po dokončení a přepnutí kontextu a následném pádu systému bude přepnutí kontextu trvalé). U transakcí na úrovni OS tak zajišťujeme zejména **A** (atomicity - atomicitu), tedy že transakce je provedena v celém svém rozsahu, nebo není provedena žádná z jejich částí (nemohou být při přepnutí kontextu ponechány stejné registry a pouze změněn zásobník).

Typickým **příkladem** transakčního zpracování na úrovni OS je **zápis na disk**. Využívá se zápisu do **žurnálu** a řešení chyb pomocí **REDO** a **UNDO**, tedy velmi podobný princip databázové transakci. Těžko ale v tomto případě zajistíme **C** (consistency, konzistence) zapsaných dat **ve smyku jejich významu**, můžeme pouze zajistit, že byty jsou zapsané přesně tak, jak vyžaduje proces.

40. Objektová orientace (základní koncepty, třídně a prototypově orientované jazyky, OO přístup k tvorbě SW).

Objektově orientované programování (OOP) je programovací paradigma, které se začalo hojně objevovat v 80. letech minulého století. Základem tohoto paradigmatu je abstrahovat a modelovat principy reálného světa. Řeší se tak pomocí objektů a jejich vzájemné komunikace. V dnešní době je toto paradigmá jedno z nejrozšířenějších, zejména pro velké aplikace. Mezi **výhody** patří **analogie reálného a softwarového modelu, flexibilita, znovupoužitelnost**. Nevýhodou může být složitější sémantika, delší učící se křivka či režie spojená s prací s objekty . Mezi základní koncepty OOP patří:

- **Abstrakce** (Abstraction): pomocí objektů,
- **Zapouzdření** (Encapsulation): pomocí viditelnosti atributů,
- **Mnohotvárnost** (Polymorphism): abstraktní funkce a VMT,
- **Dědičnost** (Inheritance): generalizace/specializace.

Klasifikace objektově orientovaných jazyků

- Dle přístupu k vytváření objektů:
 - **Beztrídní** OOJ (JavaScript, Lua - prototype-based, class-less),
 - **Třídní** OOJ (C#, C++ - class-based)
- Dle čistoty objektového modelu:
 - **Čisté** (Ruby, Python - vše je objekt),
 - **Hybridní** (C#, C++ - míchání s jinými paradigmaty, doplněny objekty),
- Dle platformy pro běh OO programů:
 - **Překládané** (C++ - efektivita běhu, více zdrojového textu),
 - **Interpretované** (Python, PHP - pomalé, velmi flexibilní),
 - **Částečně interpretované** (C#, Java - bytecode, mezikód, virtuální stroj)
- Dle dědičnosti (počet přímých předků):
 - **Jednoduchá** (C#, Java - oba tyto jazyky ale podporují vícenásobnou dědičnost rozhraní),
 - **Vícenásobná** (C++, Python - problematická, nutno řešit kolize)
- Dle předmětu dědění:
 - **Dědičnost implementace** (C++, C#, Java, Python, ...)
 - Třídní dědičnost: nadřída (superclass), podřída (subclass)
 - Delegace
 - **Dědičnost rozhraní** (C#, Java)

Klasifikace nejen OOJ

- Dle způsobu určení typů:
 - **Beztypové** (Lambda kalkul - nemají žádný typ),
 - **Netypované** (Python, JavaScript - uživatel nepracuje s typem, typ ale mají a lze na vyžádání zjistit),
 - **Typované** (C++, C# - uživatel explicitně specifikuje typ, nebo musí být odvoditelný)
- Dle důslednosti kontroly typů:
 - **Silně typované** (Rust, Go - nelze provádět implicitní konverze, type-safe),
 - **Slabě typované** (C, C++ (méně než C) - implicitní konverze)
- Dle doby kontroly typů:
 - **Staticky typované** (C, C++ - při překladu, před spuštěním),
 - **Dynamicky typované** (Python, JavaScript - za běhu, run-time).

Minimální model OO výpočtu

Minimální výpočetně úplný OO model výpočtu potřebuje pouze:

- proměnné,
- konstrukt objektu (způsob vytváření objektů),
- zasílání zpráv (komunikace mezi objekty).

Základní koncepty OOP

Tyto koncepty by měly být více méně shodné pro každý OOJ.

Abstrakce - Objekt

Objekt je **autonomní výpočetně úplná** entita, obvykle je tvořena **atributy** a **metodami**. Identita objektu ale nezávisí na jeho attributech. U třídních jazyků jsou objekty **instancemi tříd**.

Kompozice

Objekt může obsahovat jako položky i jiné objekty.

Zapouzdření

Jedná se o **uzavřenost** vůči okolním objektům, dnes je to obvykle implementováno pomocí **identifikátorů viditelnosti (public, private, protected)**. V případě **zpráv** mluvíme o přístupu přes:

- **veřejný** protokol umožňuje okolním objektům zasílat zprávy tomuto objektu.
Zaslání zprávy veřejným protokolem může vést na:
 - **chybu** - objekt nerozumí zprávě,
 - **invokaci** odpovídající metody a **navrácení** výsledné hodnoty odesílateli.

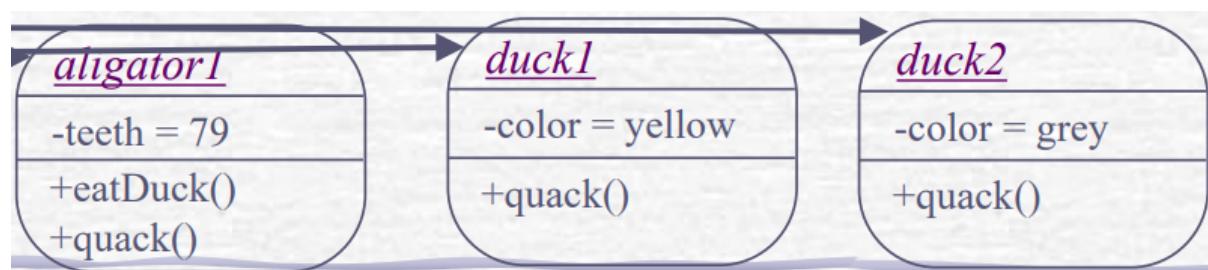
- **interní** protokol se používá, když objekt zasílá **zprávu sám sobě** (this, self). Zaslání zprávy interním protokolem může vést na:
 - **chybu** - objekt nerozumí zprávě,
 - **přístup k atributu** (čtení nebo zapsání),
 - **invokaci** odpovídající metody a návratu výsledku,

Zpráva

Zpráva je tvořena z **příjemce** (objekt, kterému je zpráva zaslána), **selektoru** (metody) a **argumentů**.

Polymorfismus (Mnohotvárnost)

Vychází z toho, že **stejnou zprávu lze zaslat různým objektům** (často je to ale **omezeno** typovým systémem, aby nedocházelo k chybám typu "nerozumí zprávě") a stejné zprávy lze zasílat různým objektům jen v rámci dědičnosti). Protokol umožňuje **individuální reakci** na zaslané zprávy (volající díky zapouzdření nezná implementaci zprávou invokované metody, implementace se u různých objektů může lišit).



Dědičnost

Vychází z toho, abychom nemuseli neustále **opakovat implementaci** podobných vlastností. Jedná se o jednu z hlavních předností OOP, a to **znovupoužitelnost**. Dědičnost umožňuje **sdílení** společných **položek** (atributy a metody) **od předků** a možnost definovat **individuální položky v potomcích**. Zděděný (specializovaný) objekt (třída) tak sdílí všechny atributy a metody předka (jejich použití objektem závisí na viditelnosti - musí být **public** nebo **protected**) a dále může být specializován:

- **přidáním** nových atributů a metod,
- **modifikací** metod (**override**),
- někdy lze také zakázat některé položky, většinou ale **ne**, viz dále.

Zahrnutí typu (subsumption)

Pokud B je podtřída třídy A, lze instanci třídy B využít kdekoliv je očekávána instance třídy A. Pak se provádí přístup k **objektu třídy B přes protokol A**.

Problém vícenásobné (třídní) dědičnosti

- nadtřídy obsahují položky **stejného jména a typu**,

- nadřídy obsahují položky **stejného jména**, ale **různých typů**,
- **inicializace instancí** - pořadí volání konstruktorů,
- **uložení instancí v paměti** - instance třídy **C** (**C** je přímá podtřída **A** i **B**) lze využít **kdekoliv** očekávám instance tříd **A** nebo **B**.

Řešení problémů vícenásobné (třídní) dědičnosti

Nejjednodušší je vícenásobnou dědičnost **zakázat**, stejně se ukázalo, že není často při programování nutná. Pokud ji ale potřebujeme:

- **Metoda stejného jména a typu**: zakázat, použít první výskyt (dle pořadí zapsání dědění v kódu), programátor musí explicitně vybrat jednu, skrytí (např. `A::m()` přístupná jen v `A`, ne v `C`).
- **Metody stejného jména a různých typů (parametrů)**: zakázat, povolit přetěžování metod (overloading), lze ale pouze u metod s rozdílnou signaturou (návratová hodnota není součást signatury), také s rozdílnými parametry.
- **Atributy stejného jména a typu**: zakázat, skrytí a existence obou (např. `A::d` přístupný jen v `A`, ne v `C`), sloučení.
- **Atributy stejného jména a různého typu**: zakázat, nebo ponechat, ale při práci s objektem musí být možné atributy vždy odlišit.

Tvorba nových objektů

- Vytvoření **prázdného objektu** a přidání položek (atributů a metod).
- Klonování (kopie) a přidání či úprava položek. Klonovanému objektu říkáme **prototyp**. Kopie může být **mělká** (pouze 1. úroveň atributů), nebo **hluboká** (celý objekt). Princip klonování používají beztřídní OOJ.
- Vytvořením pomocí **předlohy - třídy** a naplnění atributů hodnotami. Děje se tak pomocí **konstruktoru**. Pro každý objekt je definována jeho třída - předloha. Třída také může být objektem první kategorie.

Vzájemné vazby mezi třídami/objekty

Řeší se přes **ukazatele/reference** a **dopřednou deklaraci**. Dopředná deklarace se poté používá i pro metody a ty se definují mimo tělo funkce, aby v metodách bylo možné odkazovat atributy cyklicky závislých proměnných.

Třídně orientované jazyky

Třídně orientované jazyky využívají pro tvorbu nových objektů - **instancí šablony**, které nazýváme **třídy**. Třída může být sama o sobě objekt nebo entita, která obsahuje:

- seznam **instančních atributů** (atributů, které bude mít objekt po vytvoření) včetně **metadat**,
- **data třídních atributů** (pokud je třída taky objekt),

- implementace **instančních metod** (sdílené mezi instancemi),
- reference na její třídu,
- **statické** (!= třídní) položky - **atributy** a **metody** (pokud je třída entitou jazyka a není objekt).

Instance - objekt je poté tvořen svoují **identitou**, **referencí na třídu**, **daty instančních atributů**.

Třída jako objekt první kategorie

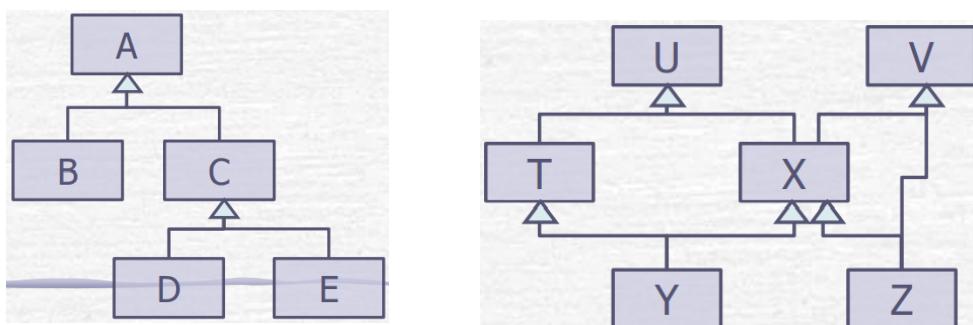
Třída je také objekt a pracuje se s ní analogicky jako s objekty, tj. **zasíláním zpráv**. Ve třídních metodách lze použít **self/this** parameter pro přístup k objektu třídy.

Třídní dědičnost

Stejný význam jako dědičnost na úrovni objektů (ve většině jazyků používáme hlavně třídní dědičnost). Dědičnost tříd je šablonou pro dědičnost vzniklých (instanciovaných) objektů. Hlavní účel je **sdílení/znovupoužití** položek skrz třídy, **rozšíření** o nové položky a **redefinice** u specializovaných tříd.

Hierarchie třídní dědičnosti

Matematicky můžeme mluvit o dědičnosti jako **relaci částečného uspořádání na množině tříd**. Hierarchie třídní dědičnosti je poté tvořena grafem relace dědičnosti. Levý obrázek ukazuje dědičnost u jazyků pouze s **jednoduchou** dědičností, na pravém je znázorněna vícenásobná dědičnost.



Statické volání funkce (žádná vazba)

Jedná se o běžnou metodu/funkci, která je pouze ve jmenném prostoru třídy. Uvnitř metody **nelze** používat **self/this** parameter, není implicitně předán. Může ale používat jiné statické proměnné třídy.

Brzská (časná) vazba

Při překladu je jasné, jaká implementace bude volána, **není** třeba mít **odkaz na metodu** v konkrétním objektu, překladač určí volání metody na základě jeho typu. V metodách lze odkazovat objekt, nad kterým je metoda volána pomocí **self/this** parametru, který je do metody implicitně předán. (ve skutečnosti překladač převede

zápis obj.call() na call(obj))

Pozdní vazba

Umožňuje **polymorfismus** (volání virtuálních metod), volání metody nad objektem se řeší **až za běhu** dle jeho konkrétního typu. Stejně jako u brzké vazby lze v polymorfních metodách odkazovat objekt nad kterým je metoda volána (**self/this**). U překládaných OOJ se řeší pomocí **tabulky virtuálních metod**.

Implementační problémy

Problémy implementace vychází z toho, že OOJ používají **objektovou paměť**, která je **asociativní** a **heterogenní** a modelem výpočtu je **zasílání zpráv** (invokace metod) objektům. Paměť počítáče je ale **homogenní** a **neasociativní** a modelem výpočtu je **volání instrukcí** a použití **zásobníku**. Je nutné řešit:

- **Uložení instancí a přístup k atributům**, tak aby byla reflektovaná dědičnost. Problematické je zejména zajistit přístup k atributům objektu při vícenásobné dědičnosti, tak aby šlo k objektu, který dědí z více objektů, přistupovat protokolem každého z obecných objektů. U jednonásobné dědičnosti to nemusí být problém, hodnoty specializovaných atributů se ukládají za hodnoty obecných atributů a při přístupu se specializované atributy ignorují (paměť se přetypuje na obecný objekt). Ale jak uložit atributy při mnohonásobné dědičnosti? Pouhé "přetypování" nebude fungovat.
- **Uložení a invokace polymorfních metod**. Ukládání kódu metody v objektu je **nesmysl**, stačí metodu ukládat **jednou**, např. ve třídě a objekt implicitně předávat jako parametr (nultý parametr) při volání metody. Do objektu by tedy mohlo stačit uložit **odkazy na metody**, což řeší i **polymorfismus** tj. každý objekt má odkazy na takové metody, které doopravdy implementuje. Tento způsob je jednoduchý ale má 2 zásadní problémy:
 - **přístup k metodám předků** (pomocí **super/base**) po jejich redefinici - objekt by musel obsahovat odkazy na jeho redefinici metody, všechny různé redefinice metod předky a originální definici metody,
 - **plýtvání pamětí** - při vytvoření tisíce stejných objektů bude v paměti uloženy zbytečně tisíce stejných odkazů na metody.

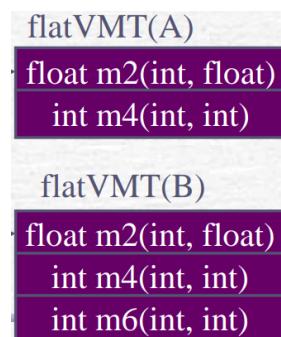
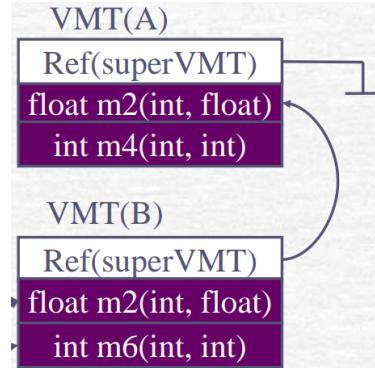
Řešením problémů je použití **tabulky virtuálních metod** (virtual method table - VMT), kterou odkazuje objekt na místo všech jeho metod.

Tabulka virtuálních metod (VMT)

Je to opravdu **tabulka virtuálních metod** (NE virtuální tabulka). Pro každou třídu (typ objektu) existuje **pouze jednou** a všechny instance (objekty) této třídy na ní odkazují. Dědičnost je řešena tak, že z VMT specializovaného objektu vede odkaz na VMT jeho předka a z VMT předka odkaz na VMT dalšího předka atd.

Tento princip **zanořeného** odkazování je ale **pomalý**, proto se za cenu mírného zvýšení spotřeby paměti tabulka pro danou třídu odkazuje **všechny polymorfní**

metody, i když nebyly přepsány (**flat table**). Tabulka **neobsahuje** odkaz na **rodičovskou** tabulkou. Rodičovskou tabulkou při volání přes **self/this** dokáže **určit** překladač na základě typu self/this, protože každé **tabulce přiděluje statickou adresu**.



Vytváření instancí (objektů)

Většinou se provádí pomocí klíčového slova **new** a je tvořena ze 2 kroků:

1. **Přidělení paměti** (většinou na hromadě) na pro danou instanci (objekt).
2. **Inicializace atributů** pomocí **konstruktoru** (speciální metoda), nutné si v konkrétním jazyce zjistit, jaké je pořadí volání konstruktorů při dědičnosti.

Rušení instancí (objektů)

Při rušení objektů se volá speciální metoda tzv. **destruktur** (opět je nutné znát pořadí volání destruktorů při dědičnosti) a může být provedena:

- **Implicitně** pomocí správce paměti (**Garbage Collector**) objekty jsou z paměti mazány, když již jsou nedosažitelné. Programátor **nemá pod kontrolou** kdy se tak stane a jak dlouho to zabere → nelze použít např. u real time systémů s požadovanou odezvou. Algoritmy uvolňování ale obvykle **nezpůsobují** memory leaks.
- **Explicitní** použitím klíčového slova **delete** (nebo jiného) řeší **programátor** a má nad uvolňováním paměti **plnou kontrolu**, nicméně **může** vést na

memory leaks.

Reflektivita/Reflexe (Reflection)

Vlastnost jazyků (interpretovaných a částečně interpretovaných), která umožňuje:

- zkoumat vnitřní reprezentaci (**Introspection**) entit programu (objektů),
- měnit vnitřní reprezentaci (méně časté).

Na základě prováděných změn a zkoumání vnitřní struktury dělíme reflektivitu na:

- **Strukturální:** pracuje se **statickými strukturami** (balíčky, třídy, metody), např. zjišťujeme názvy atributů objektu.
- **Behaviorální:** pracuje s **prováděním programu** (invokace metody, přiřazení, zásobník volání), např. voláme metody na základě hodnoty řetězce, který obsahuje její název.

Reflektivita umožňuje jednodušeji vytvářet generický software. Nejlepším příkladem je serializace a deserializace. Na základě reflexe můžeme zjistit názvy atributů, získat jejich hodnoty a vytvořit např. JSON reprezentaci. Taková funkce pak může mít za parametr libovolný objekt a nepotřebujeme mít speciální funkci pro každý objekt.

Typy vs. Třídy

- **Typ** udává množinu validních hodnot a operací nad nimi,
- **Třída** udává množina vnitřních stavů a operací nad nimi.

Typy jsou obecnější než třídy. Třída často určuje typ, ale né naopak. Typ může být určen:

- **jménem** např. int, Car, Person, ... Pak pokud máme funkci s parametrem **typu A**, můžeme jí volat s:
 - **instancemi** třídy **A**,
 - **instancemi** libovolné **podtřídy A** (kompatibilní podtyp).
- Pro zajištění kompatibilního typu musíme použít dědičnost - **vyžádaná dědičnost a polymorfismus** se vyskytuje také **pouze v rámci dědičnosti**.
- **výčtem položek** tj. nezáleží, jestli jde o třídu Car nebo Person. Typ se zjistí prozkoumáním položek instance - **test implementace** potřebného podtypu:
 - **staticky** během překladu (často se k tomu využívá rozhraní/interface).
 - **dynamicky** za běhu

Umožňuje **polymorfismus nezávislý na třídní dědičnosti**, např. funkce má jako parameter typ JmenoStari, která obsahuje dva atributy "jméno" a "stáří", pokud Car i Person mají (i mimo jiné) tyto atributy, lze je použít jako parametr pro volání této funkce (obdobně s metodami a kombinací metod a atributů).

- **kombinace** např. vícenásobnou dědičností nebo rozhraním

```

class A is
    int d1;
    float d2;
    float m(int);
endOfClass
subclass C of A is
    int d3;
    float m(int);
    int g(float);
endOfClass
  
```

```

class B is
    float d2;
    int d1;
    int h(float);
    float m(int);
endOfClass
  
```

```

static p(o : InstanceOfType(A));
static q(o : InstanceOfType(C));
α = new A;
β = new B;
γ = new C;
  
```

Vyžádaná dědičnost

		Vyžádaná dědičnost	Skutečné podtypy
call p(α)	OK	call p(α)	OK
call p(β)	ERROR	call p(β)	OK
call p(γ)	OK	call p(γ)	OK
call q(β)	ERROR	call q(β)	ERROR?
call q(γ)	OK	call q(γ)	OK

Implementace skutečného podtypu je **náročná** (změna pořadí položek atd. kód metody je ale statický a očekává položky na stejných pozicích, ...) je nutné skutečný podtyp řešit **až na běhu**. Skutečný podtyp se tak typicky vyskytuje u **interpretovaných** (nebo alespoň částečně interpretovaných) a **dynamicky typovaných** jazyků. Objekt je ve skutečnosti **slovník** s názvem položky (klíč), hodnotou, typem, aj. a interpret hledá, jestli ve slovníku existují dané položky. U **překládaných a staticky typovaných** jazyků lze podtyp nahradit **rozhraním**.

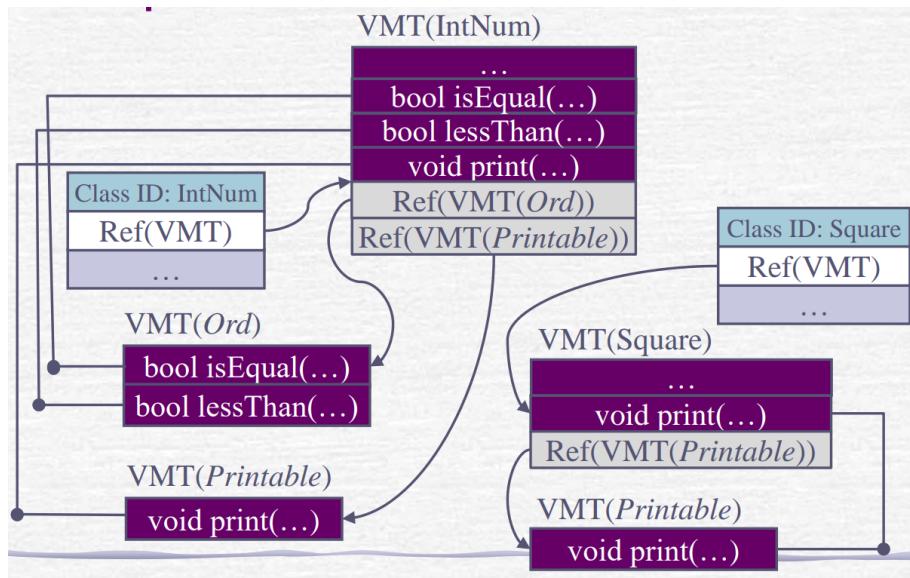
Rozhraní

Jedná se o **náhradu vícenásobné třídní dědičnosti a skutečného podtypu**. Rozhraní je schéma, které deklaruje sadu metod. **Nelze** vytvářet **instance rozhraní**, ale rozhraní se **přiřazuje třídám**, které jej implementují (třída, které je přiřazeno rozhraní, musí implementovat všechny metody, které rozhraní deklaruje). Deklarace atributů v rozhraní je pouze teoretická a není většinou podporována (v C# lze v rozhraní specifikovat tzv. properties s { get; set; }, to ale **nejsou atributy** jedná se vlastně o funkci, která pomocí get a set získává a nastavuje daný atribut - field v C#). **Rozhraní lze kombinovat** s jednoducho **dědičností** a třída může **současně implementovat více rozhraní**. Samotná rozhraní mohou také **dědit** (částečné uspořádání na rozhraních) a **případně i vícenásobně**.

Koncept **rozhraní** umožňuje **polymorfismus** i mimo třídní dědičnost. Libovolná třída může implementovat funkce daného rozhraní libovolně. Pokud poté přistupujeme k objektu pomocí rozhraní, je nám skryto, o jaký objekt se jedná, takže volaná metoda může mít libovolnou funkcionalitu (v rámci zachování logiky programu na to musíme dávat pozor, i.e. funkci implementujeme tak aby měla očekávanou funkcionalitu např. podle jména).

Implementace rozhraní

Objekt (instance třídy) odkazuje na více VMT, pro **každé rozhraní jiná VMT**, respektive z VMT objektu jsou odkazy na VMT realizující rozhraní (odkazující na metody, které deklaruje rozhraní).



Beztrídní objektově orientované jazyky

U beztrídních jazyků je tvorba objektů závislá na **klonování prototypů**. Prototyp je také objekt. Např. v JS je prototypem každého objektu **Object** a ten už nemá další prototyp. Pomocí prototypů se realizuje v JS **dědičnost**, každý objekt má jeden **přímý** prototyp, ten pak může mít další přímý prototyp atd. (tzv. **prototype chain**). Obecně ale může mít objekt více přímých předků (prototypů). **Specializace** objektů je prováděna **dynamicky** za běhu (případně by mohla být prováděna staticky před překladem) přidáním nové položky (**atributu** nebo **metody**). Klonováním prototypu se **nekopírují metody** klonovaného objektu. **Atributy** jsou obecně **zkopírovány** a je jim při klonování nastavena buď zadaná hodnota, nebo implicitní hodnota (hodnoty v prototypu). Metody zůstávají pouze u prototypu. Invokace těchto metod u naklonovaného objektu se řeší pomocí **delegace** (výpočetní systém se nejdřív snaží metodu spustit na daném objektu, pokud ji nemůže najít, deleguje ji na prototyp). **Polymorfismus** je zajištěn tak, že u naklonovaného objektu **definujeme stejně pojmenovanou metodu**. Např. JS je **dynamicky typovaný netypovaný** jazyk a implementuje **skutečný podtyp**, což umožňuje i polymorfismus i dědičnost. Pokud objekt neimplementuje danou metodu (nerozumí zprávě) a ani žádný z jeho prototypů, dochází k chybě. Příklad klonování z JS:

```
const personPrototype = {
  greet() {
    console.log('hello!');
  }
}

const carl = Object.create(personPrototype);
carl.greet(); // hello!
```

`personPrototype` je zde objekt (složené závorky je mimo jiné způsoby v JS syntax pro tvorbu objektu), jeho prototypem je tedy v tomto případě `Object` (vznikl jeho klonováním, i když to není na první pohled zřejmé). U tvorby objektu `carl` je již ale zřejmé, že se jedná o klonování objektu `personPrototype`. Objekt `carl` tak má za přímý prototyp `personPrototype` a ten má přímý prototyp `Object`. Voláním metody `greet` nad objektem `carl` vede na delegaci a volá se metoda `greet` objektu `personPrototype`.

Delegace

Delegace je obdoba zasílání zprávy. Zpráva je v tomto případě zasílána rodiči objektu (**příjemce** je rodič objektu) a kromě názvu metody (**selektoru**) a **parametrů** obsahuje i **objekt**, kterému byla zpráva prvně adresována (**původního příjemce**).

Sloty (v jazyce SELF)

Slot obsahuje **název položky** a **odkaz** na:

- **datový objekt**,
- **objekt s metodou**,
- **rodičovský objekt**.

Rysy (traits v jazyce SELF)

Rys je objekt, který obsahuje pouze sdílené **metody** a **rodičovské sloty** (neobsahuje jiné atributy/sloty), které umožňují delegaci. Prototyp pak obsahuje datové **sloty pro atributy** a jejich **implicitní hodnoty** (použité při klonování) a **rysy**, na které jsou **delegovány** zprávy (delegace) při volání "instančních" metod. Mezi **rysem a třídou** lze nalézt v tomto směru **analogie**, třída i rys nese instanční metody, které jsou z konkrétních instancí (objektů) odkazovány.

OO přístup k tvorbě SW

Při objektově orientované tvorbě SW využíváme skutečnosti, že **objekty reálného světa lze modelovat** (do jisté úrovně detailu) **objekty v SW**. A snažíme se využít základních vlastností OOJ, a to **abstrakci, zapouzdření, dědičnost a polymorfismus**. Např. v reálném světě potřebujeme vytvořit fakturu, což znamená, že dle nějakých konvencí ji musíme sepsat (např. vyplnit informace o dodavateli a odběrateli, položkách faktury, celkové ceně atd.). Fakturu poté můžeme odeslat buď poštou, nebo ji naskenovat a odeslat emailem.

Pokud by k nám přišel někdo, že už nechce psát faktury ručně a chce abychom mu pro to napsali SW můžeme postupovat následovně (posloupnost kroků nemusí odpovídat realitě):

1. Setkáme se s touto osobou (zákazník) a na základě jeho požadavků můžeme pomocí **UML** (standardizovaný grafický jazyk podporující objektovou orientaci) vytvořit objektově orientovaný návrh např. pomocí **diagramů tříd** a

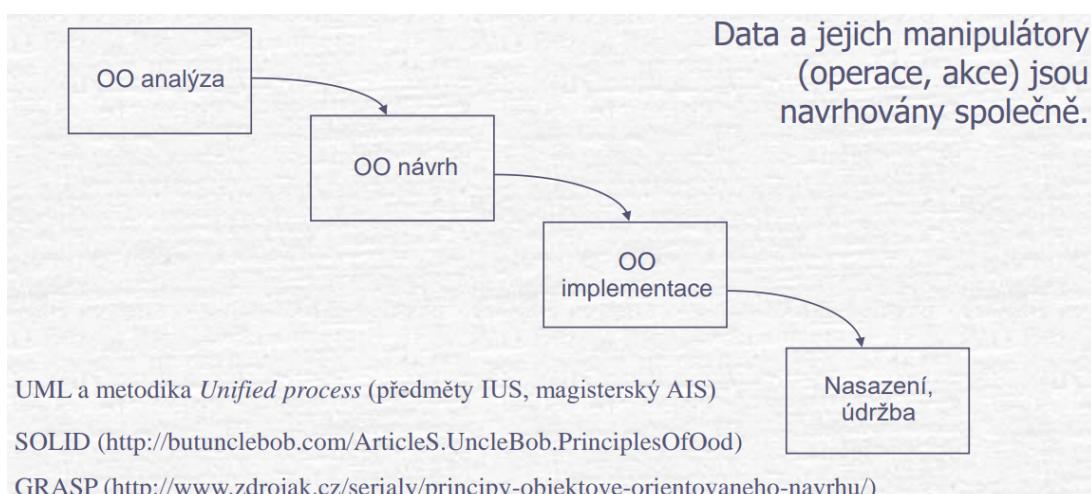
diagramů objektů.

2. Následně zjistíme, jestli již **neexistuje** nějaká objektová orientovaná implementace, která řeší tvorbu faktur na základě **vytvořených diagramů**. Pokud najdeme vhodné již implementované **třídy** (objekty u bezřídního jazyka), které osabují část atributů a metod. Můžeme použít tyto - využijeme jednu z výhod OOP, a to **znovupoužitelnost**.
3. Následně z vyhledaných objektů **zdědíme** (použijeme **dědičnost**) a do zděděného objektu **doplníme** potřebné atributy a metody (provedeme specializaci).
4. Metody, které jsou již v původním objektu implementovány, ale nevyhovují našemu zadání přepíšeme (**override**) a využijeme vlastnosti OOJ **polymorfismu**.
5. Pro ukládání faktur (tj. objektů faktura) můžeme využít buď **objektové databáze**, ale mnohem častější je využití SQL databází a vrstvy mezi databází a OOJ, která převádí relační data na objekty - **ORM (Objektově relační mapování)**

Shrnutí

Při OO přístupu k vývoji SW se snažíme využít:

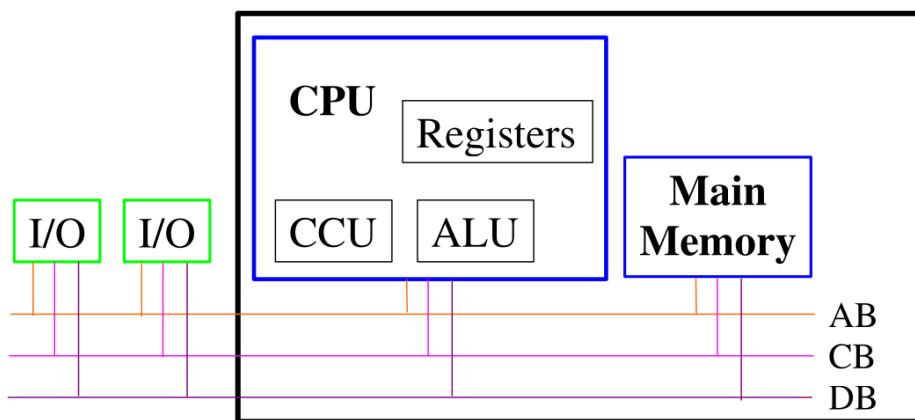
- **analogie** objektů reálného světa (mají nějaké znaky a funkce) se SW objekty (mají atributy a metody), Snažíme se zachovat vztah dat a jejich manipulátorů, respektive akcí nad nimi.
- při návrhu využíváme nástroje podporující objektový přístup, jako je jazyk **UML - formální reprezentace OOP**,
- využíváme **dekompozice** složitých objektů na jednodušší,
- při implementaci se snažíme využít **výhod OOP**, zejména **znovupoužitelnost**,
- při implementaci využíváme základní **koncepty OOP**, a to **abstrakci, zapouzdření, dědičnost a polymorfismus**.
- snažíme se využít nástroje, které umožňují převádět jiný SW na objektové paradigmá, např. **ORM**.
- snažíme se využít **návrhových vzorů** (poskytují radu/návod/vzor, jak vhodně řešit často vyskytujících se problémy a umožňují lepší znovupoužitelnost, např. singleton, factory, adapter, MVC, MVVM, MVP, Facade)



41. Programování v jazyku symbolických instrukcí (činnost počítače, strojový jazyk, symbolický jazyk, assembler)

Jazyk symbolických instrukcí je nízkoúrovňový programovací jazyk, který obsahuje **symbolické reprezentace strojových instrukcí**. Je definován výrobcem daného hardware, je založen na zkratkách jmen instrukcí (např. compare -> CMP)

Činnost počítače



CPU	Central Processing Unit (procesor)
ALU	Arithmetic and Logic Unit
CCU	Central Control Unit (řadič)
I/O	Input/Output Unit
AB, CB, DB	Address Bus, Control Bus, Data Bus

Hlavními částmi počítače jsou **CPU** (skládající se z aritmeticko logické jednotky (**ALU**), **registrů**, **řadiče** a **zdroj hodin**), **paměť** a **vstupně-výstupní zařízení**, všechny tyto komponenty jsou propojeny **sběrnicemi**. Významnými registry pro základní činnost počítače jsou **akumulátor** (střadač) pro výpočty, **instruction register** (IR) obsahující současnou instrukci a **instruction pointer** register (IP pro 16-bit a EIP pro 32-bit) ukazující na současnou instrukci v paměti. Paměť tak současně obsahuje:

- **data**, se kterými se pracuje,
- **instrukce**, které tuto práci (výpočet) řídí.

Princip činnosti PC

Princip činnosti počítače je následující (opakuje se v cyklu):

1. **fetch**: do **IR** se uloží obsah paměti, na který ukazuje **IP**,
2. aktualizuje se **IP** (na následující instrukci - přičte se počet bytů odpovídající provedené instrukci),
3. **decode**: dekóduje se IR, určí se operace a operandy,
4. **execute**: provede se daná instrukce (skoky ještě aktualizují **IP**).

V případě přerušení je obsah některých registrů (instruction pointer, flags, ...) nahrán na zásobník a po obsluze přerušení je stav těchto registrů ze zásobníků obnoven.

Strojový kód

Kód specifický pro daný počítač, **binární jedničky a nuly**, není téměř vůbec přenositelný, v dnešní době nemožné v něm programovat. Výhodou je vysoká efektivita, nevýhodou **nepřenositelnost** a **složitost psaní**. Právě z těchto důvodů se zavádí symbolický jazyk, který je čitelnější, ovšem stále má víceméně 1:1 mapování na strojový kód (tedy výhodu efektivity) a poté dále vyšší programovací jazyky, které díky překladačům zajišťují přenositelnost.

Symbolický jazyk

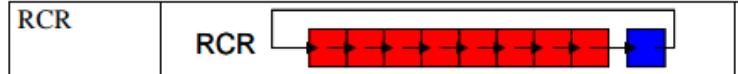
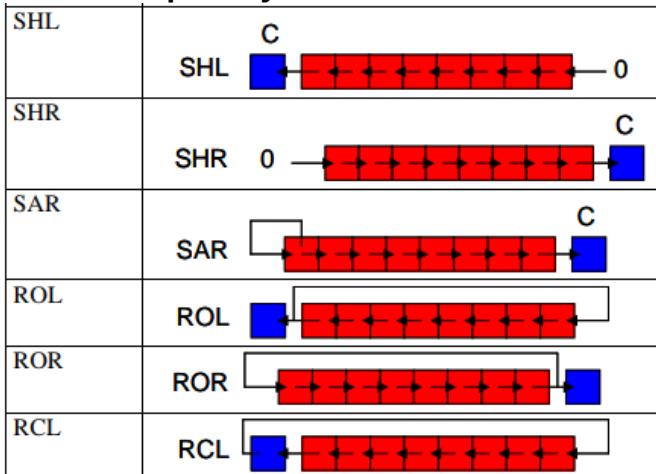
Také jazyk symbolických instrukcí.

Instrukce

Instrukce jsou **příkazy pro procesor**, obsahují jednoznačný **operační kód** a operandy, příp. adresy operandů. Některé instrukce mohou mít ve své specifikaci napevno nastavené, s kterými registry pracují. Základními typy instrukcí jsou:

- **přenosové**: MOV, PUSH, POP,
- **aritmetické**: ADD, SUB, INC,
 - MUL - pozor, 32b*32b=64b, využívá se EDX:EAX pro výsledek; IMUL - znaménkové násobení,
 - DIV - umí zase dělit 64 bit číslo z EDX:EAX v případě 32-bit musíme rozšířit znaménko do EDX, jinak to nebude fungovat, výsledek je v EAX, v EDX je pak modulo - zbytek, IDIV - znaménkové dělení.

- posuvy a rotace: SHL, ROL, ...



- logické: AND, OR, NOT, TEST, XOR, ...
- skokové: JMP, JA, LOOP - skáče podle hodnoty counteru ECX - rozdílné od nuly znamená skoč, jinak ne. Skoky mohou prováděny **podmíněně** dle příznaků (např. přetečení).
- skoky pro znaménkovou aritmetiku

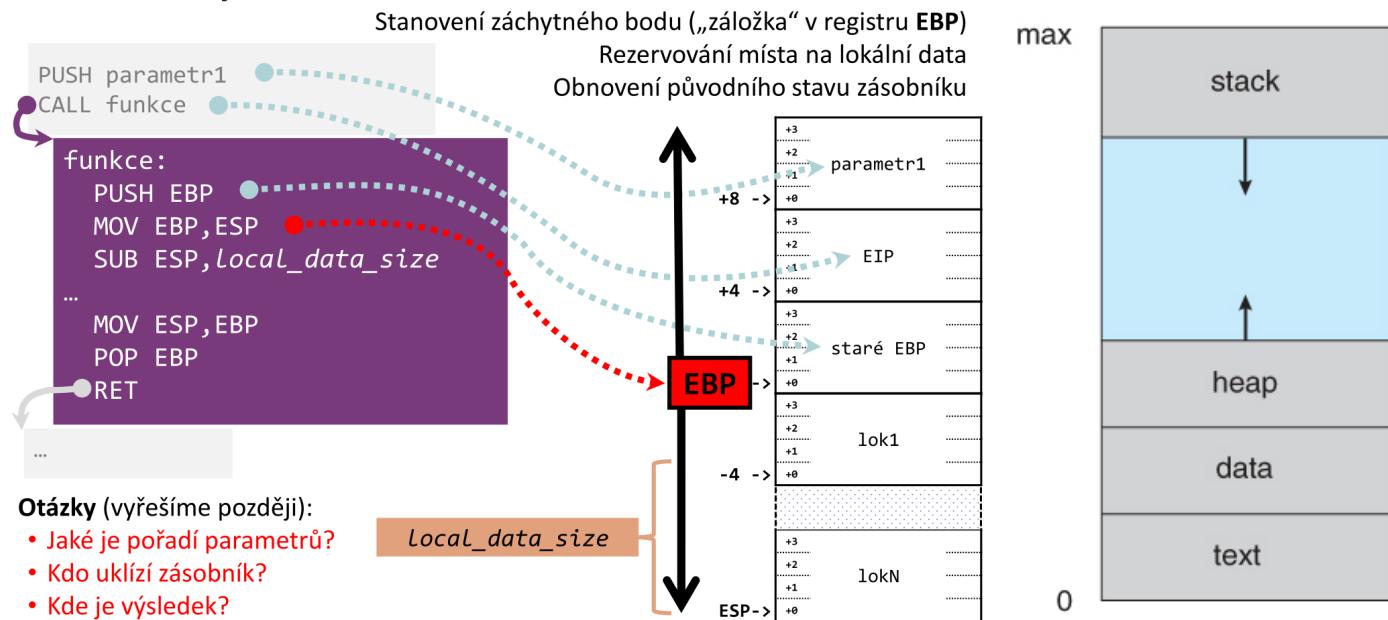
Mnemonic	Description	Condition Tested
JG, JNLE	Jump if Greater, Jump if Not Less or Equal	ZF = 0 and SF = OF
JGE, JNL	Jump if Greater or Equal, Jump if Not Less	SF = OF
JL, JNGE	Jump if Less, Jump if Not Greater or Equal	SF ≠ OF
JLE, JNG	Jump if Less or Equal, Jump if Not Greater	ZF = 1 or SF ≠ OF

Mnemonic	Description	Condition Tested
JA, JNBE	Jump if Above, Jump if Not Below or Equal	ZF = 0 and CF = 0
JAE, JNB	Jump if Above or Equal, Jump if Not Below	CF = 0
JB, JNAE	Jump if Below, Jump if Not Above or Equal	CF = 1
JBE, JNA	Jump if Below or Equal, Jump if Not Above	ZF = 1 or CF = 1

- řetězové: umožňují jednodušší a rychlejší práci s poli (automaticky inkrementují/dekrementují hodnotu ukazatelů v ESI a EDI). Lze je kombinovat s prefixy REP* pro opakování dle ECX.
 - MOVS(B/W/D) - přesun hodnot odkazovaných z ESI do EDI (B po bytech, W po slovech a D po 4 bytech),
 - CMPS(B/W/D) - porovnání hodnot odkazovaných ESI a EDI,
 - SCAS(B/W/D) - porovnání hodnoty EAX s hodnotou odkazovanou EDI, lze využít pro hledání prvku v poli,
 - LODS(B/W/D) - načtení z adresy odkazované ESI do akumulátoru,
 - STOS(B/W/D) - uložení akumulátoru (AL/AX/EAX) na adresu

odkazovanou z EDI.

- **řídící:** CALL, RET (umožňuje odebrat n slabik ze zásobníku)
 - CALL je skok, co uloží na zásobník **návratovou adresu** (RET si ji pak vezme a vrací řízení za instrukci CALL)
 - Parametry se předávají v **globálních proměnných, registrech** nebo na **zásobníku** (dle konvence volání), či kombinací.
 - Zásobníkový rámec je typická konstrukce ve volání funkcí, tvoří se "záložka" pro jednoduché vyčištění např. lokálních proměnných ze zásobníku (vyčištění znamená změnu hodnoty ukazatele na vrchol zásobníku v registru **ESP**, uložená data tam zůstávají, dokud nejsou přepsána dalším použitím zásobníku - nikdy tato data ale už nelze použít z důvodu možného vzniku přerušení). Register **EBP** pak funguje jako reference, abychom věděli, jak moc věcí jsme už na zásobník zapsali. Při vstupu do funkce **uložíme starou referenci** (cizí referenci toho, kdo funkci volal) z **EBP** na zásobník a vytvoříme si novou referenci - aktuální vrchol zásobníku **ESP**. Místo, které budeme na proměnné potřebovat pak vyhradíme odečtením počtu bytů od **ESP**, čímž se posune vrchol zásobníku a vytvoří se požadované místo (odečítá se, protože zásobník je vzhůru nohama a jeho dno je na maximální adrese). Před výstupem z funkce pak musíme obnovit původní obsah registrů **ESP** a **EBP**.



Obecně mají instrukce **proměnnou délku** (ta je např. dáná způsobem adresování operandů nebo jejich počtem). Formát instrukce na procesoru intel Pentium je takový (délka může být od **1 B** do **16 B**):

Předpony	Operační kód	ModR/M	SIB	Posunutí	Operand
0-4 slabiky	1-2 slabiky	0-1 slabika	0-1 slabika	0/1/2/4 slab.	0/1/2/4 slab.

Konvence volání

- pascal** - parametry uklízí **volaný** (tj. uvnitř funkce), parametry se dávají na zásobník **zleva doprava**, tzn. na vrcholu zásobníku je při předání řízení do funkce hodnota **posledního parametru** (využito v Pascalu),
- cdecl** - parametry uklízí **volající** (tj. až po návratu za funkcí), předávají se **zprava doleva**, tzn. na vrcholu zásobníku je po předání řízení do funkce **první parametr**. Tento způsob umožňuje funkce s proměnným počtem parametrů - první parametr specifikuje jejich počet (jazyk C),
- stdcall** - parametry uklízí **volaný**, předávají se **zprava doleva** (winapi)

Registry

Máme mnoho registrů pro různé účely různých velikostí. Základní datové registry (například střadač), mají více možností, jak se odkazovat na jejich část, např. EAX = celý 32b registr, AX = pouze spodních 16b, AH = vrchní 8b AX, AL = spodních 8b AX

EAX (<i>Accumulator</i>)	střádač
EBX (<i>Base</i>)	ukazatel na data v datovém segmentu
ECX (<i>Counter</i>)	čítač řetězových a smyčkových operací
EDX (<i>Data</i>)	rozšíření střádače, ukazatel na I/O zařízení
ESI (<i>Source Index</i>)	ukazatel na zdrojová data řetězových operací, index
EDI (<i>Destination Index</i>)	ukazatel na cílová data řetězových operací, index
EBP (<i>Base Pointer</i>)	ukazatel na data v zásobníkovém segmentu
ESP (<i>Stack Pointer</i>)	ukazatel zásobníku

Základními registry jsou: EAX, EBX, ECX, EDX, pro práci se zásobníkem se využívá ESP (stack pointer), EBP (base pointer), řetězové instrukce využívají ESI (zdrojová data) a EDI (cílová data). Speciálním je registr **EFLAGS** obsahující příznaky, které se nastavují dle operací:

- **OF** (overflow flag) = 1 při přetečení znaménkové operace, jinak 0,
 - **CF** (carry flag) = 1 při přetečení bezznaménkové operace, jinak 0,
 - **SF** (sign flag) = 1 při záporném výsledku
 - **ZF** (zero flag) = 1 když je výsledek 0
 - **DF** (direction flag) = určuje směr řetězových instrukcí - směr průchodu polem (nastavujeme instrukcemi **CLD** - clear direction flag - nastaví na 0, což znamená zvyšování ukazatele na data po každé iteraci, **STD** - set direction flag - nastaví na 1, což způsobí snižování ukazatele na data po každé iteraci)

DF = clear(0); increment SI a DI

DF ≡ set (1): decrement SJ a DI

Skoky

- **Short** – 2-bytová instrukce, která dovoluje skočit na místo v rozsahu +127 and -128 byte od místa, které následuje za příkazem JMP.
 - **Near** – 3-bytová instrukce, která dovoluje skočit v rozsahu +/- 32KB od následující instrukce v rámci běžného kódového segmentu.
 - **Far** – 5-bytová instrukce, která dovoluje skočit na jakékoli místo v celém adresovém prostoru.

Paměť'

V základním režimu (16b) má fyzická adresa 20b, tedy pro adresaci pomocí registrů chybí 4b. K řešení tohoto problému se vytváří **segmenty** (úseky paměti o velikosti 64 KiB, které lze adresovat 16 bitů), každý segment začíná na adrese, která je násobkem 16 (je **posunutá** o 4 bity doleva). Fyzická adresa je pak kombinací **segment:offset** (ReqReg:Offset), kdy **fyzická adresa = 16 * segment + offset**.

Little a Big Endian



FPU

Dosud tato otázka řešila pouze operace v ALU, tj. integer a unsigned aritmetika, pro operace s floating point (IEEE754 viz jiné otázky) se používají specializované jednotky (koprocesory), například **FPU** (floating point unit).

Registry FPU pracují v **zásobníkovém režimu**, vršek zásobníku je ST0. Instrukce jsou prefixovány písmenem F, například **FADD**, **FABS**, **FSIN** a mají mnoho možností operandů (implicitní operandy, explicitní operandy, popnutí zásobníku registrů atd.). Např. **FADD st1** provede **st0 = st0 + st1**.

Assembler

Assembler je překladač jazyka **symbolických instrukcí do strojového kódu**, příkladem je **NASM**. Aby mohl být proveden překlad, potřebuje nejen samotné instrukce, ale i dodatečné informace, typicky v podobě **pseudoinstrukcí** nebo **direktiv** (direktiva je příkazem pro překladač, nikoliv instrukcí pro procesor a nejsou překládány do strojového kódu). Jedná se především o:

- Definování konstant.
- Definování místa v paměti pro uložení dat (RESB, RESW, RESQ).
- Vytváření segmentů paměti.
- Vkládání jiných souborů (INCLUDE).

Pseudoinstrukce a direktivy tak popisují strukturu paměti, například kde **začíná** a **končí program** (SECTION .text), **definici dat** (inicializovaná - DB, DW, ... a neinitializovaná - RESB, RESW, ...), **definici konstant**, **definice jmen** (GLOBAL, EXTERN), příp. také mohou být podporována makra, která překladač textově

expanduje (jako v C) nebo nějaký další pre-processing.

Překladač dělá dvouprůchodový překlad, nejprve sestavuje **tabulku symbolů** (ukládá adresy návěstí a jmen), která následně při druhém průchodu využije (symbol musí být při druhém průchodu **jednoznačně** definován). Dvojí průchod je nutný například kvůli **dopředným** skokům.

Makra

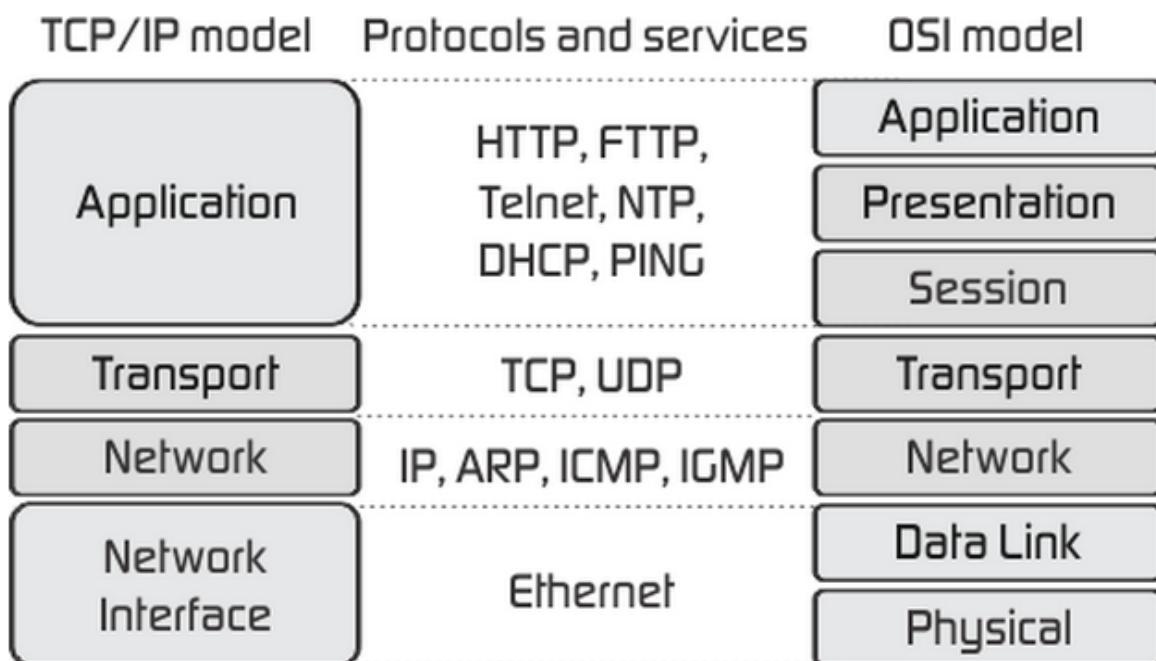
Makro je mechanismus, kterým můžeme nadefinovat posloupnost příkazů, vykonávající určitou činnost a tuto posloupnost příkazů **později vložit na různá místa programu**. Překladač tedy kód obsažený v makru kopíruje na všechna místa jeho použití. Nachází se tedy ve výsledném strojovém kódě opeakováně stejně posloupnosti jedniček a nul. Na rozdíl od funkcí, které jsou definované pouze na jednom místě a skáče se tam pomocí CALL a RET.

- **výhody**: rychlejší než funkce (nemusí se nikam skákat a vytvářet rámec),
- **nevýhody**: větší program (více kódu, což může mít špatný vliv na výkon, viz stránkování).

42. Služby aplikační vrstvy (web, e-mail, DNS, IP telefonie, správa SNMP, Netflow).

V modelu TCP/IP i v modelu ISO/OSI je aplikační vrstva poslední. Referenční model ISO/OSI předpokládá, že jednotlivé aplikace budou mít některé společné rysy, které se vyplatí realizovat samostatně a implementovat jen jednou. Síťový model vznikal více z praktických zkušeností. Předpokládá, že jednotlivé aplikace nebudou mít totík společného, aby se tyto části vyplatilo osamostatnit. Na rozdíl od referenčního modelu ISO/OSI očekává, že každá aplikace si sama zajistí to, co potřebuje a co jí nenabízí nižší vrstvy.

Na rozdíl od nižších vrstev (transportní, síťová a vrstva síťového rozhraní) není komunikace na aplikační vrstvě zajištěna OS (sockets) či HW počítače. Každá aplikace si musí komunikaci na této vrstvě zajišťovat sama. Proto, aby se nelišily způsoby komunikace aplikace od aplikace, používají se na aplikační vrstvě (stejně jako na jiných) standardizované protokoly. Protokol, který aplikace pro komunikaci používá, a určuje její změření.



Na základě požadavků aplikace může být aplikační protokol implementován nad UDP (rychlejší, ztrátový), nebo nad TCP (pomalejší, bezztrátový), příklady protokolů:

- **UDP:** **DNS** (překlad doménových jmen na IP), **TFTP** (přenos souborů), **SNMP** (správa sítě), **Netflow** (správa sítě), **NTP** (synchronizace času), **SIP** (signalizace VoIP), **RTP** (přenos zvuku a obrazu), **RTCP** (řídící informace pro RTP a QoS),
- **TCP:** **HTTP** (webové stránky), **SMTP** (mailtová komunikace), **LDAP** (adresářové služby, může být i přes UDP), **FTP** (přenos souborů), **POP3**

(stahování el. pošty), **IMAP** (stahování a čtení el. počty).

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1 Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Instant messaging	No loss	Elastic	Yes and no

Adresa

Způsob identifikace adresáta pomocí jednoznačné informace.

Web (HTTP)

World Wide Web (také jen web) je tvořen aplikacemi, které běží na protokolu **http** (Hypertext Transfer Protocol), respektive jeho šifrované verzi **https** (Hypertext Transfer Protocol Secure). WWW je systém pro **prohlížení, ukládání a odkazování dokumentů** nacházejících se **na internetu**. Dokumenty (webové stránky) jsou uloženy na **webových serverech** a přistupujeme k nim pomocí **webového prohlížeče**. Navzájem jsou propojeny pomocí **hypertextových odkazů** zapisovaných ve formě **URL**.

URI, URL a URN

URL a URN jsou podmnožinou URI. URI je textový řetězec s definovanou strukturou, který slouží k **přesné specifikaci zdroje** informací.

- **Uniform Resource Name** (URN): jasně identifikuje zdroj, ale neřeší jeho dostupnost. Schema URN je $\text{URN} ::= \text{"urn:}" \text{ <NID>} \text{ ":" } \text{<NSS>}$, např. `urn:isbn:0451450523`
- **Uniform Resource Locator** (URL): určuje, kde je identifikovaný zdroj dostupný a mechanismus pro jeho získávání. **Adresa** URL definuje, **jak lze zdroj získat**. Nejobecněji má URL tvar $\text{<scheme>} : \text{<scheme-specific-part>,}$ konkrétně pak `protokol://server.doména2.doména1:port/cesta/název?dotaz#kotva.`

Pro adresaci (jako **adresu**) na webu **používáme URL**, i když v HTTP RFC je to zobecněno na URI (ale pomocí URN zdroj na internetu nenajdeme).

HTTP

HTTP je protokol typu **klient-server**, což znamená, že požadavky **iniciuje příjemce** (klient), obvykle webový prohlížeč. Klienti a servery komunikují výměnou jednotlivých zpráv (na rozdíl od proudu dat). Klient zasílá požadavky/dotazy (**request**) a server na ně odpovídá odpověďí (**response**). Jedná se o **bezstavový** protokol, tj. že není zachována návaznost jednotlivých dotazů a odpovědí (řeší se ale pomocí Cookies).

Schéma HTTP dotazu

```
<metoda> <cesta> <verze protokolu>CRLF  
<hlavička 1>CRLF  
<hlavička 2>CRLF  
<hlavička...>CRLF  
<hlavička n>CRLF  
CRLF  
<tělo dotazu>
```

- **metoda** je:
 - **GET** - získání zdroje/dat (nemělo by se pomocí GET nikdy modifikovat),
 - **POST** - odeslání dat na server, mělo by znamenat vytvoření zdroje,
 - **PATCH** - částečná úprava zdroje,
 - **PUT** - změna aktuálního zdroje s daty v těle dotazu,
 - **DELETE** - smazání zdroje,
 - aj. (**HEAD, CONNECT, TRACE**)
- **cesta** specifikuje umístění zdroje na serveru,
- **verze protokolu** je např. **HTTP/1.1**,
- **hlavičky** jsou ve tvaru název: hodnota, nejdůležitější hlavičkou je hlavička Host, která specifikuje adresu serveru.
- **tělo** dotazu je tvořeno daty, které odesíláme na server, může jít o JSON, HTML, XML, binární data, ... specifikováno pomocí hlavičky **Content-Type** a **MIME** (application/json, text/html, application/xml) typu data.

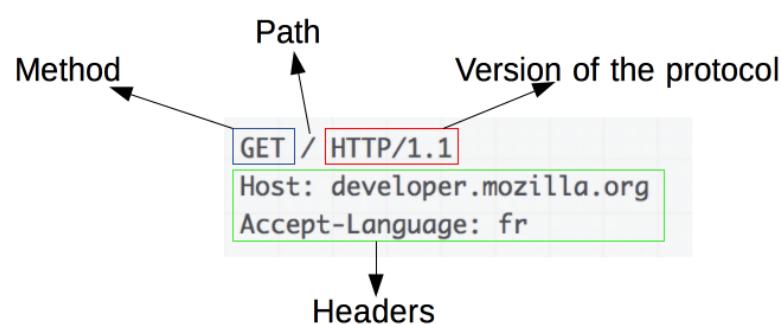
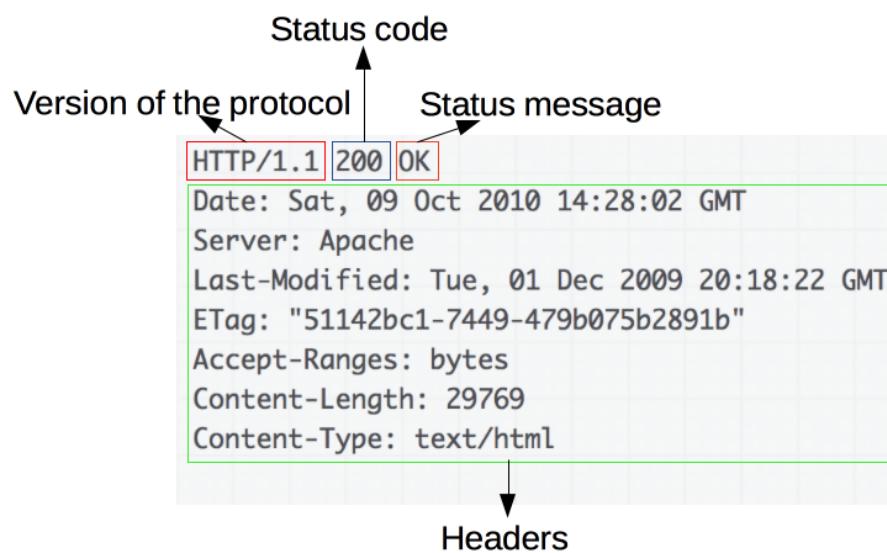


Schéma HTTP odpovědi

```
<verze protokolu> <status code> <status message>CRLF  
<hlavička 1>CRLF  
<hlavička 2>CRLF  
<hlavička...>CRLF  
<hlavička n>CRLF  
CRLF  
<tělo dotazu>
```

- **verze protokolu** je např. **HTTP/1.1**,
- **status code** je jeden ze skupiny:
 - **100-199**: **informační odpověď**,
 - **200-299**: odpověď hlásící **úspěch**,
 - **300-399**: odpověď hlásící **přesměrování**,
 - **400-499**: odpověď hlásící **chybu na straně klienta**,
 - **500-599**: odpověď hlásící **chybu na straně serveru**.
- **status message** dále upřesňuje status code, např.:
 - 200 OK,
 - 201 created,
 - 202 updated,
 - 400 bad request,
 - 401 unauthorized,
 - 403 forbidden,
 - 404 not found.



E-mail

Email je mechanismus pro zasílání elektronické pošty. Je tvořen:

- **obálkou** - tu vytváří poštovní **server** (MTA - Mail Transfer Agent) a obsahuje:
 - MAIL FROM:<odesilate@domena.com>

- RCPT TO:<prijemce@domena.com>
- **hlavičkou** - vytváří poštovní **klient** (MUA - Mail User Agent), má tvar <název>:<hodnota> a může obsahovat (mimo jiné):
 - From, To, Subject, Cc, Bcc, Date, Return-Path, Received, ...
- **tělem** - vytváří poštovní klient a obsahuje **7-bit ASCII** data (původně, hlavička a obálka jsou také 7-bit), dnes rozšíření **MIME**.

K identifikaci adresáta (**adresa**) se u emailu používá **emailová adresa**.

Kódování

V případě emailu se jedná o **převod** binárních nebo jiných **8-bit** znaků na **7-bit ASCII** znaky. Rozhodně se **nejedná o šifrování**.

Quoted-printable

Nahrazují se pouze 8-bit znaky a převádějí se na **3 znaky**, a to rovnítko a 2 **hexadecimální znaky** (reprezentují index v 8-bit poli znaků), např. =F9 nebo =B9. Rovnítko musí být také kódováno jako (=3D). Teoreticky může dojít k prodloužení textu až o **200%**. Prakticky se používá při kódování textu (jazyka, např. čj, zůstává částečně čitelné), kde může mít menší prodloužení než **Base64** (záleží také na národní abecedě), nevhodné pro binární data.

Base64

Kódují se vždy všechny znaky, a to po trojicích. **Trojice 8-bit** znaků je převedena na **čtveřici 6-bit** znaků ($3 \times 8 == 4 \times 6$). Šestibitové znaky jsou následně převáděny na 7-bitové indexací do pole:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/. Chybějící znaky do čtveřice jsou doplnovány pomocí rovnítka. Text je vždy prodloužen o **33%**, kódování je vhodné pro binární data.

Simple Mail Transfer Protocol (SMTP)

Aplikační protokol nad TCP pro **posílání pošty** na portu **25**. S protokolem SMTP pracují 3 programy:

- **MUA – Mail User Agent**: poštovní klient, který zpracovává zprávy u uživatele.
- **MTA – Mail Transfer Agent**: server, který se stará o doručování zprávy na cílový systém adresáta. Ignoruje hlavičku a tělo zprávy, pro doručování **používá pouze obálku** (analogie s poštou).
- **MDA – Mail Delivery Agent**, program pro lokální doručování, který umísťuje zprávy do uživatelských schránek, případně je může přímo automaticky zpracovávat (ukládat přílohy, odpovídat, spouštět různé aplikace pro zpracování apod.).

Pro vyhledání serveru adresáta, na který má být předána zpráva, vyhledává MTA záznam **MX v DNS**. Dotaz na DNS obsahuje **doménové jméno** za znakem '@', DNS vrací **IP** SMTP serveru adresáta.

SMTP příkazy

HELO, HELP, MAIL FROM, RCPT TO, RSET, QUIT, ...

Post Office Protocol (POP3)

Protokol sloužící pro stahování elektronické pošty přes TCP port 110. Protokol je mnohem **jednodušší** než IMAP a umožňuje především **stahovat** poštu ze serveru (a zobrazovat počet příchozích zpráv). Po stáhnutí je pošta ze serveru **implicitně smazána** (může být nastaveno jinak, třeba po určité době bude pošta smazána). Implicitní mazání neumožňuje mít jednu schránku na více zařízeních a i pokud je nastaveno jinak, není protokol vhodný pro více schránek (např. přesun zpráv, označení zpráv atd. se děje pouze lokálně, na serveru se nic nemění). Protokol je vhodný, pokud na serveru máme **málo místa**.

Internet Mail Access Protocol (IMAP)

Protokol sloužící pro **prohlížení** a stahování elektronické pošty přes TCP port 143. Pomocí protokolu se navazuje s emailovým serverem trvalé spojení. Zprávy není nutné stahovat celé, stačí pouze základní informace (hlavičky zpráv). Zprávu je ze serveru stažena celá, až když ji chce uživatel na klientovi číst (poté může být zase smazána). Na **serveru zprávy zůstávají neustále**, dokud je uživatel explicitně nezmaže. IMAP **reflektuje změny** provedeny uživatelem v klientovi **na server** (např. označení zprávy za přečtenou, přesun zprávy do jiné složky, přidání/odebrání štítku atd.). Je proto **vhodný** při přístupu z **více klientů** (zařízení), změny provedené na jednom jsou synchronizovány na druhý.

Pretty Good Privacy (PGP)

Protokol SMTP a IMAP lze přenášet zabezpečeně přes internet (SSL, TLS, HTTPS). Tyto protokoly ale **zabezpečují obsah zprávy**, ta zůstává na emailovém serveru v čitelné formě. Zabezpečit tělo zprávy lze např. pomocí **PGP**. PGP zajišťuje:

- **integritu dat**,
- **autentizaci odesílatele** (autentizace a integrita dat zajišťují společně **neodmítnutelnost**),
- **šifrování**.

Postup zabezpečení:

1. Odesílatel vypočte **heš** zprávy a svým privátním klíčem ji **podepíše**.
2. **Komprimuje** obsah zprávy.
3. **Zašifruje** obsah zprávy vygenerovaným symetrickým klíčem.
4. Symetrický klíč **zašifruje veřejným** klíčem příjemce (dešifrovat může pouze příjemce svým privátním) a připojí jej k zašifrované zprávě.
5. Převede obsah do **Base64** (v mailu mohou být pouze 7-bit znaky, viz výše).

Secure/Multipurpose Internet Mail Extensions (S/MIME)

Další možnost zabezpečení obsahu emailu založená na **asymetrické kryptografii** a

digitálních podpisech.

Domain Name System (DNS)

DNS je **globální decentralizovaný adresář** názvů počítačů a dalších identifikátorů síťových zařízení a služeb. Hlavní funkcí DNS je **překlad doménových jmen na IP adresy**. DNS rozděluje globální prostor doménových adresy (jmén) **hierarchicky**, což umožňuje delegaci. Záznamy jsou uloženy na třech typech serverů, a to **primární, sekundární a záložní**. Princip vyhledávání v DNS se nazývá **rezoluce**. Protokol DNS pro rezoluci používá primárně UDP, port 53. TCP lze použít při přenosu většího množství dat, např. při přenosu DNS zón.

Prostor doménových jmen

Hierarchické uspořádání DNS záznamů do stromové struktury.

- Kořenovou doménu “.” má na starost organizace **ICANN**.
 - Top Level Domain - TLD (domény 1. řádu): zprávu těchto domén deleguje ICANN na další organizace, v ČR je to CZ.NIC. Jde o domény (.cz, .com, .eu, .org, .net, ...).
 - Domény 2. a nižších řádů: tyto domény spravují konkrétní organizace. Např. u CZ.NIC si může každý zakoupit doménu *.cz (* je název 2. řádu) a poté ji spravovat a případně prodávat domény *.*.cz. Jde o domény (seznam.cz, vut.cz, google.com, ...)

Reverzní mapování

Umožňuje na základě **IP** adresy vyhledat doménové jméno. Reverzní mapování zajišťují 2 podstromy **in-addr.arpa**. pro IPv4 a **ipv6.arpa**. pro IPv6. Pro IPv4 adresy se zapisuje běžným zápisem ale **zprava doleva**. U IPv6 se každá hexadecimální číslice odděluje tečkou, zápis je také **zprava doleva**.

Záznamy DNS

Existuje velké množství DNS záznamů, zde jsou ty nejdůležitější:

- **SOA:** zahajuje záznam zónového souboru, každá zóna má **právě jeden** záznam SOA. Obsahuje mimo jiné název primárního serveru a emailovou adresu správce.

```
> dig SOA fit.vutbr.cz
fit.vutbr.cz. 14386 IN SOA guta.fit.vutbr.cz. ; primary DNS server
                                         michal.fit.vutbr.cz. ; email to the responsible person
                                         202110180          ; Serial number
                                         10800              ; Refresh - 3 hodiny
                                         3600               ; Retry - 1 hodina
                                         691200             ; Expire - 8 dní
                                         86400              ; Minimum - 1 den
```

- **NS:** určuje **autoritativní** server pro danou doménu.

```
> dig NS fit.vutbr.cz
```

fit.vutbr.cz.	7012	IN	NS	gate.feec.vutbr.cz.
fit.vutbr.cz.	7012	IN	NS	kazi.fit.vutbr.cz.
fit.vutbr.cz.	7012	IN	NS	guta.fit.vutbr.cz.
fit.vutbr.cz.	7012	IN	NS	rhino.cis.vutbr.cz.

- **A:** přímé mapování doménového jména na IPv4 adresu.

```
> nslookup isa.fit.vutbr.cz
```

```
isa.fit.vutbr.cz. 14347 IN A 147.229.176.18
```

- **AAAA:** přímé mapování doménového jména na IPv6 adresu.

```
> dig AAAA www.cesnet.cz
```

```
www.cesnet.cz. 3600 IN AAAA 2001:718:1:1f:50:56ff:feee:46
```

- **MX:** Slouží pro zjištění adresy poštovního serveru (ta může být opět jako doménové jméno), může obsahovat více záznamů rozlišených prioritou (menší číslo značí vyšší prioritu).

```
> dig MX stud.fit.vutbr.cz
```

stud.fit.vutbr.cz.	10384	IN	MX	10 eva.fit.vutbr.cz.
stud.fit.vutbr.cz.	10384	IN	MX	20 kazi.fit.vutbr.cz.

- **CNAME:** slouží jako alias, mapuje jej na jméno počítače (síťového zařízení), aliasy nesmí být na pravé straně záznamů a PC jich může mít více.

```
> dig email.fit.vutbr.cz
```

email.fit.vutbr.cz.	14400	IN	CNAME	hermina.fit.vutbr.cz.
hermina.fit.vutbr.cz.	3907	IN	A	147.229.9.15

- **PTR:** mapuje IPv4 nebo IPv6 adresu na doménovou adresu, obsahuje **reverzní** mapování. Využívají speciální podstromy **in-addr.arpa.** pro IPv4 a **ipv6.arpa.** pro IPv6.

```
> dig -x 147.229.9.23
```

```
23.9.229.147.in-addr.arpa. 14400 IN PTR www.fit.vutbr.cz.
```

```
> dig -x 2001:67c:1220:809::93e5:917
```

```
7.1.9.0.5.e.3.9.0.0.0.0.0.0.0.9.0.8.0.0.2.2.1.c.7.6.0.1.0.0.2.ip6.arpa.
14400 IN PTR www.fit.vutbr.cz.
```

- **SRV**: slouží k lokalizaci služeb a serverů, např. SIP. Mají tvar `_service._protocol.domain_name.`
- ```
> dig SRV _sip._udp.cesnet.cz
_sip._udp.cesnet.cz. 1706 IN SRV 100 10 5060 cyrus.cesnet.cz.

> dig SRV _ldap._tcp.zcu.cz
_ldap._tcp.ZCU.CZ. 600 IN SRV 0 100 389 pleiades.pool.zcu.cz.

> dig SRV _ldap.vutbr.cz
_ldap._tcp.vutbr.cz. 14313 IN SRV 0 100 389 dcb.vutbr.cz.
_ldap._tcp.vutbr.cz. 14313 IN SRV 0 100 389 dca.vutbr.cz.

• a další, dohromady asi 91 DNS záznamů.
```

### Přehled DNS záznamů

| Záznam | Mapování                   | Příklad                                                                                         |
|--------|----------------------------|-------------------------------------------------------------------------------------------------|
| A      | DNS adresa → IP adresa     | tereza.fit.vutbr.cz → 147.229.9.22                                                              |
| AAAA   | DNS adresa → IPv6 adresa   | www.cesnet.cz. → 2001:718:1:101::4                                                              |
| NS     | doména → doménový server   | fit.vutbr.cz. → gate.feec.vutbr.cz.                                                             |
| MX     | doména → poštovní server   | fit.vutbr.cz. → kazi.fit.vutbr.cz.                                                              |
| SOA    | doména → info o zóně       | fit.vutbr.cz. → guta.fit.vutbr.cz.<br>michal.fit.vutbr.cz. 202010201<br>10800 3600 691200 86400 |
| CNAME  | DNS adresa → DSN adresa    | www.vutbr.cz. → piranha.ro.vutbr.cz.                                                            |
| SRV    | služba → DNS adresa + port | _sip._udp.cesnet.cz → cyrus.cesnet.cz. + 5060                                                   |
| PTR    | IP adresa → DNS adresa     | 22.9.229.147.in-addr.arpa. → tereza.fit.vutbr.cz.                                               |
| PTR    | IPv6 adresa → DNS adresa   | 4.0.0.0.0.0.0.0.0.0.0.1.0.1.0.0.0.8.1.7.0.1.0.0.2.ip6.arpa. → www.cesnet.cz.                    |

### DNS rezoluce

Jedná se o proces vyhledání odpovědi v systému DNS. Komunikace DNS je typu klient-server. Dotazy aplikací (klientů) vyřizuje systémová rutina OS tzv. resolver (protože OS může některé DNS záznamy mít ve VP a také aby to každá aplikace nemusela implementovat). Využívá stromovou strukturu jmen a kořenové DNS servery (těch je 13 a obsahují kořenovou zónu). Používají se dva způsoby rezoluce. Pro to, aby počítač mohl provést DNS rezoluci, musí nejdříve znát IP adresu DNS serveru. Ta může být zadána buď ručně (staticky) nebo jí získá od **DHCP** (IPv4) serveru (ten přiděluje: **IP adresu, masku sítě, implicitní bránu** (default gateway) a **adresu DNS serveru**). Adresu **IPv6 DNS** serveru lze získat pomocí **Router Solicitation** zprávy a odpovědi **Router Advertisement** (ICMPv6), která mimo jiné může vracet IP adresu DNS serveru.

## Rekurzivní dotaz

Rekurzivní dotaz provádí **rekurzivní server**, který po dotazu vždy vrátí IP adresu (pokud existuje). Při dotazu na rekurzivní server mohou nastat tyto situace:

- rekurzivní server **zná** doménové jméno (má ho v zónovém souboru nebo cache) a vrátí ho klientovi,
- rekurzivní **nezná** doménové jméno a dotazuje se jiného DNS serveru (pokud nezná žádnou část doménového jména, tak je to **kořenový server** - zjistí ze zóny **hint** (součást instalace), jinak může volat autoritativní server pro danou část doménového jména, např. při dotazu na **stud.fit.vutbr.cz**. zná již **vutbr.cz.**, tak se ptá tam), jiný DNS server může být:
  - **rekurzivní**, který vrací přímo **IP** (a dotazující se server si ji může uložit do cache),
  - **iterativní**, který buď vrací **IP**, pokud ji zná, nebo vrací **NS** záznam serveru, který ji může znát. **Rekurzivní** DNS server poté musí kontaktovat tento server.

Rekurzivní servery jsou většinou pouze servery nejblíž počítačum (klientů) poskytované od **ISP**.

- **výhody**: rychlejší než iterativní DNS servery, protože si v průběhu rezoluce mohou ukládat výsledek do cache a ten příště ihned zprostředkovat.
- **nevýhody**: bezpečnostní rizika, možnosti útoku jako **cache poisoning** nebo **DNS amplification attack** (DDoS viz <https://www.cisa.gov/uscert/ncas/alerts/TA13-088A>)

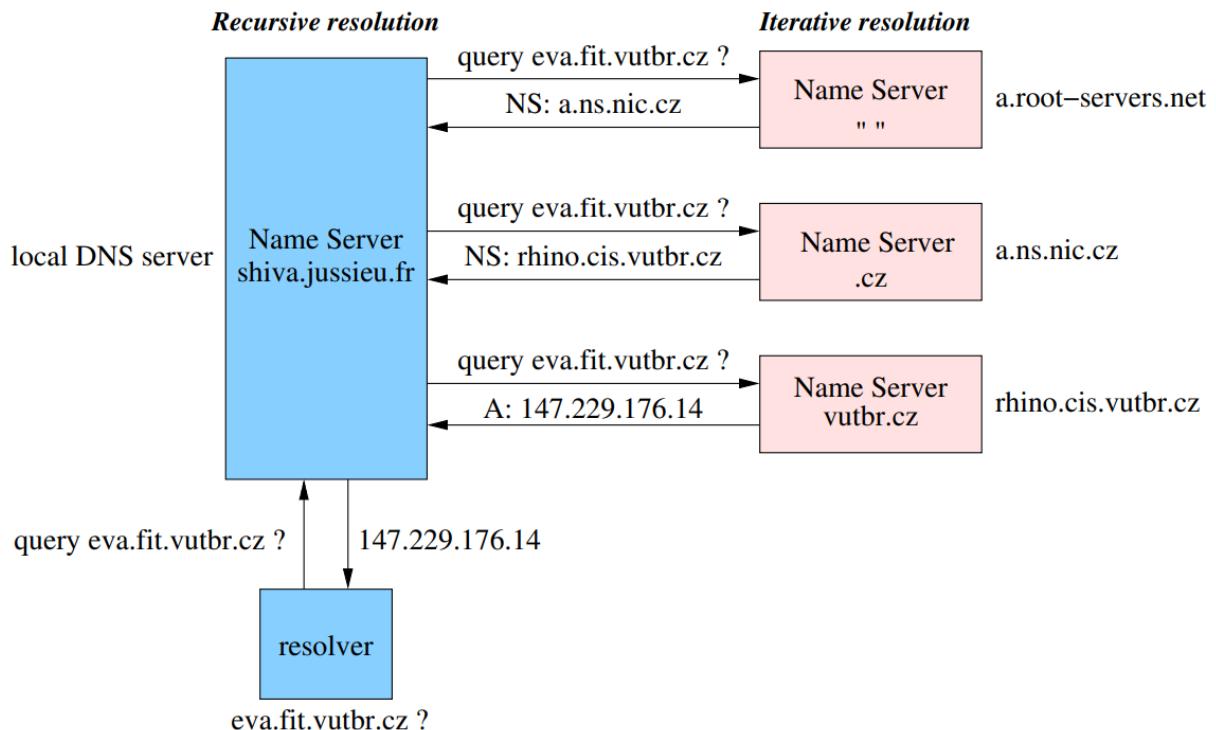
## Iterativní dotaz

Iterativní dotaz provádí **iterativní server**, který po dotazu vrátí nejlepší možnou odpověď, a to:

- **zná** mapování doménového jména na IP adresu, tak ji vrátí.
- **nezná** mapování na IP adresu
  - ale **zná DNS server**, který ji může znát (tentu server je na cestě od kořene DNS stromu k hledanému uzlu) a vrací **NS** záznam (doménové jméno autoritativního serveru) tohoto serveru,
  - **nezná** ani žádný **DNS server**, který by ji mohl znát a vrací NS záznam kořenového DNS serveru.

Dotazující se (klient nebo rekurzivní server) poté musí rezoluci opakovat s takto získaným **NS** záznamem.

## Obrázek rekurzivního i iterativního dotazu



## Přenos zón

Mechanismus, který umožňuje kopírovat zónové soubory z primárních DNS serverů na sekundární pro zvýšení rychlosti DNS rezoluce, zlepšení distribuce a rozdělení zátěže. Sekundární server musí aktualizovat zónové soubory s intervalu specifikovaném v záznamu **SOA** (DNS polling). Existují dva způsoby přenosu zón, oba jsou realizované přes **TCP** (rezoluce je přes UDP).

- **Celkový přenos zón** (AFXR): primární server posílá po vyžádání sekundárním celý zónový soubor.
- **Přírůstkový přenos zón** (IFXR): sekundární server posílá s výzvou o přenos záznam SOA, pro který chce zónový soubor. Primární server zkонтroluje, k jakým došlo změnám a odesílá pouze záznamy, která byly změněny.

Případně může primární server notifikovat (**DNS notify**) sekundární servery o změně zóny a ty si ji poté aktualizují, pak je zajištěno, že i sekundární servery mají vždy **aktuální** záznamy.

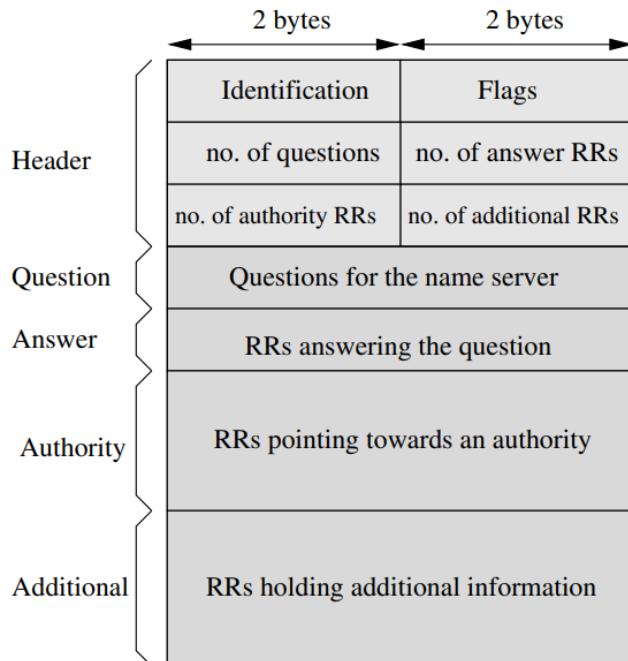
## Druhy DNS serverů

- **Primární**: poskytují vždy autoritativní odpověď pro daný SOA záznam, pro každou doménu je vždy **pouze jeden primární** server.
- **Sekundární**: získávají primární zónové soubory od primárních serverů, slouží jako záloha, poskytuje také autoritativní odpověď, protože má **celý zónový soubor** pro daný SOA záznam.
- **záložní/cach**: Pouze přijímá dotazy, které předává dalším DNS serverům a ukládá si odpovědi do VP. Pokud má uloženou odpověď ve VP, může jí vrátit při rezoluci, tato odpověď je ale **neautoritativní**. Záznamy z VP jsou po

určitém čase (uvezeno v záznamu od primárního/sekundárního serveru) odstraňovány/aktualizovány.

Každý server může být **současně** primární, sekundární i záložní. Každou funkci ale plní pro jiné domény.

## Struktura DNS zprávy



Example: answer to www.fit.vutbr.cz query

|                                                                                                                                                                                                                                                        |                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| 59058                                                                                                                                                                                                                                                  | qr, rd, ra, ad |
| 1                                                                                                                                                                                                                                                      | 1              |
| 4                                                                                                                                                                                                                                                      | 6              |
| www.fit.vutbr.cz IN A                                                                                                                                                                                                                                  |                |
| www.fit.vutbr.cz 13963 IN A 147.229.9.23                                                                                                                                                                                                               |                |
| fit.vutbr.cz. 13869 IN NS guta.fit.vutbr.cz.<br>fit.vutbr.cz. 13869 IN NS kazi.fit.vutbr.cz<br>fit.vutbr.cz 13869 IN NS rhino.cis.vutbr.cz<br>fit.vutbr.cz 13869 IN NS gate.feec.vutbr.cz                                                              |                |
| guta.fit.vutbr.cz. 14062 IN A 147.229.8.11<br>kazi.fit.vutbr.cz. 13869 IN A 147.229.8.12<br>rhino.cis.futbr.cz. 86211 IN A 147.229.3.10<br>guta.fit.vutbr.cz. 14156 IN AAAA 2001:67c:1220:809<br>rhino.cis.vutbr.cz. 43692 IN AAAA 2001:67c:1220:e000: |                |

Provádět DNS rezoluci můžeme např. pomocí nástrojů **nslookup**, **dig** a **host**.

## Struktura DNS záznamů

Resource Records Format

|                         |
|-------------------------|
| Name (variable length)  |
| Type (16 bits)          |
| Class (16 bits)         |
| TTL (32 bits)           |
| RDLENGTH (16 bits)      |
| RDATA (variable length) |

Example

|                       |
|-----------------------|
| email.fit.vutbr.cz    |
| CNAME                 |
| IN (0x0001)           |
| 4106 (1 h 8 min 26 s) |
| 10                    |
| hermina.fit.vutbr.cz  |

## Bezpečnost DNS

Jedná se o veřejnou a nešifrovanou službu, která může být terčem útoků. Mezi útoky patří:

- **podvržení odpovědi**: Útočník zašle neautorizovanou odpověď na dotaz

klienta před tím, než to stihne DNS server. Útočník hádá ID odpovědi. Tento útok vede na to, že klient obdrží nesprávnou IP a v případě např. http stránek může na této adrese existovat identická stránka (např. internetové bankovnictví), která je ale ovládaná útočníkem.

- **cache poisoning:** vložení nesprávné informace pro mapování DNS záznamů do VP záložních serverů (ty pak poskytují špatné adresy při rezoluci a nastává problém viz předešlý bod). Využívá se k tomuto útoku sekce **Additional**.
- (Distributed) **Denial of Service ((D)DoS):** Snaží o přetížení DNS serveru, které zamezuje poskytovat odpovědi na legitimní dotazy. Dnes je distribuce DNS tak vysoká, že tento útok je prakticky nereálný.

### Zajištění integrity a autentizace DNS záznamů

Zajištění těchto dvou vlastností, **chrání** před útokem **podvržením** i před **cache poisoning**. Zajišťujeme je pomocí **DNSSEC** - mechanismus, který používá **asymetrickou kryptografií** a **podepisování DNS záznamů a klíčů**. Pro implementaci **DNSSEC** se využívají další DNS záznamy:

- **DNSKEY:** veřejný klíč pro ověřování podpisů,
- **RRSIG:** podpis daného záznamu,
- **NSEC, NSEC3:** odkaz na další záznam při dotazu na neexistující doménu.
- **DS:** záznam pro ověření záznamu **DNSKEY**, uložen v nadřazené doméně.

### Řetězec důvěry (chain of trust)

Implementace zajištění **autentizace a integrity dat** DNS záznamů (implementace DNSSEC). Pro vytvoření řetězce důvěry se používají dva klíče (respektive 4 každý klíč tvoří dvojice **soukromý/veřejný**):

- **Zone Signing Key (ZSK):** slouží k podpisu dat **privátním** klíčem, která DNS server poskytuje v odpovědích. **Veřejný** klíč pro dešifrování odpovědi je poté uložen (a podepsán KSK) na dotazovaném serveru.
- **Key Signing Key (KSK):** privátní klíč slouží pro podepsání ZSK, veřejný je uložen na serveru rodičovské domény. Resolver tak může **validovat pravost** ZSK.

Princip **řetězce důvěry** je založen na **rekurzivním podpisu** klíčů. Kořenové servery mají **veřejně známé KSK** (není je třeba ověřovat), kterými podepisují své **ZKS**.

Pomocí **ZKS** kořenových serverů jsou pak podepsány záznamy **DS** (a ostatní) obsahující **KSK** domén **1. řádu** (TLD). Domény **1. řádu** pak tímto **KSK** podepíší svůj **ZKS** a tím podepisují záznamy ze svého zónového souboru, které obsahují také **DS** záznamy s **KSK** domén **2. řádu**... a takhle to jde až k listům doménového stromu.

### Šifrování provozu DNS

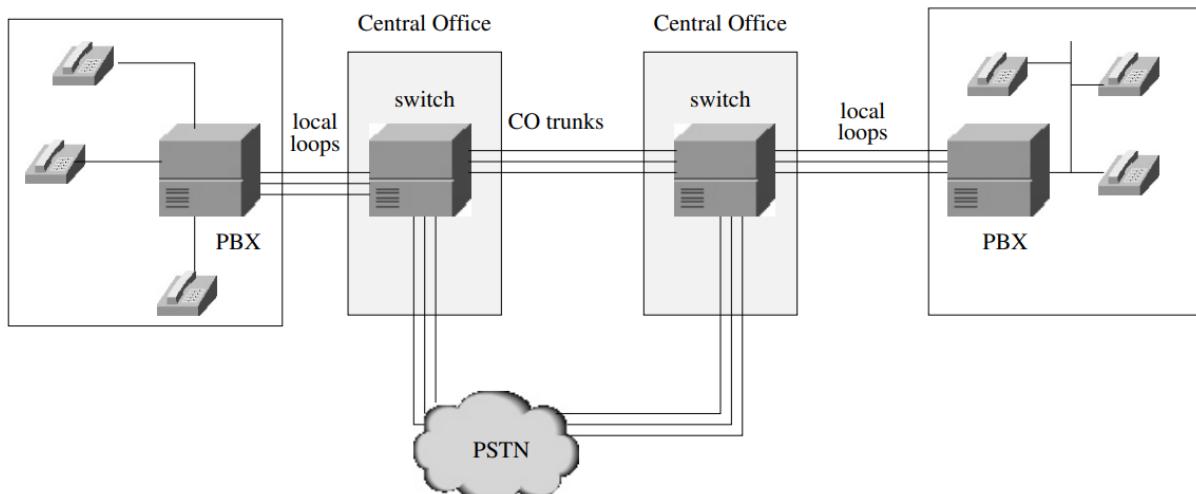
**Autentizaci a integritu** dat lze ještě rozšířit o **šifrování** záznamů. To je realizováno pomocí **DNS over TLS (DoT)** a **DNS over HTTPS (DoH)**. Oba vyžadují **TCP**. DNS dotazy jsou tak viditelné pouze klientovi a danému dotazovanému serveru. Řeší to sice problém s odposloucháváním, ale **neřeší to problém soukromí**. Server, na

který se dotazujeme (např. Google) pořád **má přehled** o tom, co navštěvujeme a může to tak využít např. pro reklamy.

## IP telefonie

Také známá pod zkratkou VoIP (Voice over IP). IP telefonie implementuje způsob komunikace (tj. primárně telefonní hovory), která je normálně běžná pro sítě s **přepojováním okruhů - Public Switched Telephone Network (PSTN)**. PSTN je tvořena:

- koncovými zařízeními (telefony), které jsou připojeny k **Private Branch Exchange (PBX)**.
- lokální smyčkou (spoj mezi PBX a ústřednou (Central office)),
- ústřednami (realizuje spojení s další ústřednou),
- páteřními spoji (trunks), které spojují ústředny.



Telefonní hovory pomocí **PSTN** se vyznačují:

- **garantováním šířky pásma** a spolehlivého přenosu,
- **dobrou kvalitou** přenosu u digitálních ústředen,
- **napájením koncových zařízení** (telefony fungují, i když vypadne proud - nesmí současně vypadnout i v ústředně),
- **spolehlivostí a bezchybností**, které jsou zajištěny dedikovanými spoji.

Stejné **požadavky** jsou tak očekávány i u **IP telefonie** na paketových sítích. Některé požadavky ale mohou být těžce realizovatelné, např. zajištění dostatečného přenosového pásma a s tím související kvalita hlasu (i když dnes to již se stoupající rychlostí internetu není problém) nebo zajištění napájené telefonů (nutnost použití záložních zdrojů). Dále je po IP telefonech požadována **integrace s veřejnou PSTN a mobilními sítěmi**.

## Architektura IP telefonie

Architektura IP telefonie je tvořena několika zařízeními:

- **DHCP server** - přiděluje IP adresu telefonům a ostatním zařízením, umožňuje získat adresu DNS serveru,
- **DNS server** - provádí překlad **tel. čísel a sip URI** na IP adresy,

- **Ústředna** (Call Server/Gatekeeper) - zajišťuje registraci VoIP zařízení, které pak lze vytáčet,
- **IP telefon** (IP phone) - může se jednat o tzv. **soft phone** tj. softwarový telefon, nebo normální telefon připojený k internetu,
- **Brána** (Trunk Gateway) - zajišťuje propojení s PSTN.

## Úkoly IP telefonie

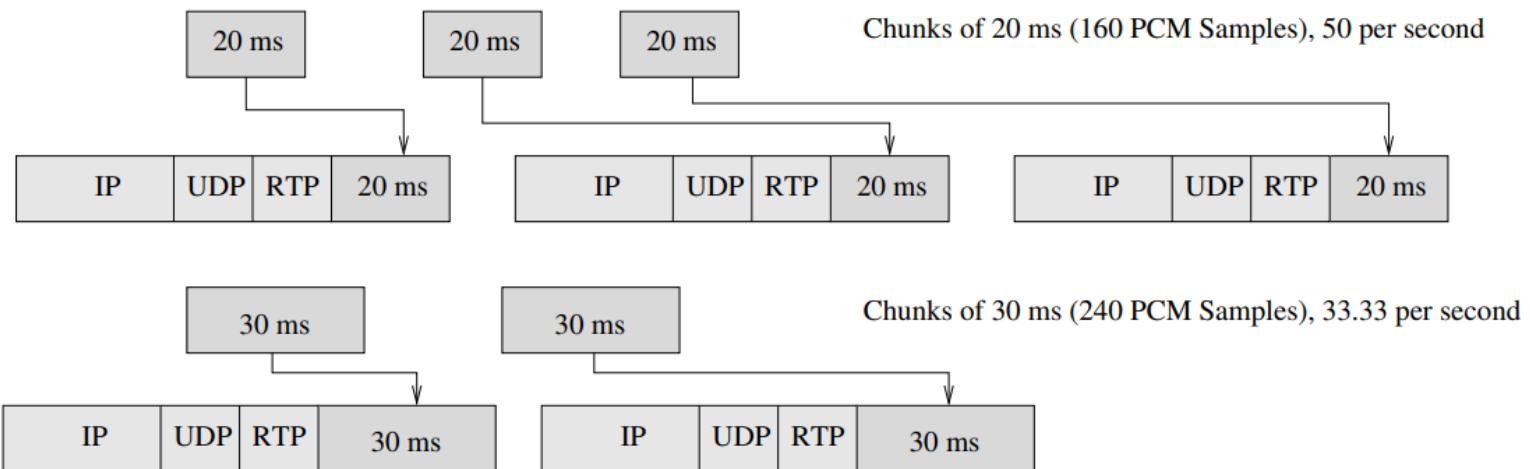
- **Převod hlasu na IP datagramy:** hlas - **analogový** signál, IP datagram - **digitální** a ještě k tomu nespojitý (nespojítý myšleno, že jsou data přenášena po kusech).
- **Řízení komunikace:** komunikuje se přes ústřednu - gatekeeper (klient-server) a mezi telefony (peer-to-peer). Nejčastější je ale zahájení spojení přes ústřednu a následná peer-to-peer komunikace. Je nutné zajistit:
  - **registraci účastníků** (na ústřednách),
  - **adresování hovorů** pomocí tel. čísel a SIP URI,
  - **směrování hovorů** tj. hledání cesty, kudy budou putovat pakety,
  - **vytváření hovorů** (obvykle peer to peer spojení mezi 2 telefony) a jejich udržování.
- **Připojení** do klasického telefonního systému **PSTN** a **mobilní sítě** pomocí brány (trunk gateway).
- **Aplikační služby** jako vyhledávání uživatelů pomocí LDAP nebo WWW či navazování spojení pomocí DNS a DHCP.

## Kódování hlasu

Hlas je nejdříve nutné **navzorkovat** a poté navzorkovaná data **rozdělit do paketů**, což se dělá pomocí **kódování hlasu** (kodeku). Kodeky využívají faktu, že lidská řeč není (obvykle) na celém rozsahu lidského slyšení a **ignorují okrajové frekvence**, tím provádí (ztrátovou) kompresi.

- Lidské **icho** vnímá na **20 Hz** až **20 kHz**, řeč je obvykle na **200 Hz** až **9 kHz** (9 kHz je ale spíš vysoký zpěv).
- U telefonní linky vzorkujeme frekvence **300 H** až **3.4 kHz**, případně až **4 kHz**.
- Pokud chceme vzorkovat **4 kHz** frekvenci, musíme vzorky vytvářet **2x** rychleji (Shannonův teorém) tzn. **8 kHz**. Řekněme, že jeden vzorek má 8 bitů (1 B). Za 1 sekundu budeme muset zpracovat  **$8000 \cdot 8 = 64 \text{ kb}$** . **Šířka pásma** tak bude **64 kb/s**.

Tato šířka pásma bere v potaz pouze samotná data, v paketových sítích je musíme ale zapouzdřit. Na aplikační vrstvě je to protokol **Real-time Transport Protocol** (RTP), který je dále zapojen do **UDP**, **IP** a **Ethernetového rámce**. Data navíc nemůžeme posílat po příliš velkých kusech (problém se ztrátou a zpožděním). Obvykle se posílají data po **20-30 ms**, což může díky hlavičkám jednotlivých protokolů způsobit navýšení potřebné šířky pásma třeba až o polovinu.



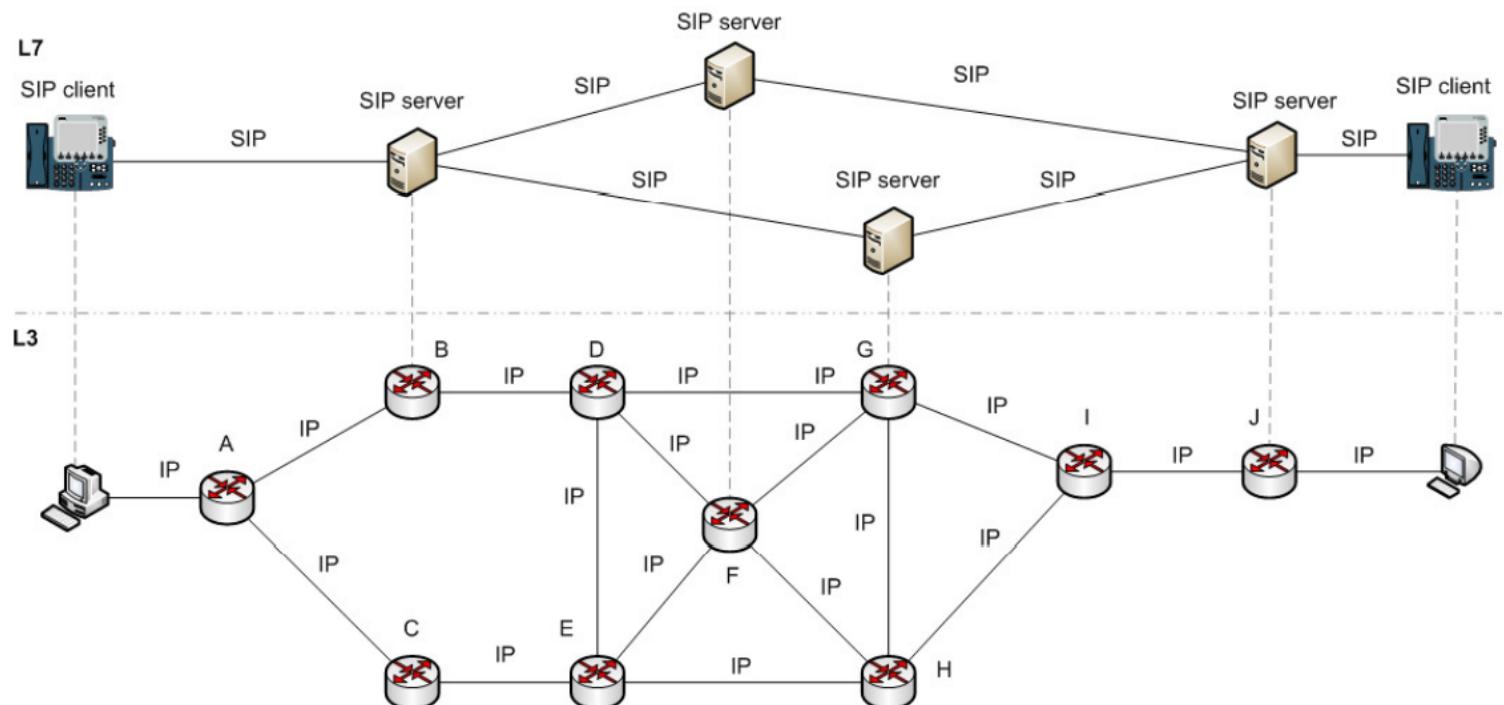
## Session Initiation Protocol (SIP)

Aplikační protokol nad UDP určený pro **signalizaci** VoIP (vytáčení atd.), neřeší přenos dat hovoru. Zajišťuje:

- registraci uživatelů (na bránu),
  - navázání spojení a směrování hovorů,
  - adresování pomocí URI (**sip:user@domain**) - adresa v IP telefonii.

Protokol **neprovádí**: správu relací po jejich vytvoření, nezajišťuje kvalitu hovoru, nezajišťuje přenos hlasových dat.

Jedná se o starý protokol a byl navržen na ISO/OSI modelu, používá tak L7 vrstvu.



## Architektura SIP

- **User Agent Server (UAS - SIP server)** - může mít jednu nebo více z funkcí:

- **Proxy server**: analyzuje zprávy a směruje hovory,
- **Lokalizační server**: má informace o umístění klientů,
- **Server pro směrování**: není nutný, jedná se pouze jen o další bod spojení. Směrování se provádí buď pomocí DNS nebo staticky (předkonfigurováno). Směrovací informace jsou v **SIP hlavičce**.
- **Registrační server**: přijímá žádosti REGISTER a aktualizuje lokaci.
- **User Agent Client (UAC - SIP klient)** - koncové zařízení, SIP telefon (fyzický nebo SW)

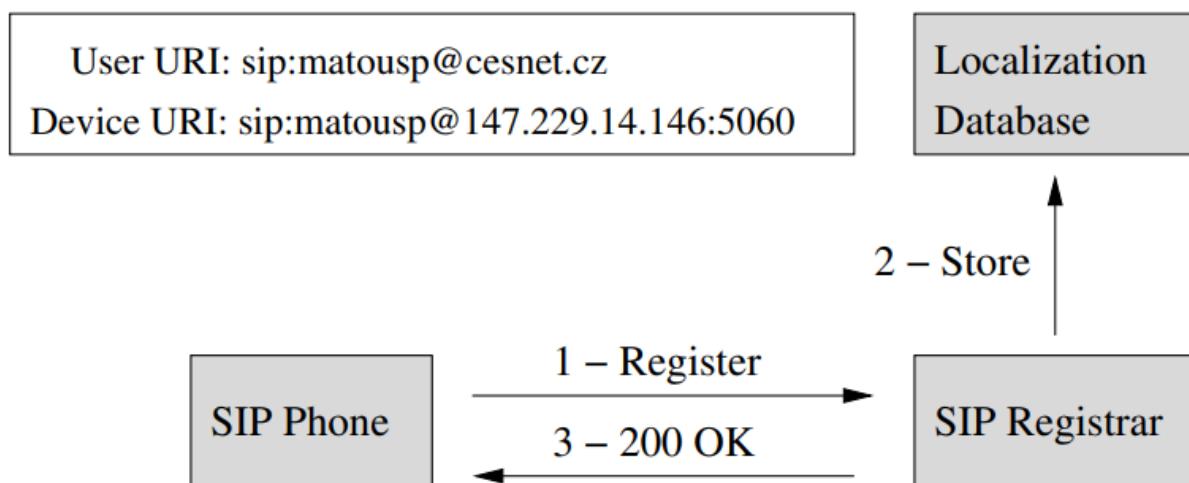
IP telefonie je pomocí SIP **globální** (zahrnuje celý internet) a **distribuovaná** (neexistuje centrální správa, SIP domény se dynamicky propojují). Systém tak umožňuje vyhledat a směrovat hovor na libovolný do sítě zapojený telefon.

### Příkazy protokolu SIP

- **REGISTER** - žádost o registraci,
- **INVITE** - zahájení komunikace vytáčení **volajícím**,
- **OK** - potvrzení komunikace **volaným**,
- **ACK** - potvrzení komunikace **volajícím**, už jde **peer-to-peer**,
- **CANCEL** - zrušení vyváření spojení,
- **BYE** - ukončení vytvořeného spojení, jde **peer-to-peer**,
- **OPTIONS** - získání možností přenosu.

### Registrace SIP telefonu

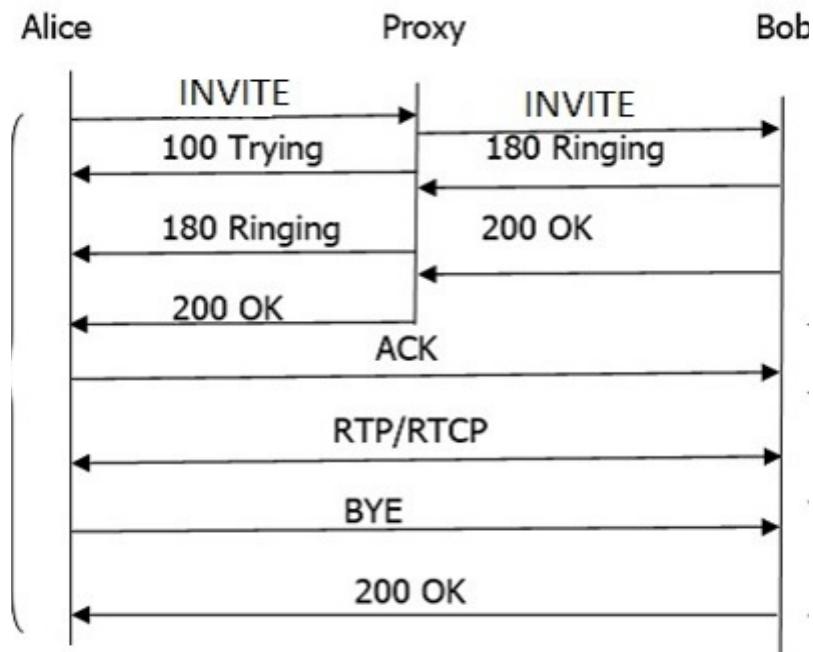
1. SIP klient pošle svou **IP adresu a port**, na kterém běží IP telefon, SIP serveru ve své doméně.
2. SIP klient mapuje ID uživatele (user URI - sip:user@sip.vutbr.cz) na adresu zařízení (device URI - sip:user@192.168.10.20:5060).
3. Registrační server si udržuje **lokalizační údaje** (SIP URI - IP adresy) o všech svých připojených klientech ve **své SIP doméně**. Používá k tomu **lokalizační databázi**.



sip:matousp@cesnet.cz  
147.229.14.146:5060

## Ustanovení hovoru

1. Volající zašle zprávu **INVITE**, která je směřována architekturou SIP (obecně více proxy serverů) až k příjemci.
2. Volaný (pokud chce komunikovat - tj. zvednutím sluchátka) zasílá zprávu **OK**, která je směřována architekturou SIP zpět k volajícímu.
3. Volající posílá zprávu **ACK** již přímo volanému (peer-to-peer).
4. Probíhá hovor tj. přenos zvukových dat pomocí protokolů **RTP** a **RTCP**.
5. Jeden z účastníků ukončí hovor zprávou **BYE** (peer-to-peer).
6. Druhý odpovídá s **OK**.



## Session Description Protocol (SDP)

Obsahuje **informace** potřebné pro **navázání datového spojení** pro přenos hlasu a videa (použité kodeky, bitrate, IP adresa spojení atd.). Protokol je zapouzdřen do SIP zpráv **INVITE** a **OK** při zahajování přenosu.

## Real-Time Transport Protocol (RTP)

Aplikační protokol pro **přenos hlasových a obrazových dat** pomocí protokolu **UDP**. Pro každý **směr** a typ **médií** se otevírá **samostatný RTP tok**, tzn. při videohovoru má volaný a volající otevřené mezi sebou 4 RTP toky. Hlavíčka protokolu mimo jiné obsahuje **typ přenášených dat, sekvenční číslo a časovou značku**, aby bylo možné data poskládat ve správném pořadí, pokud dojde k jejich přehození po cestě.

## RTP Control Protocol (RTCP)

RTCP poskytuje **řídící informace** pro RTP tok dat, ale sám žádná data nenesese. Používá se k pravidelnému přenosu **kontrolních** paketů účastníkům streamované

multimediální relace. Hlavní funkcí RTP je **poskytování zpětné vazby na kvalitu služeb** (QoS) poskytovanou RTP. Pro zajištění kvality je třeba eliminovat:

- **ozvěnu** (echo),
- **zpoždění paketů**,
- **ztrátu paketů** - ztráta pod 1% znamená dobrou kvalitu, ztráta od 1% do 2.5% je akceptovatelná a ztráta převyšující 5% již má velký dopad na přenos hlasu a videa,
- **rozptyl následujících paketů** (jitter).

## SIP a DNS

- **NAPTR**: DNS záznam, který se nejčastěji používá společně s **IP telefonií**, slouží k mapování adres SIP serverů. Záznamy **NAPTR** se běžně používají pro ověření existence SIP služeb na doméně.

```
>nslookup -type=NAPTR cesnet.cz
```

```
cesnet.cz. naptr = 200 50 "s" "SIP+D2T" "" _sip._tcp.cesnet.cz.
cesnet.cz. naptr = 100 50 "s" "SIP+D2U" "" _sip._udp.cesnet.cz.
```

- **SRV**: slouží k vyhledání příslušného SIP serveru.

```
>nslookup -type=SRV _sip._udp.cesnet.cz
```

```
_sip._udp.cesnet.cz. service = 100 10 5060 cyrus.cesnet.cz.
```

- **A** nebo **AAAA**: pro vyhledání IP adresy SIP serveru.

## Zabezpečení VoIP

bezpečnostní **rizika** a jejich řešení:

- **Odposlech**: lze řešit fyzickým zabráněním přístupu k síti nebo pomocí zabezpečení spojení (IPSec, Secure RTP)
- **Viry, Spam over IP Telephony, Phishing over IP Telephony**: antivirus, filtry, vzdělání uživatelů.
- **Neautorizované použití**: vyžadovat autentizaci.
- **Výpadky napětí**: Power over Ethernet, záložní zdroje pro IP telefony a síťovou infrastrukturu (switches a routery).

## Alternativy k VoIP

- **rozdíly**: Používají proprietární nestandardní protokoly. Mohou používat rozdílnou architekturu (peer-to-peer, hybridní - jako VoIP, pouze klient-server). Proprietární správa účtů. Negarantují službu a nemusí např. poskytovat tísňová volání, což IP telefony musí. Mají omezené možnosti propojení s PSTN, mobilními sítěmi a standardním VoIP.
- **společné vlastnosti**: adresování, směrování a přenos hlasových dat a stímem spojené udržování spojení, vytváření účtů a registrace uživatelů a jiné službu kromě hovorů, např. Instant Messaging, přenos souborů, přenos obrazu,

sdílení plochy atd.

## Správa sítí

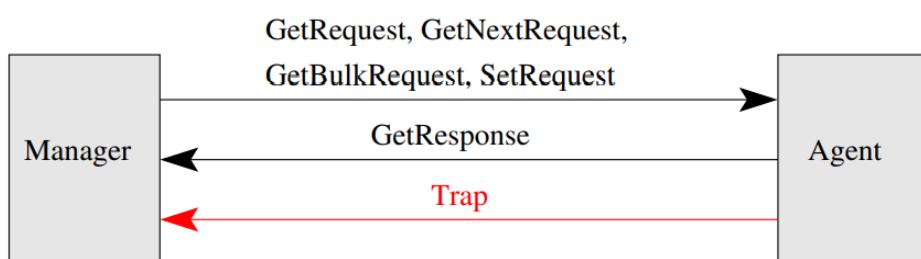
Správa sítí zahrnuje řešení problémů známých pod akronymem **FCAPS**:

- **Fault management**: zaznamenání, izolování a opravení chyby.
- **Configuration management**: zavádění konfigurací zařízení, zálohování konfigurací, zaznamenávání změn, aktualizace SW.
- **Accounting management**: zisk statistik využitívání sítě (např. IP telefonie) a stím spojené účtování.
- **Performance management**: zajištění dostatečného výkonu sítě (např. šířky pásmo pro VoIP)
- **Security management**: zabezpečení sítě (autentizace, autorizace, zabránění fyzického přístupu k zařízením atd.)

## Simple Network Management Protocol (SNMP)

SNMP je protokol pro přenos informací o zařízeních na síti, pod pojmem SNMP ale obecně myslíme celý mechanismus pasivní správy sítí (aktivní je např. ICMP ping, telnet, PortQuery, ...) za použití tohoto protokolu. SNMP zpráva sítě je tvořena:

- **Management Station**: jedná se o server na kterém běží SW, který se pomocí **protokolu SNMP** obvykle pravidelně dotazuje (polling) **SNMP agentů** na portu **161** a stahuje si od nich tak **monitorovací informace**, které ukládá a např. umožňuje nějak graficky zobrazit. Dotazy na data **MS** realizuje příkazy **Get**, **GetNext**, **GetBulk** a MS také může konfigurovat SNMP agenta pomocí příkazu **Set**.
- **SNMP Agent**: jedná se o **zařízení v síti** (switche, routery, tiskárny, počítače atd.), u síťových zařízení je nainstalován SW pro SNMP protokol obvykle od výrobců, na PC si jej musíme stáhnout. SNMP agent **sbírá informace o zařízení**, na kterém běží (např. vytížení CPU, vytížení paměti, stav toneru, počet papírů v zásobníku, počet odeslaných paketů, počet přijatých paketů atd.) a na vyzvání Management Station je jí odesílá. Nevede si však historii, tu musí zaznamenávat MS. SNMP agent může kontaktovat MS **sám** od sebe při **vzniku** nějaké závažné **chyby** (nakonfigurováno jaká chyba, resp. jaká úroveň závažnosti). Tato komunikace je realizována pomocí **asynchronní** (tj. MS si ji nevyžádala) zprávy **Trap** na portu **162**.
- **SNMP protokol**: **nestavový** protokol sloužící pro přenos informací o zařízeních na síti, jedná se o protokol typu request-response a existuje několik verzí **SNMP**, **SNMPv2** (přidává autentizaci), **SNMPv3** (přidává šifrování).



|                        |                         |                         |
|------------------------|-------------------------|-------------------------|
| <b>SNMP header</b><br> | Version                 | Version                 |
|                        | Community String        | Community String        |
|                        | PDU type: GetRequest    | PDU type: GetResponse   |
|                        | Request ID              | Request ID              |
|                        | 0                       | Error Status            |
|                        | 0                       | Error Index             |
|                        | Object Name 1   Value 1 | Object Name 1   Value 1 |
|                        | Object Name 2   Value 2 | Object Name 2   Value 2 |
|                        | Object Name 3   Value 3 | Object Name 3   Value 3 |

GetRequest,GetNextRequest,SetRequest

GetResponse

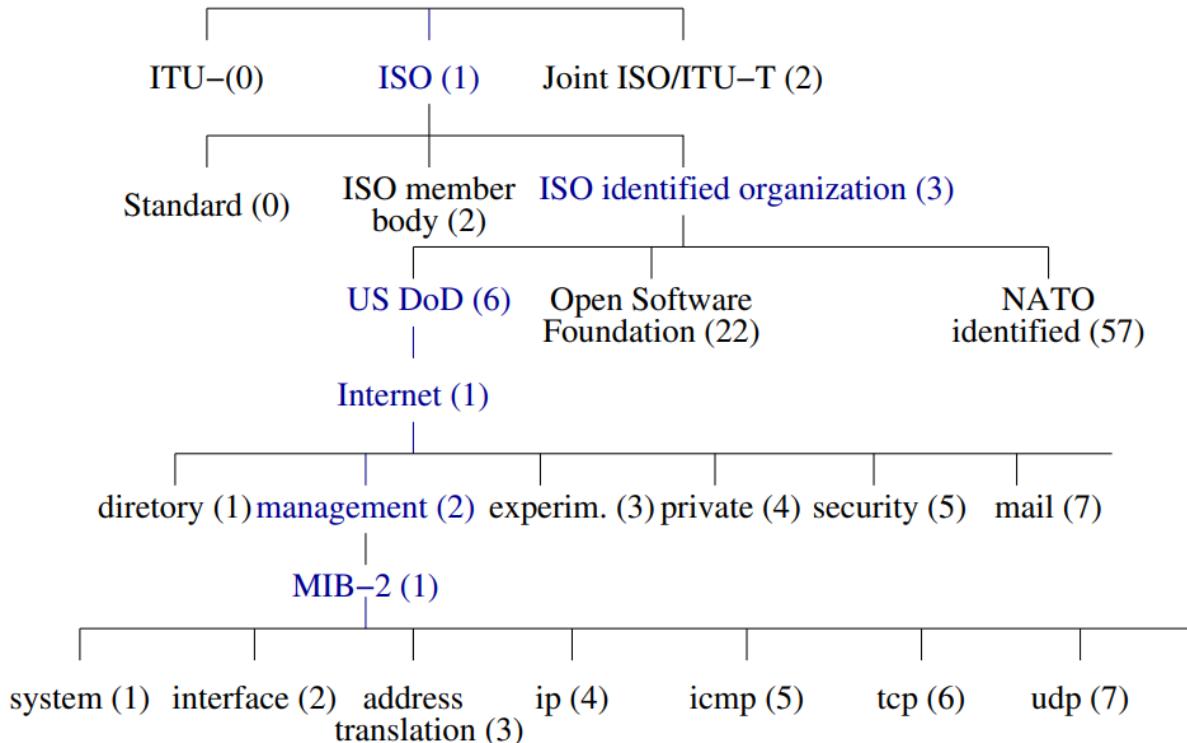
Trap

## Monitorované objekty

Objekty jsou uspořádané do **skupin** v **databázi objektů MIB** (Management Information Base). Databáze MIB má stromovou strukturu:

- **nelistové uzly**: reprezentují skupiny objektů,
- **listové uzly**: reprezentují konkrétní objekt.

Objekty jsou adresováný pomocí **OID**, které **reprezentuje cestu ve stromové struktuře** a má tvar čísel oddělených tečkami, kde čísla identifikují uzel na dané úrovni stromu a tvoří tak cestu. Např. **OID 1.3.6.1.2.1.4** představuje objekt **ip**.



## Definice monitorovaných objektů

Objekty jsou definovány a popisovány pomocí jazyka **Structure Management Information** (SMI). Popis objektu je tvořen:

- **jménem objektu**, což je **OID** - jednoznačný identifikátor objektu,
- **syntaxí**, která určuje (abstraktní) datový typ spojený s objektem, např (Counter32 - přeteče zpět na 0, Gauge32 - nepřeteče a setrvává na max hodnotě, **OBJECT IDENTIFIER**, **SEQUENCE**, **INTEGER**, **IpAddress**, **NetworkAddress**, **OCTET STRING**, ...)
- **kódováním pro přenos po síti** - používá se kódování **BER**.

Příklad definice objektu **ipInDelivery** ve skupině **ip**:

```

ipInDelivers OBJECT-TYPE -- OID 1.3.6.1.2.1.4.9
 SYNTAX Counter32
 MAX-ACCESS read-only
 STATUS current
 DESCRIPTION "The total number of input datagrams successfully
 delivered to IP user-protocol (including ICMP)."
 ::= { ip 9 }

```

## Basic Encoding Rules (BER)

Jedná se o binární kódování informací po za sebou jdoucích trojicích tvořených:

- **Type**: typ objektu uložen v **1. bytu**.

8 7 6 5 4 3 2 1

| Class            | P/C | Tag number                           | Length | Value |
|------------------|-----|--------------------------------------|--------|-------|
| Identifier octet |     | Specifies the end of the value field |        |       |

- **Length:** délka hodnoty v bytech, implicitně je uložena pouze na 1 B (tj. na druhém bytu). Pokud nelze vyjádřit na 1 B (tj. v tomhle případě větší než 0x80), pak je délka vyjádřena v N následujících bytech a **N = hodnota 2. B - 0x80.**
- **Value:** hodnota záznamu.

## Nasazení SNMP

Při praktickém nasazení SNMP musíme brát v potaz:

- **kolik zařízení** bude monitorováno,
- jaká bude **frekvence sběru** dat,
- jaký bude **objem přenášených dat**.

Tyto tři faktory musí být vyváženy tak, aby nepředstavovaly významné zatížení sítě. Dále musíme brát v potaz, že protokol **SNMP** je **zapouzdřen do UDP** a může tak docházet ke **ztrátám** datagramů (není zde potvrzení). Např. zpráva **Trap nemusí dorazit** nebo zápis konfigurace pomocí příkazu **Set se nemusí provést** atd. Systém SNMP je **centralizovaný** a všechna data jsou na jedné stanici (Management Station), při poruše této stanice ztrácíme přehled o síti. Měli bychom používat **šifrovanou verzi** SNMPv3 kvůli **bezpečnosti**, pokud by se někdo naboural do sítě a zde běželo SNMP starších verzí, získá informace o téměř všech zařízeních. Program pro SNMP: **snmpwalk**.

## NetFlow

NetFlow narodil od SNMP neumožňuje monitorovat zařízení v síti a získávat informace jako (zatížení CPU a paměti, stav toneru, počet papírů v tiskárně atd.). NetFlow se zaměřuje více na monitorování komunikace na síti v podobě síťových toků, tj.:

- **kdo a s kým komunikuje** (ip adresy a jiné adresy),
- **kolik dat si posílají**,
- **kdy komunikují**,
- **jaké používají protokoly**,
- atd.

Informace o stavu zařízení a jestli funguje nelze získat pomocí NetFlow. NetFlow byl původně **proprietární** protokol vyvinutý firmou **Cisco**, dnes je již **standardizován** pomocí **RFC**. Stejně jako u SNMP nepovažujeme NetFlow pouze za protokol, ale mechanismus monitorování a správy sítě. Architektura NetFlow je tak tvořena:

- **Exportérem:** jedná se o sondu/router pro získávání statistik o tocích. Často se jedná o samostatnou sondu která pouze **odposlouchává** provoz a

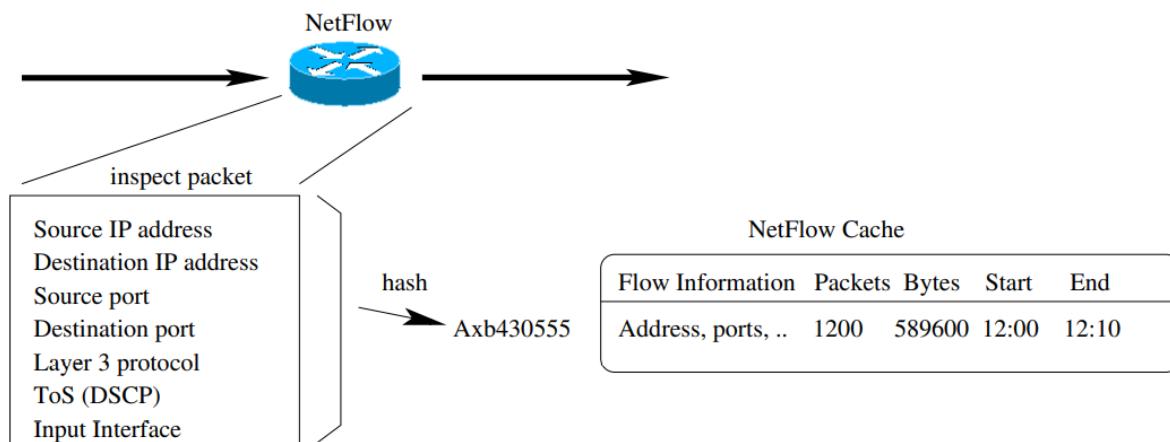
zaznává jej, protože řešení v rámci routeru může být příliš drahé.

- **Kolektorem:** zařízení, které ukládá záznamy o tocích. Jedná se o nějaký server (podobně jako MS u SNMP) tzn. **centralizované** řešení.
- **Protokolem:** NetFlow\_v5, NetFlow\_v9, IPFIX.
- **Nástroje pro zobrazení dat:** grafy, statistiky, výpisy atd.

## Síťový tok

**Posloupnost paketů** mající **společnou vlastnost** (např. IP adresu), které prochází určitým **bodem pozorování** za **určitý časový interval**. Záznam o toku obsahuje informace o toku, **nikoliv přenášená data**. Mezi tyto informace patří:

- zdrojová a cílová **IP adresa**,
- zdrojový a cílový **port**,
- **typ protokolu** (může být z více vrstev tj. transportní a aplikační),
- **časové razítko** (obsahuje začátek a konec toku),
- **počet přenesených dat** v B.



## Exportér NetFlow

**Síťové zařízení** a software (často dedikované - sonda), které **monitoruje** procházející provoz. Vytváří **záznamy** o tocích a **aktualizuje** starší záznamy ve své **NetFlow cache**. **Vyhledávání** záznamů pro aktualizaci provádí **pomocí heše** informací identifikující tok (IP adresy a porty). To znamená, že pro **každou komunikaci vytváří 2 toky** (záleží na směru). Exportér také může provádět nějaké agregace dat, více viz dále. Na rozdíl od SNMP odesílá **exportéry** data kolektoru **sami bez vyžádání** po:

- ukončení toku (TCP - **FIN, RST**),
- po **vypršení časovače**
  - **neaktivní timeout:** nebyl odeslán žádný paket v daném toku (defaultně 15 s),
  - **aktivní timeout:** odesílá data u příliš dlouhých nepřerušovaných toků, může jít např. o nějaké stahování (délka může být třeba 30 min),
- **zaplnění NetFlow cache**.

## Kolektor NetFlow

Kolektor je na síti **jeden** a přijímá pakety NetFlow z **jednoho či více exportérů**.

Jeho funkce jsou poté následující:

- **Zpracování záznamů** o tocích a jejich **ukládání** na disk nebo do databáze, ukládají se binárně ve formátu optimalizovaném pro vyhledávání (před uložením mohou být data ignorována),
- **Grafické zobrazování** dat (např. nějaká **heat mapa** vytíženosti) pomocí SW, např. **Flowmon**,
- **Realizovat dotazování** nad daty pomocí např. **nfdump** (provádět agregace uložených neagregovaných dat - chci zobrazit všechny toky z jedné IP, druhá mě ale nezajímá, a sečít objem přenesených dat - typický příklad pro účtování)
- **Automatizace** např. automatické odhalení anomalií - podezřelých toků (PC v kanceláři s pracovní dobou od 9 do 17 začne ve 22 posílat/přijímat velké množství dat → možná je to útok).

## Protokol NetFlow

Příklad paketu **verze 5**:

| Version  | Count             | sysUptime              |           |                   |     |  |  |
|----------|-------------------|------------------------|-----------|-------------------|-----|--|--|
|          | Unix secs         | Unix nsecs             |           |                   |     |  |  |
|          | Flow sequence     | Engine Type            | Engine ID | sampling interval |     |  |  |
|          | Source IP address | Destination IP address |           |                   |     |  |  |
|          | Next hop          | Input                  |           | Output            |     |  |  |
|          | Packets           | Octets                 |           |                   |     |  |  |
|          | First SysUptime   | Last SysUptime         |           |                   |     |  |  |
| Src Port | Dst Port          | Pad1                   | Flags     | Proto             | ToS |  |  |
| Src AS   | Dst AS            | Src mask               | Dst mask  | Pad2              |     |  |  |

Paket je tvořen:

- **hlavičkou**, která obsahuje
  - počet NetFlow záznamů přenášených v datové části,
  - čas odeslání tohoto NetFlow paketu,
  - identifikaci sondy, ze které byl paket odeslán
  - atd.
- **daty** (obsah paketu) obsahuje množinu záznamů o tocích, každý záznam představuje statistiku o jednom toku.

## Vylepšení ve verzi 9 a IPFIX (IP Flow Information Export)

NetFlow verze 5 má **fixní** formát (neumožňuje např. export IPv6 adres), **verze 9** to řeší zavedením šablon. Hlavička zůstává a obsahuje informaci o **ID šablony**, podle

které jsou zapsána data v datové části paketu (šablony zasílá exportér kolektoru, ten si ji pak zapamatuje a již není součástí paketů - datová část tak může obsahovat šablony, informace o datových tocích nebo oboje, **většinou ale pouze** informace o **datových tocích**). **IPFIX** je poté ještě **více flexibilní**, definuje více polí, které lze použít v šablonách, jinak se výrazně neliší.

## Použití NetFlow

- **Monitorování sítě**, plánování sítě, **bezpečnostní analýza**,
- Sledování aplikací, uživatelů a zajištění **účtování**,
- **Dlouhodobé ukládání** informací o tocích (někteří poskytovatelé to mají dané ze zákona - pak je nutné počítat poměrně s velkou paměťovou náročností objem NetFlow statistik je roven asi 1-2 % objemu provozu sítě, tj. při dení zátěži 100 GB bude třeba uložit 1-2 GB NetFlow dat).

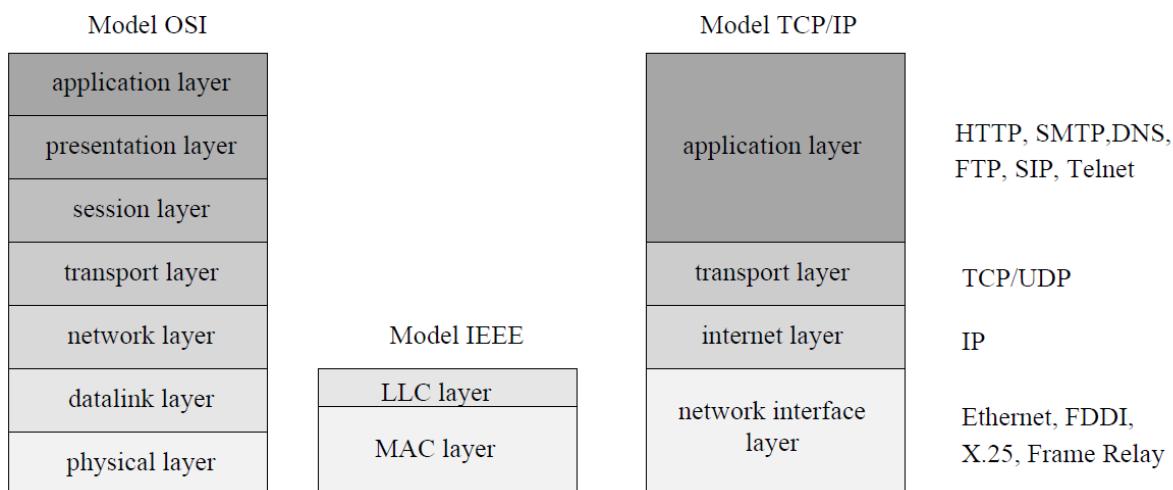
# 43. TCP/IP komunikace (model klient-server, protokoly TCP, UDP a IP, řízení a správa toku TCP)

## Model TCP-IP

Model TCP/IP je v současnosti hlavní využívaný síťový model. Má **5 vrstev** (v některé literatuře 4, kdy spodní 2 jsou spojeny do vrstvy síťového rozhraní). Je tvořen:

- **Aplikační vrstva** (L7 - používá se i pro TCP/IP), obstarává komunikaci na úrovni aplikací (viz předchozí otázka) - adresa je např. URL nebo emailová adresa,
- **Transportní vrstva** (L4) řeší komunikaci mezi **2 logickými procesy** - adresou je číslo portu,
- **Síťová vrstva** (L3) řeší komunikaci mezi **2 stroji** napříč celou sítí - adresou je IPv4 nebo IPv6 adresa,
- **Linková vrstva** (L2) řeší komunikaci dvou **síťových rozhraní** - adresou je MAC adresa,
- **Fyzická vrstva** (L1) pak přenos po médiu.

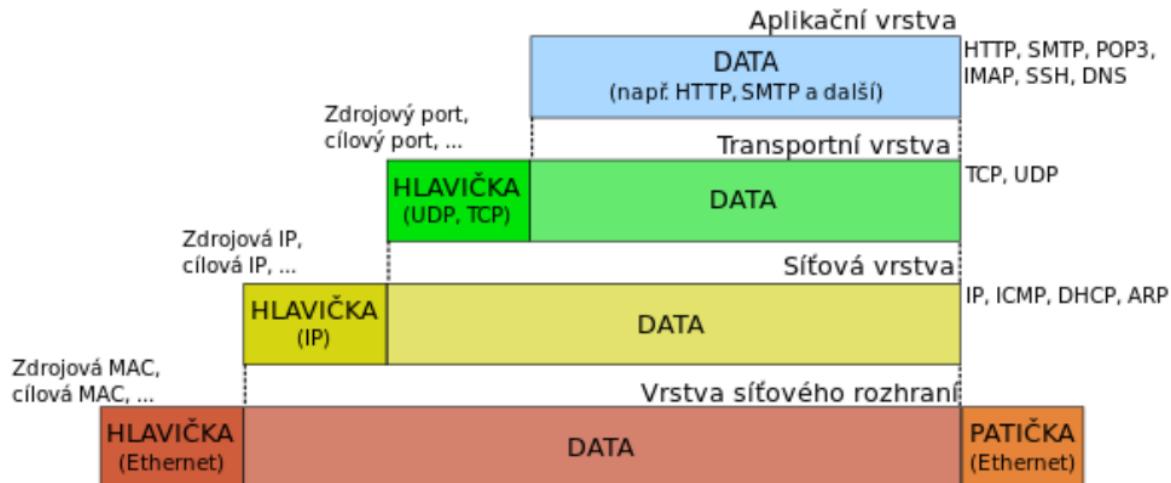
### ❖ Architektura TCP/IP



Důvodem vrstvení modelu je **oddělení logiky jednotlivých vrstev**, například L4 staví na službách L3 a své služby poskytuje vyšší vrstvě (L7). Cílem je schopnost doručit paket z libovolného uzlu do jiného uzlu za jakýchkoliv okolností.

Při průchodu paketu sítí probíhá tzv. **zapouzdření** (vzniká **PDU** - Protocol data unit), tj. čistá data jsou nejprve zabalena do L7 hlavičky, takovýto paket je zabalen do L4 hlaviček (tím vzniká UDP datagram nebo TCP segment), ten je zabalen do IP hlavičky, tak vzniká IP paket, ten je zabalen do L2 hlaviček a vzniká rámec. Při

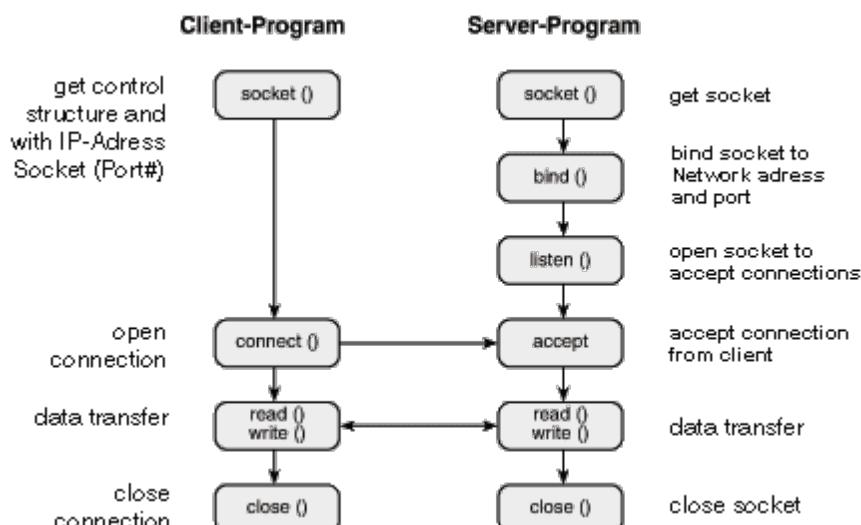
průchodu sítí síťové prvky rozbalují pouze část hlaviček dle své vrstvy, např. **switch** (L2 prvek) si přeče L2 hlavičku a při přeposlání přidá novou, podobně pak **router** (L3 prvek) nahlíží do IP hlaviček a podle nich směruje.



## Klient-server

Model klient-server je běžné komunikační schéma mezi 2 procesy. **Klient aktivně zahajuje spojení** k serveru a požaduje od něj službu (posílá mu požadavky a dostává na ně odpovědi). **Server pasivně čeká** na požadavek klienta, který následně provede a zašle odpověď odpovídající např. úspěšnosti provedení požadavku. Rozlišujeme **iterativní** a **konkurentní server**:

- **iterativní server** zpracovává jednoho klienta po druhém, nemůže zpracovávat více klientů současně, ostatní musí čekat,
- **konkurentní server** může naráz zpracovávat více klientů díky využití **dětských procesů** (funkce *fork*), kdy každý proces zpracovává požadavky jednoho klienta. Implementace je standardně následující:
  - Otevře se schránka (funkce *socket*), sváže se s portem (*bind*) a server začne poslouchat (*listen*)
  - Po přijetí spojení od klienta (*accept*) vytvoří nový proces (*fork*).
  - Synovský proces používá původní schránku, rodičovský proces ji uzavře (*close*), aby každou schránku obsluhoval jediný proces. Rodičovský proces pak ihned může zpracovat další požadavek.
  - Následně klasicky probíhá komunikace pomocí **read/write**.



- konkurentní server je **typický pro TCP**, protože lze jednoznačně přiřadit příchozí zprávu od klienta do správné schránky na základě src a dst IP adres a portů (zajišťuje OS, klient si pomocí funkce connect blokuje port - connection-oriented protocol).
- UDP typicky nebývá konkurentní, je možné tento nedostatek obejít například využitím **nového portu** pro každého klienta, UDP server klientovi řekne, ať pro další komunikaci použije **jiný port** než ten standardizovaný, protože funkce sendto může použít pokaždé náhodný port (tak funguje TFTP).

Komunikace typicky probíhá dle nějakého **protokolu**, což je soubor syntaktických a sémantických pravidel, kterými se komunikace řídí. Protokol je možné popsat:

- neformálně slovně** (např. RFC)
- formálně** (konečné automaty, gramatiky).

Protokoly typicky popisují navázání spojení, adresování, přenos dat, řízení toku komunikace.

## Dělení komunikace

Z pohledu počtu komunikujících entit rozlišujeme:

- unicast** = zpráva pro jeden uzel
- broadcast** = zpráva pro všechny uzly v síti
- multicast** = zpráva pro vybranou skupinu uzlů v síti (typicky dynamicky tvořenou)

## Transportní vrstva

Transportní vrstva vytváří logické **spojení mezi procesy**, hlavními protokoly jsou **TCP** a **UDP**. Mezi více procesy na jednom počítači a jejich komunikacemi je rozlišeno pomocí přiděleného **portu**. Ke komunikaci na L4 úrovni slouží SW prostředek **schránky** (socket). Port je 16-bitové číslo (0-65535), dělíme je:

- 0-1023, systémové = standardizované, např. http 80
- 1024-49151, uživatelské
- 49152-65535, dynamické

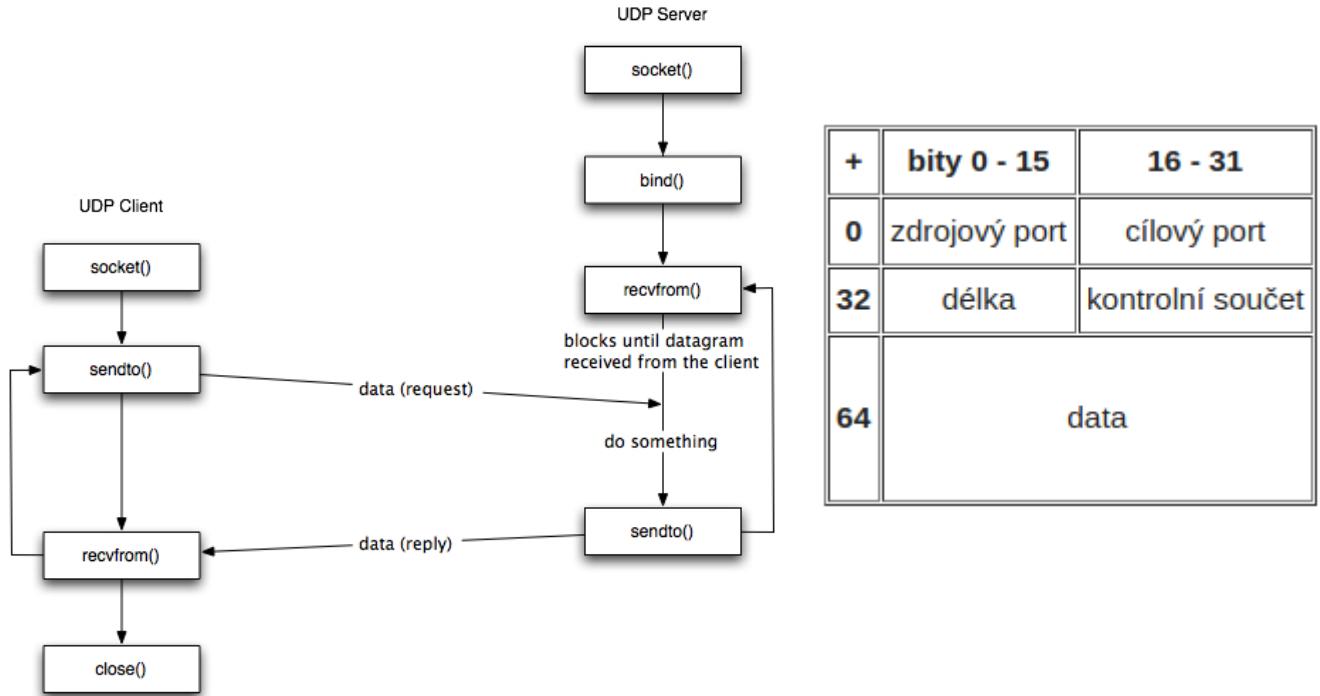
Číslo portu serveru musí být známé při navazování spojení (např. specifické číslo pro daný protokol, viz IANA). Číslo portu klienta bývá **náhodně generováno systémem**.

## User Datagram Protocol (UDP)

Connection-less protokol, **nenavazuje tedy spojení**, podobně jako na L3 doručení probíhá na bází **best-effort**. Hlavíčka je tedy velmi jednoduchá, obsahuje **zdrojový a cílový port, checksum a délku** (8B).

Protože není navázáno spojení, nezaručuje doručení, ani doručení v pořadí. Díky tomu je ovšem komunikace **rychlejší**, což je vhodné pro časově závislé aplikace

(např. přenos hlasu přes RTP, video streaming, DNS). V případě požadavků na **spolehlivost** je nutné to řešit na L7 (např. protokol TFTP pracuje nad UDP, na L7 přidává ACK datagramy, díky kterým je možné poznat, zda doručení proběhlo v pořádku).



## Transmission Control Protocol (TCP)

Spojově orientovaný protokol poskytující **spolehlivý přenos** dat (řeší ztráty paketů a pořadí, vyšší vrstvy se o případné ztrátě ani nedozví). Na začátku komunikace se vytváří spojení pomocí **3-way handshake**. Využívá pipeliningu (viz dále) a klouzavého okna, díky kterému je přenos řízen a je předcházeno zahlcení. Hlavička TCP obsahuje:

- **porty** (viz UDP),
- **checksum**,
- **sequence number** = číslo prvního B segmentu,
- **acknowledgement number** = číslo prvního B očekávaného segmentu k přijetí,

- příznaky.

| Octet | Bit | 0               |   |   |   |     |   |   |   | 1   |   |   |   |     |   |   |   | 2                                       |     |     |     |     |     |     |     | 3   |             |   |   |   |   |   |   |
|-------|-----|-----------------|---|---|---|-----|---|---|---|-----|---|---|---|-----|---|---|---|-----------------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-------------|---|---|---|---|---|---|
|       |     | 7               | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 7   | 6 | 5 | 4 | 3   | 2 | 1 | 0 | 7                                       | 6   | 5   | 4   | 3   | 2   | 1   | 0   | 7   | 6           | 5 | 4 | 3 | 2 | 1 | 0 |
| 0     | 0   | Source Port     |   |   |   |     |   |   |   |     |   |   |   |     |   |   |   | Destination Port                        |     |     |     |     |     |     |     |     |             |   |   |   |   |   |   |
| 4     | 32  | Sequence Number |   |   |   |     |   |   |   |     |   |   |   |     |   |   |   | Acknowledgment Number (If ACK Flag Set) |     |     |     |     |     |     |     |     |             |   |   |   |   |   |   |
| 8     | 64  | Data Offset     |   |   |   |     |   |   |   |     |   |   |   |     |   |   |   | NS                                      | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN | Window Size |   |   |   |   |   |   |
| 12    | 96  | ...             |   |   |   | ... |   |   |   | ... |   |   |   | ... |   |   |   | ...                                     |     |     |     | ... |     |     |     | ... |             |   |   |   |   |   |   |
| ...   | ... | ...             |   |   |   |     |   |   |   |     |   |   |   |     |   |   |   |                                         |     |     |     |     |     |     |     |     |             |   |   |   |   |   |   |

### TCP Segment Header

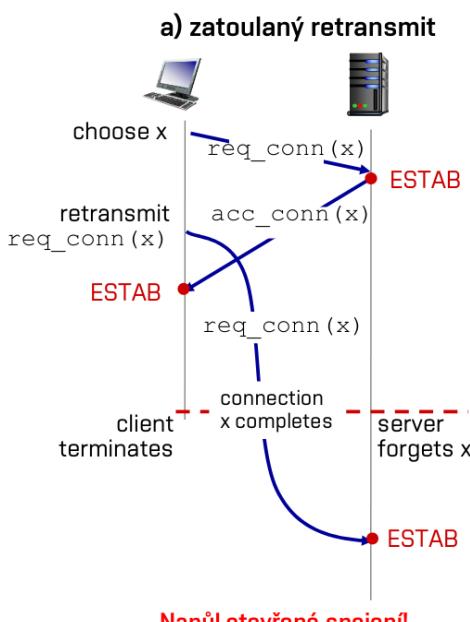
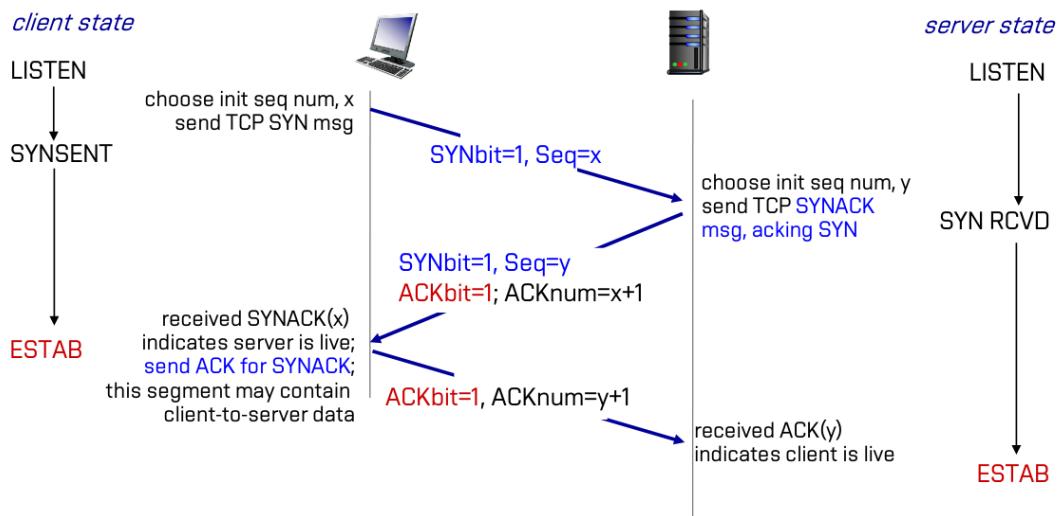
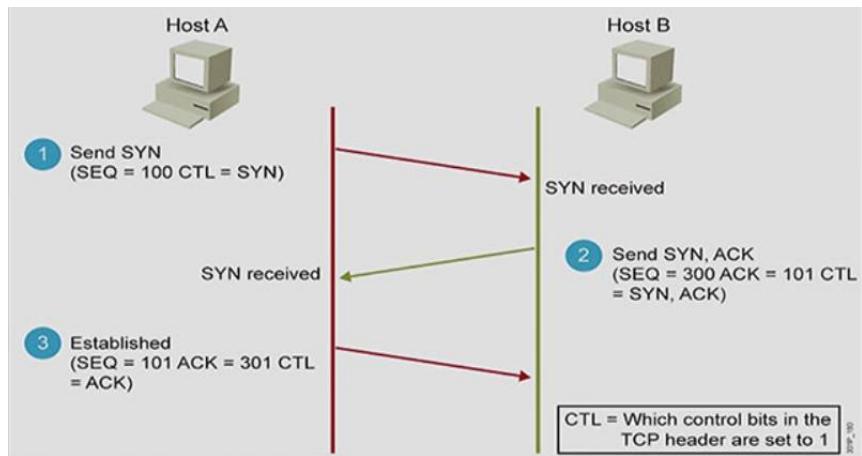
TCP rozděluje přicházející data do segmentů (je-li to třeba dle MTU) a na straně příjemce je zase skládá.

### 3-way handshake

Navázání spojení probíhá na principu 3-way handshake následovně:

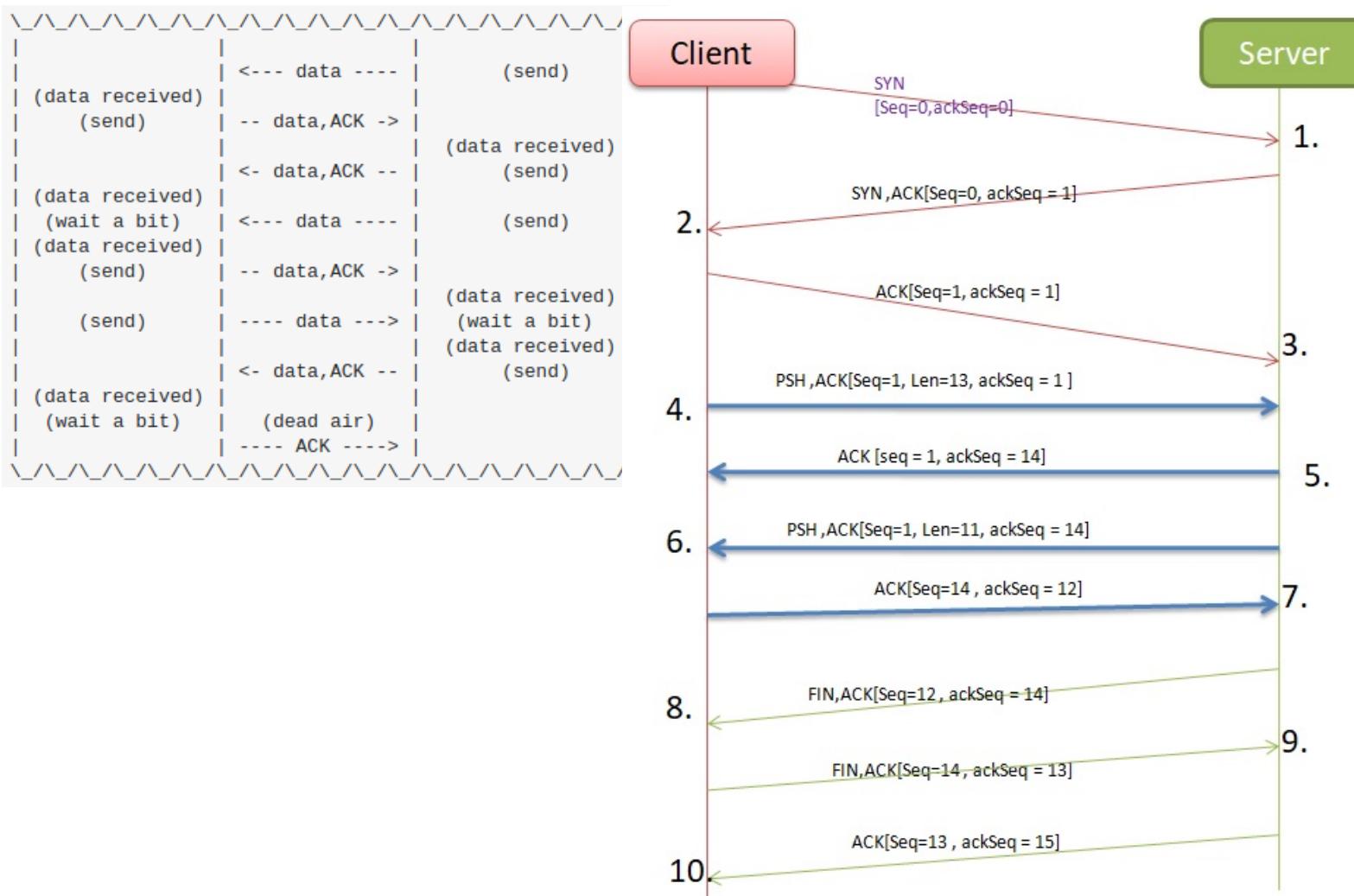
1. Klient chce zahájit spojení a zasílá paket s příznakem **SYN**, do tohoto paketu také vygeneruje náhodné **Sequence Number**, řekněme **X** (na obrázku je to 100) pro tok od klienta k serveru, Acknowledgment Number nevyplňuje.
2. Server na **SYN** paket odpovídá paketem s příznaky **SYN** a **ACK**, také generuje náhodné **Sequence Number**, řekněme **Y** (na obrázku je to 300), pro tok od serveru ke klientovi. Server navíc vyplní **Acknowledgment Number** jako **X + 1** (na obrázku  $100 + 1 = 101$ ), čímž zároveň potvrzuje přijetí prvního **SYN** paketu od klienta.
3. Klient potvrzuje zahájení komunikace pomocí **ACK** paketu reagující na **SYN+ACK** paket. Jako **Acknowledgment Number** vyplní hodnotu **Y + 1** (na obrázku  $300 + 1 = 301$ ) a inkrementuje svoje **Sequence Number**, které také odesílá.
4. Je vytvořeno spojení (respektive lze na to nahlížet jako 2 spojení) a klient se serverem si mohou vyměňovat **bezztrátově** data, klient i server jsou ve stavu **ESTABLISHED**.

2-way handshake **nestačí** kvůli možnosti znovuposlání, kdy by mohlo dojít k ponechání **napůl otevřeného spojení**, viz obrázek. Z toho důvodu se používá 3-way handshake.



## Komunikace v TCP

Při komunikaci jsou zvětšována **sériová** a **potvrzovací čísla** na základě počtu přijatých/odeslaných **bajtů** (potvrzuje se každý byte a případně jedním ACK paketem může být potvrzeno více paketů s daty). Klient i server by měl hned po obdržení paketu zasílat jeho potvrzení - **ACK**. V realitě to ale může být implementované v OS tak, že se čeká kolem 200 ms, jestli nedojde nějakou vyšší vrstvou k vytvoření odpovědi, kterou by zaslal v ACK paketu a zaslání ACK by tak bylo "zdarma", říká se tomu **Piggybacking** (např. klient zasílá GET dotaz na HTTP server a ten zvládne vygenerovat odpověď do 200 ms a pomocí schránky jí odesílá zpět, OS ji vezme a připojí ji k ACK paketu).



OS si navíc u každého odeslaného paketu ukládá čas jeho odeslání (spouští časovač). V případě, že **vyprší časovač**, znamená to, že se ztratil buď samotný paket s daty nebo paket potvrzující jejich přijetí (nebo ani jedno a jen je pomalá síť). Ve všech případech se **datový paket odesílá znovu**. V případě **duplicitního ACKu** se data také znovu odesílají (značí to, že některý paket se ztratil, nebo že došlo k jejich prohození, druhá stanice již přijala paket následující, ale stále jí jeden paket v řadě chybí, tak posílá ACK předchozího, aby byl opět zaslán ten chybějící paket v řadě). [TCP Duplicate Acks Explained // How to Troubleshoot Them](#)

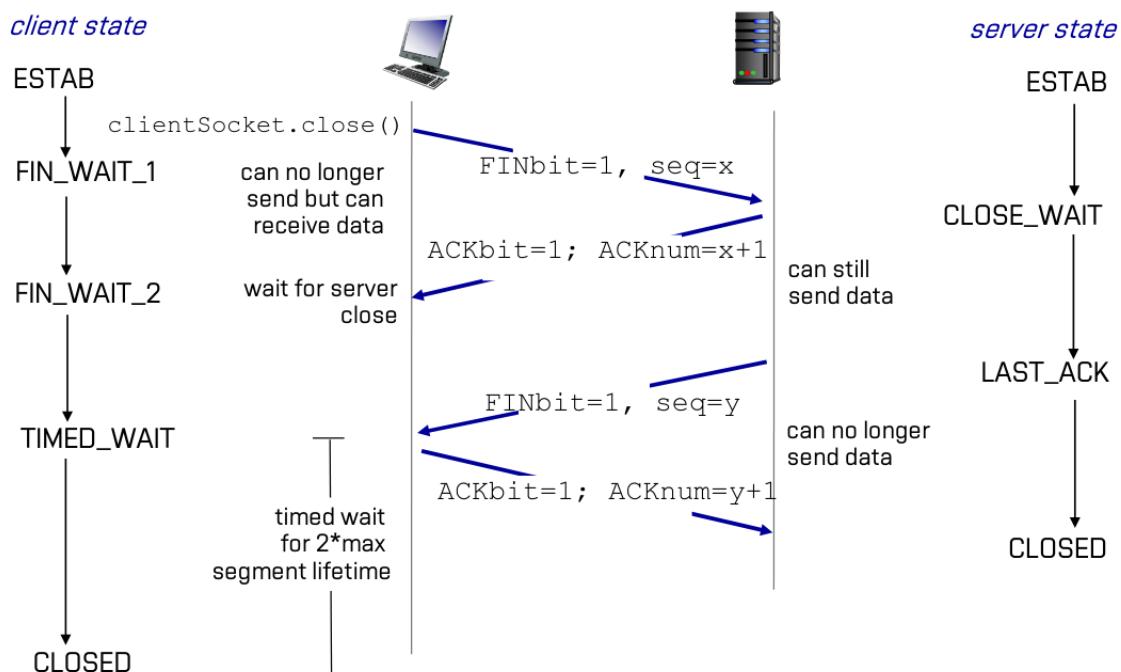
## Ukončení spojení

Ukončení spojení probíhá **dvoufázově** (také někdy označováno dvojice 2 way handshakes nebo také 4-way handshake):

- Počítač **A** (řekněme klient) se rozhodne ukončit spojení a zasílá paket s příznakem **FIN** (ten může i nést nějaká data) počítači **B** (řekněme server). Tím počítač **A** indikuje, že už nebude zasílat žádná **nová** data, ale pořád přijímá. (TCP je bezztrátový, může opakovat zaslání paketů, které se ztratily, a

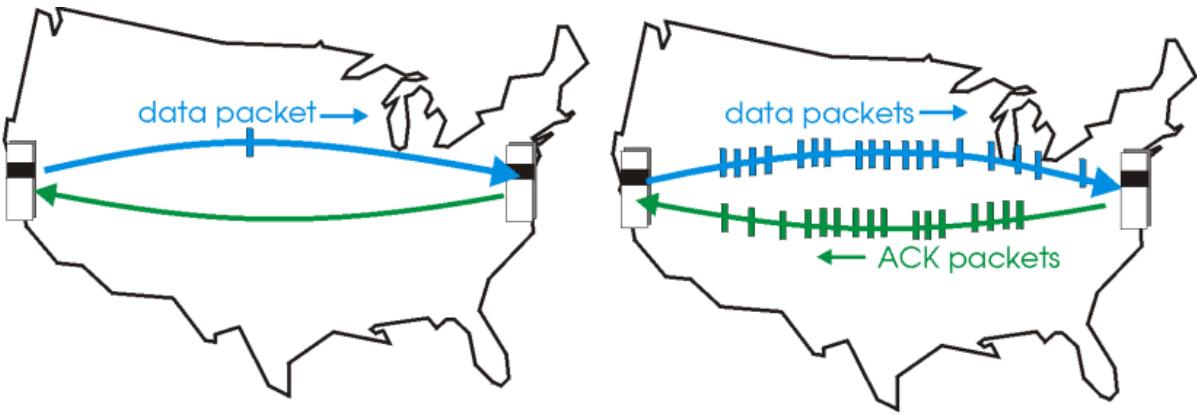
potvrzovat datové paketu od B). Počítač A přechází do stavu **FIN\_WAIT\_1**.

- Počítač B přijme **FIN** paket od A a odpovídá mu s ACK paketem, čímž přechází do stavu **WAIT\_CLOSE**. Po této operaci počítač B ještě může dále počítači A posílat rozpracovaná data (nebo případně rovnou s ACK příznakem může také zaslat FIN).
- Počítač A přijme **ACK** paket od B, přejde do stavu **FIN\_WAIT\_2** a čeká (případně znova posílá ztracené pakety nebo odpovídá na zbytek dat od serveru s ACK, nebo může taky zaslat paket RST, kterým ukončí spojení ihned, ale za cenu možné ztráty dat).
- Počítač B už odesal všechna data a pošle paket **FIN** pro potvrzení úplného konce komunikace. Přechází do stavu **LAST\_ACK**.
- Počítač A přijme paket **FIN**, odpovídá s paketem **ACK** a přechází do stavu **TIMED\_WAIT**, kde pouze čeká, jestli počítač B nezašle znova **FIN**, to nastane pouze v případě, že původní **ACK** na **FIN** se ztratil.
- Počítač B přijme **ACK** a přechází do stavu **CLOSED**.
- Počítači A vyprší čekání a přechází také do stavu **CLOSED**.



## Pipelining (zřetězené protokoly)

Na rozdíl od přístupu **Stop-and-wait**, který zašle paket, čeká na ACK a až poté zasílá další paket, dochází k odesílání více paketů naráz, aniž bychom čekali na potvrzení každého. Díky tomu je možné **maximalizovat** využití linky a **minimalizovat čekání související s potvrzováním**. Jsou k němu 2 běžné přístupy.

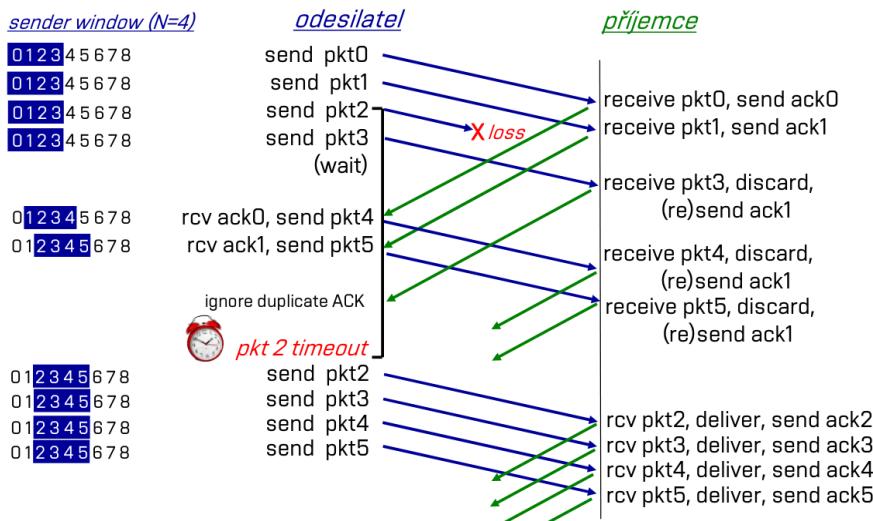


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

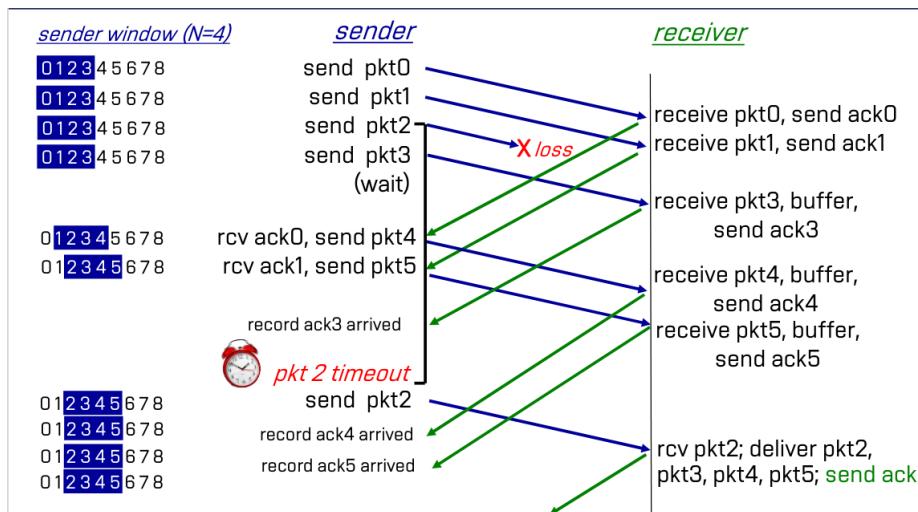
### Go-back-N

- Odesílatel má **sliding window** pro **N** paketů, každý nepotvrzený paket má časovač. V případě **vypršení** nejstaršího nepotvrzeného časovače nebo obdržení **duplicitního ACK** (příjemce odesílá duplicitní ACK, pokud mu přijde paket ve špatném pořadí a ten **zahazuje**), znovu proběhne zaslání celého aktuálního okna (nepotvrzený paket je na začátku okna).
- Příjemce může potvrdit **jedním ACK** více předchozích paketů (**kumulativní potvrzení**, např. příjemce zašle ACK4, ACK5 a ACK6. ACK4 a ACK5 se ztratí, ACK6 ale přijde, odesílatel tak považuje i ACK4 a ACK5 za potvrzené - odesílatel by jinak neodesílal ACK6, kdyby mu nedorazil paket 4 a 5. Případně někdy se ani příjemce nemusí pokoušet odesílat ACK na všechny pakety a rovnou pouze na poslední).
- **výhody**: příjemce **nepotřebuje vyrovnávací paměť** pro ukládání paketů, které přišly mimo pořadí.
- **nevýhody**: opakovaně se zasílají pakety, které už jednou byly zaslány a mohly úspěšně přijít, ale byly zahozeny.



## Selective Repeat (SACK)

- Odesílatel má **sliding window** pro **N** paketů, stejně jako u Go-Back-N.
- Příjemce selektivně potvrzuje přijatý paket (pro každý odesílá **individuální** potvrzení).
- Příjemce si ukládá pakety, které jsou mimo pořadí do vyrovnávací paměti.
- Odesílatel má **časovač pro každý paket** a znova posílá **jen ty nepotvrzené**, nikoliv celé okno.
- **výhody:** zasílají se opravdu pakety, které nebyly doručeny, snižuje množství paketů v síti a případnou šanci na zahlcení.
- **nevýhody:** příjemce musí implementovat vyrovnávací paměť.
- **Používá se dnes.**



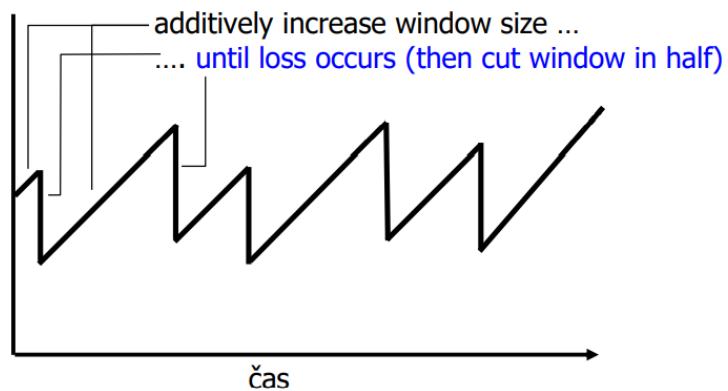
## Řízení zahlcení

Zahlcení sítě nastává, když objem přenášených dat je **větší než přenosová kapacita** linky. Směrovače mají vyrovnávací paměť (frontu), do které ukládají příchozí pakety před zpracováním. V případě, že se tato fronta zaplní (příliš mnoho paketů), začnou se **zahazovat**. Zahlcení má tendenci se zhoršovat (pokusy o znovuzasílání způsobují ještě vyšší objem dat)

TCP má implementované mechanismy řízení zahlcení, které je založeno na **sledování ztrát paketu**. V případě, že se pakety neztrácí TCP **zvyšuje rychlosť** přenosu (**zvětšuje klouzavé okno** a tím snižuje čas čekání na ACK). Pokud dojde ke ztrátě, značí to problém na lince (zahlcení) -> **snižuje rychlosť**. Snahou je co nejrychleji najít ideální velikost klouzavého okna, proto nejprve roste jeho velikost **exponenciálně** (slow start), po ztrátě paketů už jen **lineárně** (congestion avoidance). Běžné jsou 2 algoritmy - Tahoe a Reno.

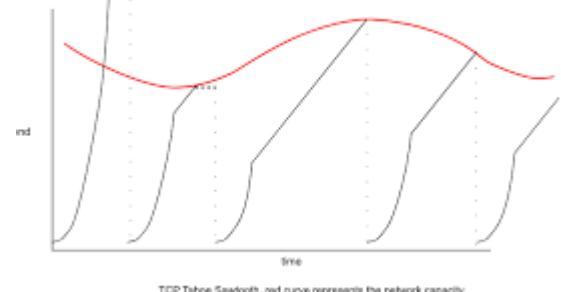
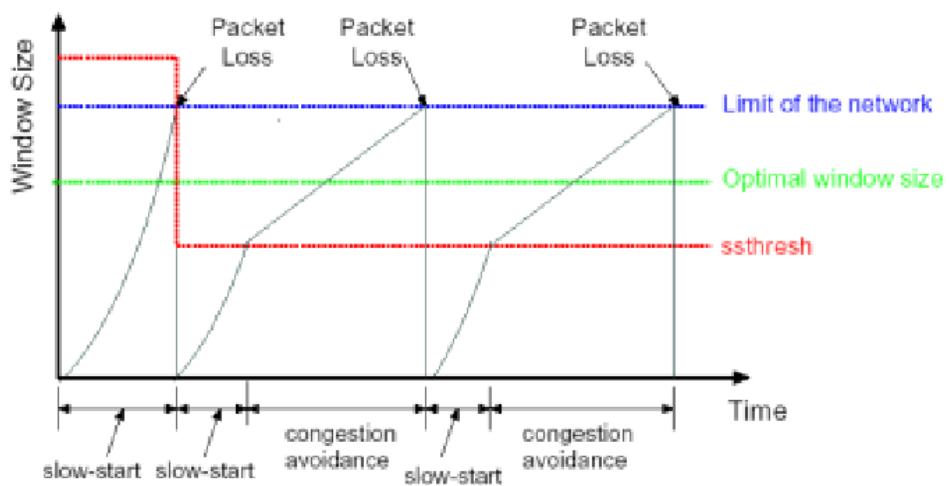
## Additive Increase, Multiplicative Decrease (AIMD)

Odesílatel zvedá rychlosť odesílání lineárně (zvětšuje velikost okna o 1). V případě ztráty sníží rychlosť na polovinu (zmenší okno na polovinu aktuální velikosti).



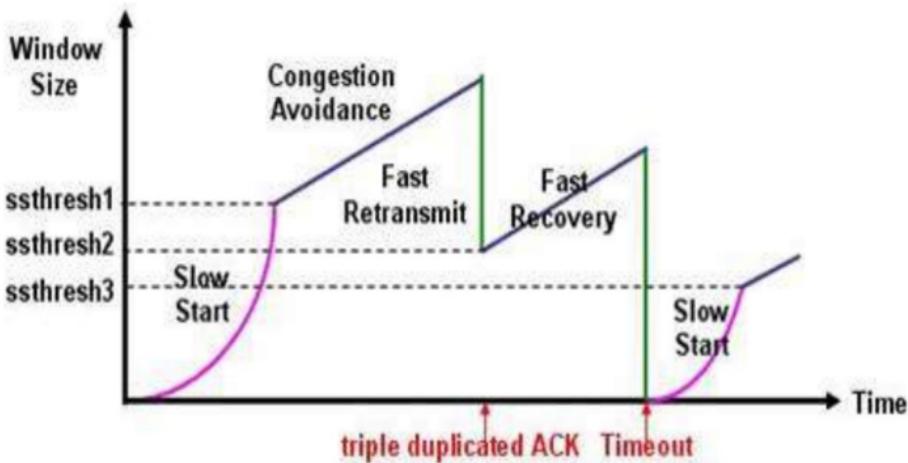
## Tahoe

Tahoe po ztrátě paketu resetuje velikost okna na **0**, pro hledání správné velikosti využívá proměnnou (slow start threshold). V případě, že je velikost okna menší než **ssthresh**, probíhá **slow start** (exponenciální nárůst okna), **jinak congestion avoidance** (lineární nárůst okna). V případě ztráty paketu se **ssthresh** zmenší na **polovinu aktuální velikosti klouzavého okna**. (obr. vlevo počítá ze stabilní rychlostí sítě, reálnější je ale obrázek vpravo, kde propustnost kolísá, lze zde také vidět změnu ssthresh)



## Reno

Reno je vylepšením Tahoe zavádí ještě jeden stav, a to **Fast Recovery** (lineární nárůst velikosti okna). U reno ztráta paketu (obdržení duplicitního ACK (konkrétně 3 duplikáty) nebo ACK mimo pořadí u SACK) nezpůsobí zásadní propad v propustnosti a velikost okna se **zmenší na polovinu**. Až v případě že dojde k **vypršení časovače**, dochází k restartování velikosti okna **na minimum**.



## NewReno

Detekuje **vícenásobnou ztrátu** paketů v jednom okně. Po první ztrátě přechází do **Fast Recovery** a při další ztrátě ve **stejném okně nesnižuje** velikost na polovinu. Velikost okna snižuje, až je zase ztráta mimo okno, na kterém došlo k první ztrátě, viz

<https://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf>

## Síťová vrstva

Síťová vrstva L3 se stará o doručení komunikace mezi stroji. Základními protokoly jsou **IPv4** a **IPv6** a dále protokoly s nimi související (**ICMP**, **DHCP**, apod.). Adresace probíhá pomocí IP adres (IPv4 32b, IPv6 128b). Důležitým prvkem této vrstvy je **směrovač** (router), který na základě směrovacích tabulek pakety **směruje síť**. Síťová vrstva je nespojová (**connection-less**), tj. doručení probíhá na bázi **best-effort**. IP pakety umožňují data rozdělit na segmentu, tak aby je bylo možné odeslat přes síť a na cílové stanici opět spojit (segmentují se kvůli MTU).

Hlavíčka **IPv4** obsahuje (nemá fixní délku):

- verzi - verze protokolu 0x4,
- Type of Service (ToS), využívá se pro QoS (viz dále),
- **délku**,
- **zdrojovou a cílovou IP adresu**,
- **TTL** = time to live (na každém routeru se snižuje jako prevence zacyklení),
- **Offset pro fragmentaci** - udává pozici dat v původním datagramu, který byl segmentován po násobcích 8 byte → data jsou odesílána jako **násobky 8 B**,
- **číslo protokolu vyšší vrstvy** (TCP nebo UDP).

Formát IP datagramu

| Bajty                       | 0                             | 1               | 2                         | 3                          |
|-----------------------------|-------------------------------|-----------------|---------------------------|----------------------------|
| Bajty 0 až 3                | verze                         | IHL             | typ služby                | celková délka              |
| Bajty 4 až 7                | identifikace                  |                 | příznaky (3 bity)         | offset fragmentu (13 bitů) |
| Bajty 8 až 11               | TTL                           | číslo protokolu | kontrolní součet hlavičky |                            |
| Bajty 12 až 15              | zdrojová adresa               |                 |                           | cílová adresa              |
| Bajty 16 až 19              | rozšířená nepovinná nastavení |                 |                           |                            |
| Bajty 20 až ((IHL × 4) - 1) | data                          |                 |                           |                            |
| ...                         |                               |                 |                           |                            |

## Maximum Transmission Unit (MTU)

MTU je největší jednotka, kterou je médium schopno přenést (na internetu jde obvykle o MTU ethernetu, která je 1500 B), pokud data z vyšší vrstvy přesahují MTU, nastává fragmentace paketů (což je naznačeno v hlavičce pomocí offsetu a provádí ji většinou router). Datagram sestavuje až koncová stanice. Případně router, na kterém by mělo dojít k fragmentaci může zaslat ICMP zprávu Packet Too Big a odesílatel musí provést fragmentování.

Adresování probíhá pomocí 32b IP adresy, která se skládá z **network ID** (adresa sítě) a **host ID** (adresa hosta). IP adresy sítím jsou přidělovány organizací **ICANN**, která přiděluje bloky **nadnárodním registrátorům**, ti je poté delegují **regionálním registrátorům** (RIR). Rozlišujeme **třídní** a **beztřídní** adresování.

### Třídní adresování

Dle třídy IP adresy je jasně dán, kde končí network ID v IP adrese (maska sítě). Třídu rozlišujeme dle prefixu bitů IP adresy (prvních několik 1 bitů po první 0).

Třídy IP adres

| Třída | začátek (bin) | 1. bajt | standardní maska | bitů sítě | bitů stanice | sítí                   | stanic v každé síti       |
|-------|---------------|---------|------------------|-----------|--------------|------------------------|---------------------------|
| A     | 0             | 0–127   | 255.0.0.0        | 8         | 24           | $2^7 = 128$            | $2^{24}-2 = 16\ 777\ 214$ |
| B     | 10            | 128–191 | 255.255.0.0      | 16        | 16           | $2^{14} = 16384$       | $2^{16}-2 = 65\ 534$      |
| C     | 110           | 192–223 | 255.255.255.0    | 24        | 8            | $2^{21} = 2\ 097\ 152$ | $2^8-2 = 254$             |
| D     | 1110          | 224–239 |                  |           |              | multicast              |                           |
| E     | 1111          | 240–255 |                  |           |              | vyhrazeno jako rezerva |                           |

Rozsahy IP adres a masky sítě

| Třída | 1. bajt | minimum   | maximum         | maska podsítě   |
|-------|---------|-----------|-----------------|-----------------|
| A     | 0–127   | 0.0.0.0   | 127.255.255.255 | 255.0.0.0       |
| B     | 128–191 | 128.0.0.0 | 191.255.255.255 | 255.255.0.0     |
| C     | 192–223 | 192.0.0.0 | 223.255.255.255 | 255.255.255.0   |
| D     | 224–239 | 224.0.0.0 | 239.255.255.255 | 255.255.255.255 |
| E     | 240–255 | 240.0.0.0 | 255.255.255.255 | —               |

Rozlišujeme několik speciálních adres. Pokud hostID je **0**, pak adresa je **adresou sítě**. V případě, že hostID je -1 (samé 1 bitově), pak se jedná o **broadcastovou adresu** v dané síti.

### Beztřídní adresování

VLSM = variable length subnet mask. Maska není specifikována třídou, ale je **"dynamická"**, součástí samotné IP adresy. Například **147.229.176.14/23** udává, že maska má 23 jedničkových bitů (255.255.254.0). Díky tomu je možné provádět například subnetting, kdy jednu větší síť rozdělíme na více menších podsítí (například z jedné /24 síťe můžeme udělat 2 /25 tím, že napevno nastavíme 25. bit zleva na 1, nebo 0).

## Protokol ICMP

Protokol ICMP je podpůrný protokol pro IPv4, slouží pro aktivní diagnostiku sítě.

Důležitými typy zpráv jsou:

- echo (ping),
- host/network/port unreachable - signalizace nedoručení paketu, protože nebyl nalezen příjemce.
- TTL expired - signalizace zahození paketu, protože prošel přes příliš mnoho směrovačů.

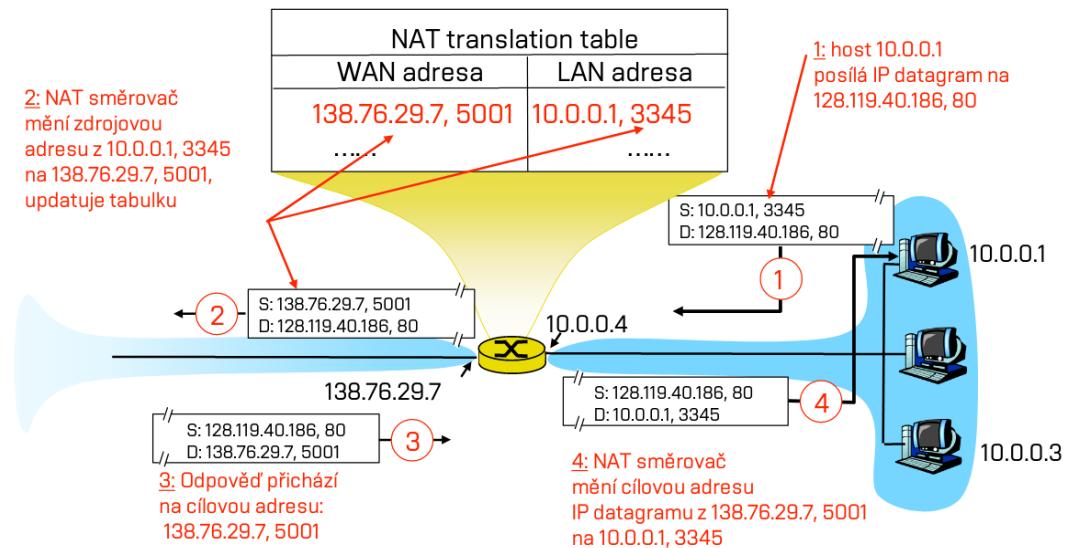
## Protokol DHCP

Slouží k dynamickému přidělování IP adres v síti, klient-server paradigm. Aplikační protokol komunikující pomocí UDP na portu 67 a 68. Komunikace probíhá následovně:

1. Nově připojené zařízení k síti vyšle DHCP **Discover** broadcast zprávu
2. Server(y) mu odpoví se zprávou DHCP **Offer**, kdy nabízí IP adresu, default gateway, DNS server
3. Klient si jednu z nabídek vybere a potvrdí ji opět broadcastově pomocí DHCP **Request**
4. Server to potvrdí pomocí DHCP **Acknowledgement**

## Network Address Translation (NAT)

NAT je mechanismus, který pomáhá "šetřit" IP adresy. Funguje na tom principu, že v lokální síti mají stroje pouze lokálně platnou IP adresu (např. 192.168.1.113 nebo 10.0.0.0/8), ovšem z lokální sítě veškerý provoz odchází přes router, kde proběhne **překlad na reálnou adresu sítě**. Běžně se používá Port overloading:



## IPv6

Protokol IP verze 6 vznikl s motivací vyřešit nedostatek adres IPv4. Adresy mají 128b, což poskytuje mnohem **větší adresný prostor**. Má narozdíl od IPv4 **pevnou**

**velikost hlavičky** (40 bytů) a **nepodporuje broadcast**. Umožňuje, aby jedno rozhraní mělo více než jednu IPv6 adresu.

IP adresa má zkrácený zápis, tvoříme jej následovně:

- úvodní nuly v čtveřicích hexa číslic vynechat (0000 zkrátit jako 0)
- nejdelší sekvenci nul nahradit za :: (při více stejně dlouhých sekvencích nahrazujeme tu 1.)

Podobnou roli jako ICMP pro IPv4 zastává **ICMPv6** pro IPv6, ovšem dále obsahuje ekvivalenty k **ARP** (překlad IP adres na MAC adresy) a **IGMP** (protokol pro multicast). Adresy jsou v síti přidělovány buď pomocí **DHCPv6**, nebo bezstavově pomocí **Router Advertisement** a **Router Solicitation** zpráv.

## IPSec

Jedná se o **dva protokoly**, které zajišťují **důvěrnost, integritu dat, autentizaci a ochranu proti přehráni** na úrovni síťové vrstvy, tak že vyšší vrstvy jsou již zabezpečené. Využívá se pro vytváření **VPN** (virtual private network).

### Authentication Header (AH)

Zajišťuje **integritu dat** (heš se počítá nad položkami hlavičky, které se během přenosu nemění, a dat), **autentizaci a chrání proti přehráni**. Přenášená data nejsou šifrována. Vždy musí být před ESP, pokud je také použito.

### Encapsulating Security Payload (ESP)

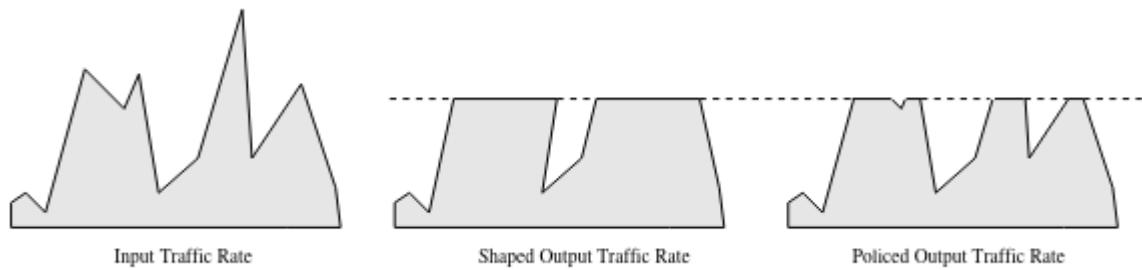
Zapouzdřuje a chrání data IP paketu. Zajišťuje **autentizaci, důvěrnost, integritu dat a chrání proti přehráni**. ESP pracuje ve dvou režimech:

- **transportní režim**: chráněn je pouze payload (tj. vyšší vrstvy),
- **tunelovací režim**: chrání i IP hlavičku tak, že je celý paket zabalen ještě do dalšího IP paketu.

## Quality of Service (QoS)

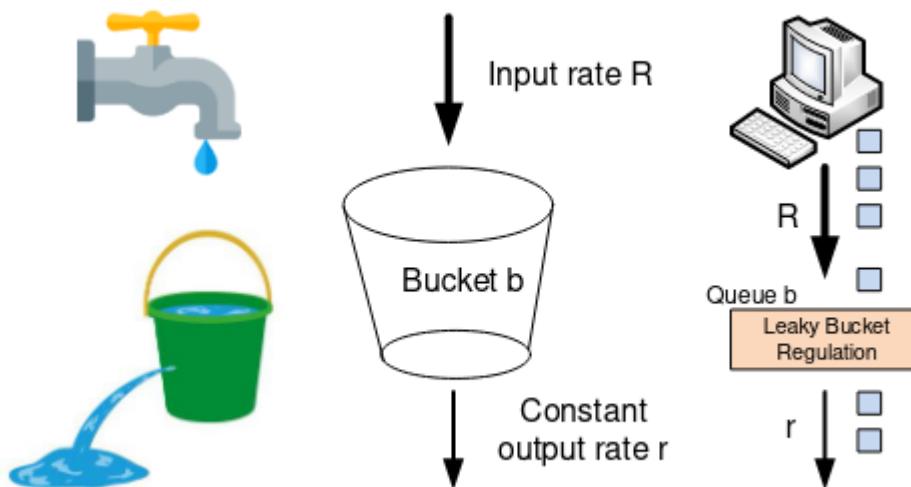
Jak bylo zmíněno výše, IP vrstva funguje na bázi best-effort delivery, nezajišťuje tedy v základu kvalitu služeb. Směrovače se to do jisté míry snaží kompenzovat a zajistit co nejspolehlivější doručení. Základními přístupy ke zvládání "špiček paketů" na směrovači jsou shaping a policing:

- Při **policingu** se provoz ořezává, nezpracovatelné pakety jsou zahoveny.
  - **výhody**: nezpůsobuje zpoždění a je jednodušší na implementaci,
  - **nevýhody**: způsobuje větší ztráty paketů, což může způsobovat ještě větší vytížení sítě.
- Při **shapingu** se provoz rozkládá pomocí bufferů, směrovač si zapamatuje některé pakety a odesílá je později.
  - **výhody**: méně paketů je zahoveno,
  - **nevýhody**: zanáší zpoždění (to je ale stále lepší než, když je paket zahoven a musí se odeslat znova)



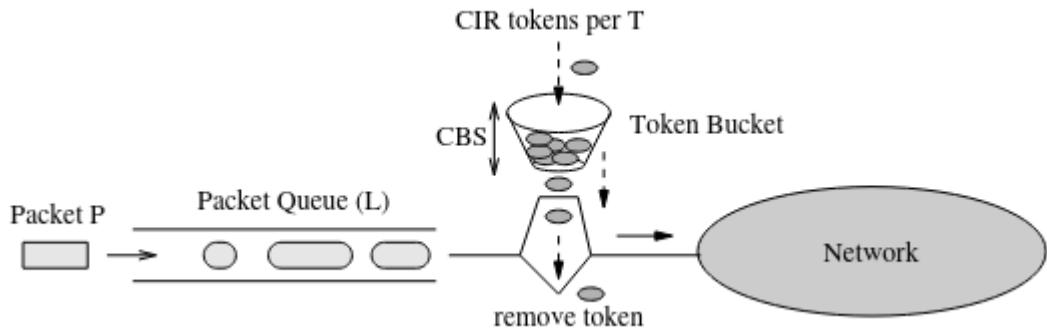
Z pohledu zpracování provozu na směrovači máme několik přístupů:

- obyčejná **FIFO fronta**,
- **prioritní FIFO fronta** (klasifikátor rozdělí provoz do front, data se zpracovávají v pořadí dle priorit front, nízko prioritní ale mohou vyhlaďovět). Klasifikace může být provedena podle protokolu, ToS v IP hlavičce.
- **Cyklické round robin fronty** - férové rozdělení výstupu mezi fronty, pokud je jedna fronta využívána daleko více než zbylé, zahazuje se pouze z ní, ostatní se stíhají zpracovat. To může být ale nevhoda, řešení viz dále.
- **Weighted Fair Queues** - podobně jako round robin, ale propustnost front je dána poměrem vah (z front se odebírá v cyklech, v každém cyklu se z každé fronty odebere různý počet paketů/bytů, čím větší priorita, tím více. Nedochází ale k vyhlaďovění nízko prioritních).
- **Tekoucí vědro** - nezávisle na vstupní rychlosti je na výstupu vždy stejná rychlosť, v případě přetečení dojde k zahození - způsobuje **shaping**, pokud má vědro větší kapacitu než 0 (existuje paměť - FIFO), jinak se jedná o **policing**.



- **Zásobník žetonů** - do zásobníku žetonů se sypou s **konstantní** rychlostí žetony (až do určitého omezeného počtu). Jeden žeton představuje určité množství dat, které lze přenést v **bytech**. Pokud je v zásobníku dostatek žetonů pro přeposlání paketu o určité délce, je **přeposlán** a žetony jsou **odebrány**. Pokud je zásobník plný, lze takhle odeslat v krátkém čase velké množství dat (burst), umí se tedy vyrovnat s krátkodobou špičkou. Poté se ale zásobník **vyprázdní** a musí se čekat, dokud se zase nenaplní žetony. V tento

okamžik jsou příchozí paketu **bud' zahazovány, nebo musí čekat ve frontě**.



## IntServ

Integrované služby implementují QoS formou **rezervace zdrojů** (před každým přenosem nebo při změně cesty) pomocí protokolu **RSVP** (stanice zažádá o spojení v určité kvalitě: **garantované služby** - garantuje vyhrazení nějakého pásma, **kontrolovaná zátěž** - kvalita provozu blížící se nezatíženému prvku, nebo **best-effort**). Protokol RSVP prakticky nelze použít na globálním internetu. Zajištění požadovaného přenosového pásma nemusí být na zatížené síti vůbec možné.

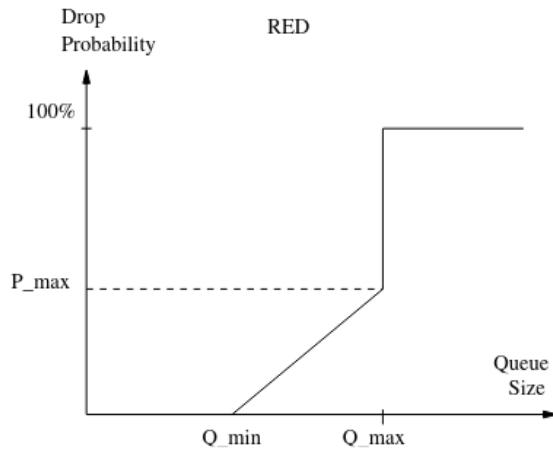
## Diffserv

Diferencované služby fungují na principu **značení provozu** na směrovači a následné **prioritizace** dle značek. Značení probíhá do IP hlavičky, pole ToS (type of service). Podle **kategorie** poté **rozlišujeme**, zda má být paket **přednostně** zpracován nebo **pouze best-effort** (rozdělení do kategorií provádí hraniční prvky sítě, tj. směrovače). Například pro **VoIP** by byla nastaveno **přednostní doporučení** s určitou propustností, protože se jedná o časově citlivou aplikaci.

## Řízení zahlcení na L3 (RED, WRED)

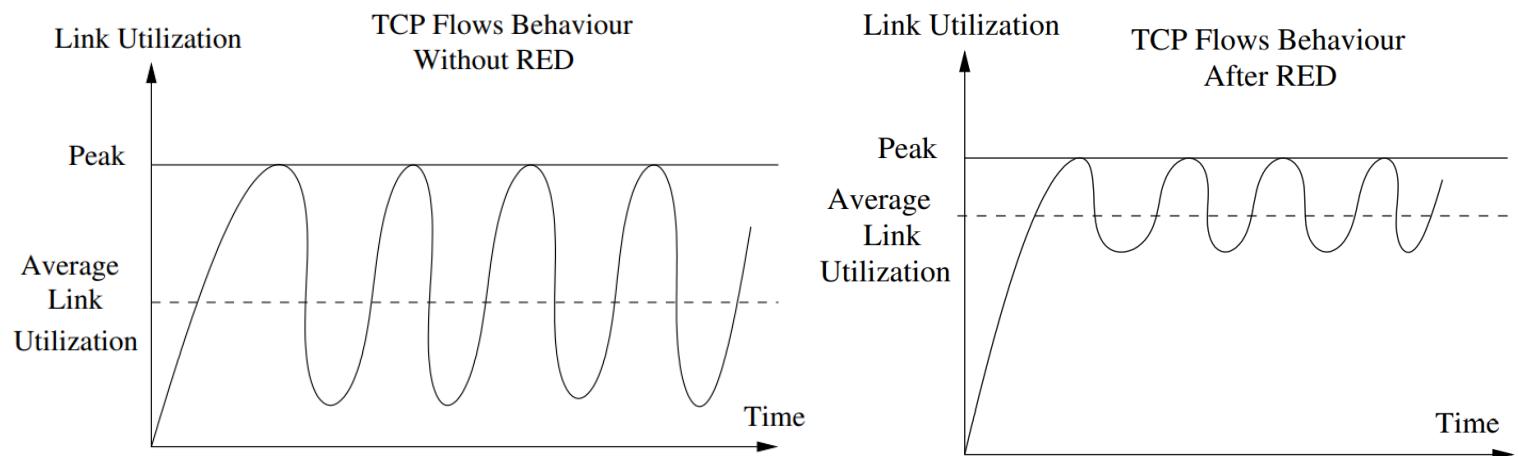
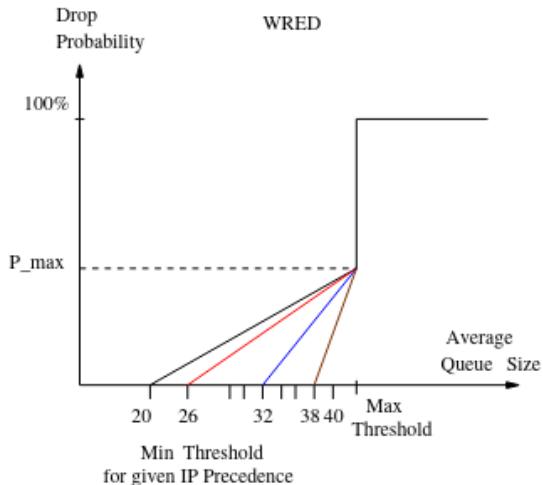
Jak bylo vysvětleno výše, **TCP řeší řízení zahlcení**, problémem ovšem je, že se jedná o L4 protokol a směrovače o něm tedy neví. V případě, že komunikují stanice zároveň **UDP** a **TCP** a linka není příliš rychlá, může lehce dojít k **vyhladovění TCP**, protože UDP aplikace bude posílat naplně pakety bez omezení, zatímco TCP budou zahazovány pakety a bude se snižovat velikost klouzavého okna, proto bude zpomalovat.

K řešení tohoto problému se na směrovačích používá algoritmus RED = **Random Early Detection**. Od určité míry zahlcení začne směrovač **preventivně zahazovat** pakety s určitou pravděpodobností. **Qmin** je zaplnění fronty, od kterého zahazujeme, **Qmax** maximální zaplnění fronty, **Qavg** současné zaplnění. Před Qmin nedochází k zahazování paketů, od **Qmin** roste pravděpodobnost zahození lineárně až u **Qmax** je 100% a je zahazován každý paket. Při **zvětšení Qmin** se pakety začnou zahazovat později, ale hrozí úplné **zaplnění fronty**. Při **nízkém Qmin** se pakety zahazují dřívěji, ale možná někdy **zbytečně** (třeba by se fronta nenaplnila).



$$\bullet \quad P_a = P_{\text{min}} \frac{Q_{\text{avg}} - Q_{\text{min}}}{Q_{\text{max}} - Q_{\text{min}}},$$

Variantou RED je potom vážený RED (Weighted RED, WRED), kdy Qmin je definováno zvlášť pro jednotlivé druhy provozu (například dle pole **ToS**), pakety s **nižší prioritou** mají **nižší Qmin** a začnou být **dříve zahazovány**. Předpokládá se, že to **uvolní síť** dostatečně, aby **nemuselo** dojít k zahazování prioritnějších paketů .



## 44. Směrování a zabezpečení přenosů v počítačových sítích (algoritmy Link-State, Distance-Vector, šifrování, autentizace a integrita dat).

Směrování (routing) je **určování cest paketů** v mezi různými počítačovými sítěmi (které jsou ale na jednom internetu). Směrování zajišťují **směrovače** (routery - pomocí **IP adres** na **síťové vrstvě - L3**). Úkolem směrování je doručit paket adresátovi (který je v jiné síti), pokud možno co **nejefektivnější** cestou (tj. co nejrychleji).

- **1. poznámka:** v rámci **jedné** sítě se **nesměruje ale přepíná**, přepínání provádí přepínač (**switch**) a děje se na **linkové vrstvě (L2)**. Přepínají se **rámce**.
- **2. poznámka:**
  - PDU na **linkové vrstvě** je **rámec** (frame),
  - PDU na **síťové vrstvě** je **paket** (packet),

- PDU na transportní vrstvě je:
  - tok (stream)** pro TCP (viz SOCK\_STREAM),
  - datagram** pro UDP (viz SOCK\_DGRAM),

Pojem **paket** je ale často také považován za data přijatá přes **TCP** a pojmen **datagram** jako data přijatá přes **UDP** a případně jinými způsoby s možnou ztrátou.

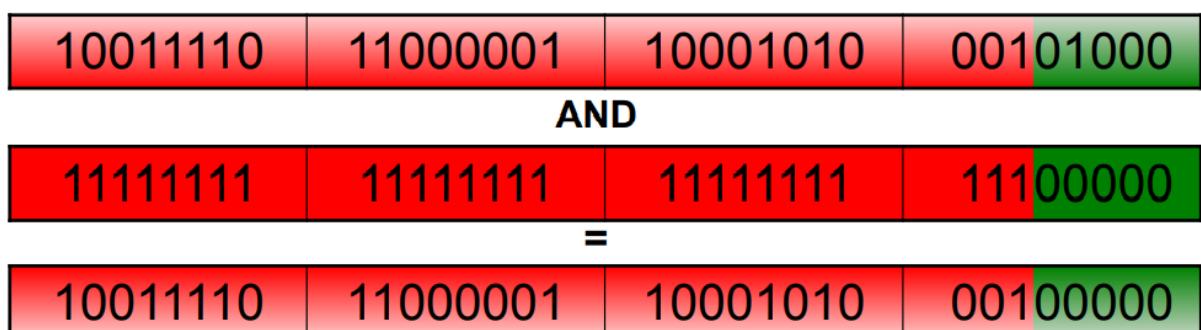
## Třídní směrování (Classful routing)

|         | octet   | octet   | octet   | octet | Class        | High order bits | Start ip address | End ip address  |
|---------|---------|---------|---------|-------|--------------|-----------------|------------------|-----------------|
| Class A | Network | Host    | Host    | Host  | A            | 0               | 0.0.0.0          | 127.255.255.255 |
| Class B | Network | Network | Host    | Host  | B            | 10              | 128.0.0.0        | 191.255.255.255 |
| Class C | Network | Network | Network | Host  | C            | 110             | 192.0.0.0        | 223.255.255.255 |
|         |         |         |         |       | Multicast    | 1110            | 224.0.0.0        | 239.255.255.255 |
|         |         |         |         |       | Experimental | 1111            | 240.0.0.0        | 255.255.255.255 |

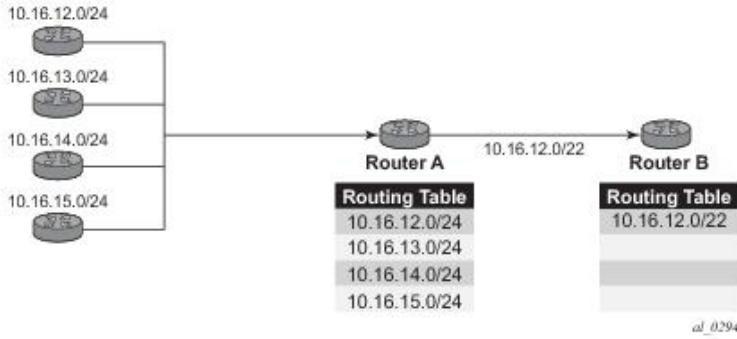
Položky ve **směrovací tabulce** (routing table) tvoří **pouze síťové adresy** o délce **8, 16 a 24** bitů, viz obrázek vlevo. Délka vychází z **masek sítí** jednotlivých tříd. Třídu lze poznat dle nejvyšších bitů, viz obrázek vpravo. V routing table je  **$2^7 + 2^{14} + 2^{21} = 2\ 113\ 664$**  adres (ne všechny jsou použitelné, např. broadcastové), ale i tak jich je mnoho.

## Beztřídní směrování (Classless routing)

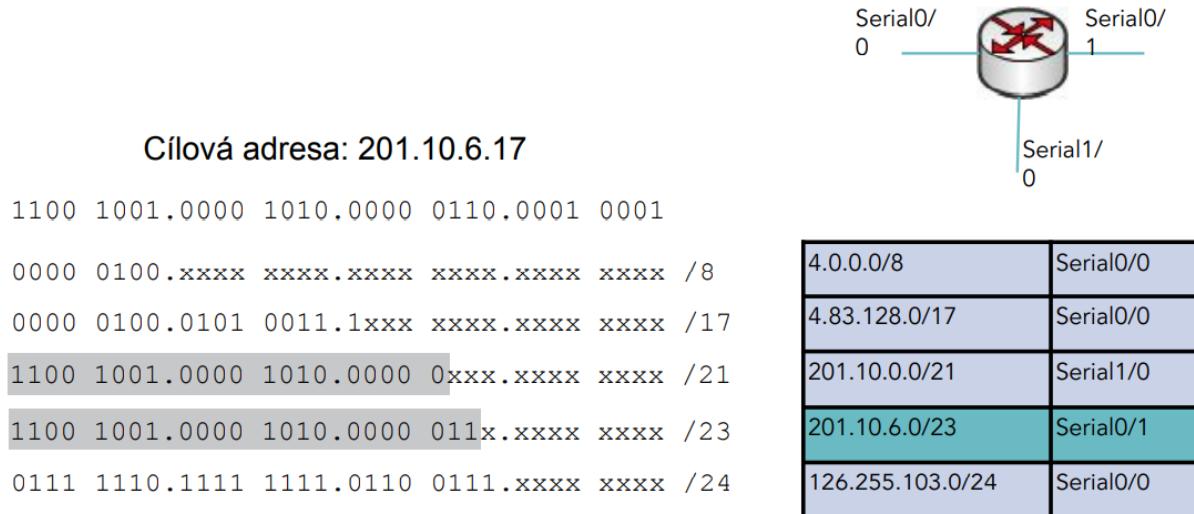
Položky ve směrovací tabulce tvoří **síťové adresy a maska sítě**. Pro každou adresu se na základě **masky sítě** nejdříve určena **síťová adresa**, až pak je provedeno směrování **158.193.138.40 & 255.255.255.224 = 158.193.138.32**.



Pro to, aby se zmenšily směrovací tabulky, **agregují** směrovače adresy sítí, pokud se pro jednu cestu **objeví více adres s rozdílnými LSB** a eliminuje se tak potřeba je rozlišovat.



Na obrázku mohl směrovač B agregovat všechny 4 IP adresy do jedné a snížit tak masku sítě na 22, protože IP adresy obsahují na bitu 9 a 8 všechny kombinace (**00, 01, 10 i 11**). To ale znamená, že když na směrovač přichází teď adresa s maskou 24, není v tabulce. Výběr cesty, kam bude takový paket poslan, pak probíhá na základě vyhledání **nejdelší shody prefixu** - adresy sítě (**Longest Prefix Match**).



## Link-State protokoly

Jedná se např. o protokoly **Open Shortest Path First** (OSPF) a **Intermediate System to Intermediate System** (IS-IS). Link state směrování je založen na tom, že každý směrovač (router) zná **nejlepší cestu** (cestu s nejnižší cenou dle nějakého hodnocení - asi rychlosť přenosu) do **všech sítí**. Tzn. že každý směrovač **zná celou topologii sítě**. Dosáhneme toho:

- **Flooding:** Každý směrovač **broadcastuje** (posílá všem) Link-State paket obsahující **ceny** cest k jeho **sousedním** směrovačům. Každý směrovač si **ukládá** informace (ceny cest) o směrování od ostatních směrovačů. Pokud směrovač přijde Link-State paket, který **už jednou dostal**, již jej dále **nepřeposílá** (už jej jednou přeposlal). **Problémem** může být **ztráta** Link-State paketů, některý směrovač tak nemusí mít kompletní přehled o síti. Řeší se pomocí **ACK** a případného přeposlání.
- Flooding je nutné **provádět** při **přidání** nového směrovače, při **odebrání** a

případně také **periodicky**, pokud se ceny cest mohou měnit.

Jakmile zná každý směrovač celou topologii sítě, má sestavený **vážený** (ohodnocený) **graf**. Nyní musí každý směrovač vypočítat **nejlepší cestu do všech sítí** (jiných směrovačů), používá se na to **Dijkstrův algoritmus** pro hledání nejkratší cesty/cest v ohodnoceném grafu, viz okruh 25. Následně každý router ví, kam má paket směrovat na základě jeho cílové destinace. (Implementovat pak lze tak, že k jednotlivým výstupům si přiřadí všechny IP adresy, které na ten výstup vedou a provádí porovnávání - pomocí asociativní paměti (adresovatelné obsahem). Pokud navíc jsou si IP adresy podobné, může při porovnávání ignorovat některé bity). [Link-State Routing Algorithm - IP Network Control Plane | Computer Networks Ep 5.2.1 | Kurose & Ross](#)

## Distance-Vector

[Distance-Vector Routing Algorithm - IP Network Layer | Computer Networks Ep. 5.2.2 | Kurose & Ross](#)

Jedná se o protokoly **Enhanced Interior Gateway Routing Protocol** (EIGRP) a **Routing Information Protocol** (RIP). Každý uzel (směrovač) komunikuje **pouze se svými sousedy** (směrovači, se kterými má přímé spojení). Směrovač tak na začátku algoritmu zná cestu pouze k sousedním směrovačům (do vzdálenosti 1) a zapíše ceny těchto cest do své směrovací tabulky, do zbytku sítě nevidí (na příkladech se to vyjadřuje nekonečným ohodnocením, v realitě ani neví, že tam něco existuje).

Algoritmus je **iterativní**.

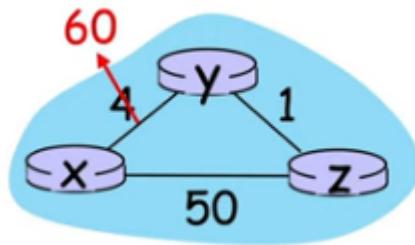
- **1. iterace:** směrovač (a všechny ostatní také) dostane směrovací tabulky od **sousedních** směrovačů, vidí tak už do **vzdálenosti 2**. Na základě získaných informací **aktualizují svou směrovací tabulku** (přidá **nově zviditelněné** směrovače a pokud se dokáže dostat přes jiný směrovač **rychleji** k uzlu, do kterého již cestu zná, aktualizuje i tuto cestu, stejně tak při zdražení cesty).
- **2. iterace:** směrovač opět dostane směrovací tabulky od svých **sousedních** směrovačů, ty ale **mohou být jiné**, než při 1. iteraci (přibyla nově viditelné směrovače, zlevnili se některé cesty, zdražily se některé cesty). Směrovač opět aktualizuje svojí směrovací tabulku, už vidí do **vzdálenosti 3**.
- ...
- **n. iterace:** směrovač dostane směrovací tabulky od svých **sousedů**, už ale **vidí do celé sítě** a nově získané informace **nezpůsobí žádnou změnu** v jeho tabulce. Dále už nepřeposílá svou tabulku, protože se nezměnila. Takhle postupně přestanou odesílat své tabulky všechny směrovače, protože každý už má **ideální směrovací tabulku**.

Z výše uvedeného postupu plyne, že bude třeba **minimálně tolík iterací**, jak je dlouhá **nejdelší nejkratší cesta** (nejkratší znamená, že mezi dvěma uzly nejde již najít nejkratší cestu, nejdelší znamená, že je to pak nejdelší cesta z těchto). Reakce na změny:

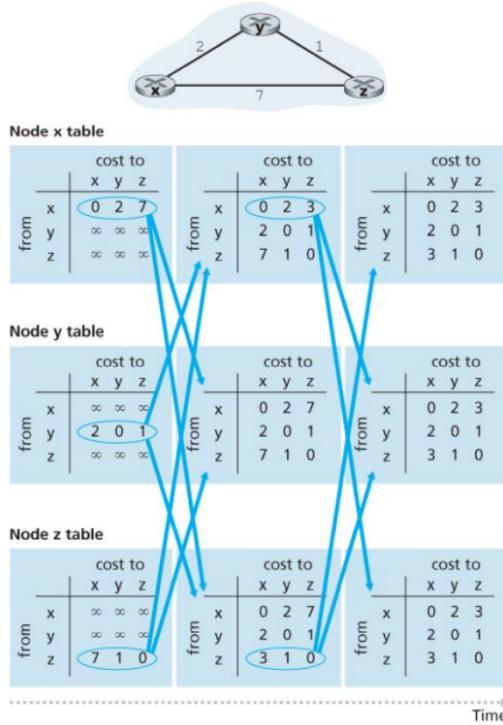
- **snížení cen cesty:** opět si směrovače iterativně vyměňují směrovací tabulky, dokud se směrování neustálí. Probíhá rychle - dobrá zpráva se šíří

rychle.

- **zvýšení cesty:** iterativní šíření. Může ale dojít k zajímavému jevu, kdy dva sousední směrovače využívají cesty **přes sebe navzájem** a ty se tak v každé iteraci **zvětšují o 1**, dokud cena nestoupne natolik, že je použita jiná cesta. Např. na obrázku si bude **y** a **z** vyměňovat tabulky, dokud cena **nepřekročí 50**, pak teprve použijí cestu přes **x**.



Distance-Vector je **distribuovaný algoritmus**, je problematický, pokud nějaký router **lže** nebo **špatně počítá cenu** (hlásí menší než doopravdy). Pak přes něj může být směřován provoz, který ale bude **pomalý**.



## Srovnání Link-State a Distance-Vector

### message complexity

LS:  $n$  routers,  $O(n^2)$  messages sent

DV: exchange between neighbors;  
convergence time varies

### speed of convergence

LS:  $O(n^2)$  algorithm,  $O(n^2)$  messages

- may have oscillations

DV: convergence time varies

- may have routing loops
- count-to-infinity problem

robustness: what happens if router malfunctions, or is compromised?

LS:

- router can advertise incorrect *link* cost
- each router computes only its *own* table

DV:

- DV router can advertise incorrect *path* cost (“I have a *really* low cost path to everywhere”): black-holing
- each router’s table used by others: error propagate thru network

## Zabezpečení na síti

 Zabezpečením sítě chceme zajistit, aby při komunikaci dvou systémů nedocházelo k:

- **falešnému vydávání se** za jeden ze systémů,
- **odposlech** jejich komunikace,
- **zásah do komunikace** pozměněním zasílaných zpráv,
- **opakované zaslání** zachycené zprávy, která již byla předtím doručena.

## Autentizace (Authentication)

Autentizace zajišťuje, že uživatel **je tím, za koho se vydává**. Autentizace obvykle probíhá na základě poskytnutí nějaké **tajné informace**. Tuto informaci může znát nebo k ní mít přístup pouze uživatel, který má být autentizován. Jde například o **uživatelské jméno a heslo, biometrické údaje, bezpečnostní žeton** atd.

Autentizaci lze také zajistit **digitálním podpisem**.

## Důvěrnost (Confidentiality)

Důvěrnost zajišťuje, že obsah zpráv **nemůže číst neoprávněná osoba**. To lze zajistit buď odepřením přístupu ke zprávě, což není ale na internetu obecně možné, nebo **šifrováním** (utajením) zprávy, což sice útočníkovi umožní číst zasílaná data, ale neumožní mu získat informaci, kterou obsahují. Data pro něj **bez dešifrování budou bezcenná**. Nauka o šifrování se nazývá **kryptografie**, v informatice hovoříme především o symetrické a asymetrické kryptografii.

## Symetrická kryptografie

**Symetrická kryptografie** používá jeden **tajný klíč**, kterým odesílatel **šifruje zprávu** a příjemce ji **tím samým klíčem dešifruje**. Nejpoužívanější algoritmy pro symetrické

šifrování jsou v současné době například **AES**, **ChaCha20** či **IDEA**.

Známější algoritmy jako DES, 3DES a Blowfish nejsou dnes již považované za bezpečné, byla u nich objevena bezpečnostní rizika nebo způsob, jak lze šifru překonat hrubou silou v dostatečně krátkém čase. Síla šifry však závisí především na kvalitě klíče a ta, vzhledem k tomu, že většina **symetrických klíčů** je generována jako **pseudonáhodná posloupnost bitů**, je dána jeho délkou. Za dostatečně bezpečné jsou dnes považovány klíče o délce aspoň **128 bitů**, u kterých není u běžných počítačů reálné jejich prolomení hrubou silou v přijatelně krátkém časovém úseku. Pro zajištění bezpečnosti i do budoucna je ale vhodné používat klíče o délce **256 bitů**.

## Asymetrická kryptografie

U **asymetrické kryptografie** se používá **dvojice klíčů, soukromý a veřejný**. Pro tyto klíče platí, že jsou **matematicky svázány**. Odesílatel **šifruje** zprávu **veřejným klíčem**, který může znát **kdokoliv**, buď je veřejně dostupný nebo jej odesílatel od příjemce získá na počátku komunikace, která není ale šifrovaná. Pro dešifrování zprávy je potřeba **soukromý klíč**, ten zná jen a **pouze příjemce**. Tako je zajištěno šifrování v jednom směru komunikace, pro opačný směr se postupuje obdobně. Obvykle si tedy na počátku komunikace respondenti vymění veřejné klíče a až poté probíhá komunikace šifrovaně. Pro asymetrickou kryptografií se používají například algoritmy **RSA** a **EIGamal**, které jsou založené na složitosti výpočtu **diskrétního logaritmu**, či algoritmus **ECC**, který pracuje na bázi **eliptických křivek**.

Stejně jako u symetrické kryptografie závisí **bezpečnost** šifry především na **délce klíče**. Soukromý klíč lze ale na základě jeho **matematické spojitosti** s veřejným klíčem rekonstruovat rychleji než při rekonstrukci hrubou silou. Proto je nutné používat delší klíče než u symetrické kryptografie. Konkrétně pro algoritmus RSA je bezpečnost klíče o délce 1024 bitů ekvivalentní s bezpečností symetrického klíče o délce 80 bitů, 2048 bitový RSA klíč odpovídá symetrickému klíči o délce 112 bitů a pro zajištění bezpečnosti odpovídající symetrickému klíči o délce 256 bitů je třeba použít RSA klíč o délce 15360 bitů, což už je prakticky nepoužitelná délka.

Algoritmus ECC je v tomto směru lepší a pro zajištění bezpečnosti na úrovni 256 bitového symetrického klíče stačí pouze 521 bitový ECC klíč

## Integrita (Data Integrity)

Integritou dat ve smyslu zabezpečení komunikace je myšleno zajištění toho, že data po cestě od odesílatele k příjemci nikdo nezmění. Při komunikaci na internetu je toto řešeno vygenerováním **charakteristiky zprávy** (message digest) o fixní délce, připojením této charakteristiky ke zprávě a jejím odesláním. Příjemce nejdříve oddělí charakteristiku zprávy a získá tak původní zprávu, stejným algoritmem jako odesílatel vygeneruje její charakteristiku a porovná ji s obdrženou charakteristikou. Pokud jsou stejné, nikdo po cestě zprávu nezměnil.

Aby výše uvedený princip fungoval, je nutné pro generování charakteristiky použít **kryptografickou hašovací funkci**. Tato funkce zajišťuje, že je velmi náročné

vygenerovat zprávu, která by **měla požadovanou charakteristiku**, stejně tak náročné je **najít dvě rozdílné zprávy se stejnou charakteristikou** a na základě charakteristiky není možné **zprávu zrekonstruovat**. Dále taková funkce musí při generování charakteristiky zprávy **zohlednit všechny její bity** a i při změně jediného bitu musí být nově vygenerovaná charakteristika od té původní natolik **odlišná**, aby se toho nedalo zneužít. Nejpoužívanější bezpečné kryptografické funkce jsou například **MD6**, **SHA-3** či **BLAKE2**.

I po splnění všech zmíněných předpokladů není stále zajištěno, že zprávu po cestě nikdo nezmění, a to z jednoho prostého důvodu, útočník může změnit zprávu a současně i její charakteristiku. Aby k tomu nedošlo, je nutné použít šifrování, není však nutné šifrovat celou zprávu, stačí **zašifrovat charakteristiku**.

## Neodmítnutelnost (Nonrepudiation)

Neodmítnutelnost zajišťuje, že uživatel nemůže popřít provedení dané akce a poskytuje mu ujištění, že jeho zpráva s požadavkem na provedení této akce byla doručena. K zajištění neodmítnutelnosti se obvykle používá **digitální podpis**. Nejznámějším algoritmem, který implementuje digitální podpis je **DSA**.

Digitální podpis je založen na **asymetrické kryptografii**, ale na rozdíl od zajištění důvěrnosti se zde pro **šifrování používá privátní klíč**. Privátním klíčem se nešifruje zpráva, pouze její charakteristika, která se získá stejným způsobem jako u zajištění integrity dat a má i stejnou funkci, a to zajistit, že zprávu nikdo nezměnil. Pro ověření odesílatele si příjemce obstará od odesílatele **veřejný klíč**, dešifruje zašifrovanou charakteristiku, vygeneruje charakteristiku příchozí zprávy a porovná je. Pokud jsou stejně, nedošlo po cestě ke změně zprávy, ale především odesílatel je opravdu ten, který zprávu podepsal, protože **jen on vlastní privátní klíč**, kterým byla zašifrována charakteristika.

Samozřejmě to není tak jednoduché, vygenerovat pář asymetrických klíčů si může kdokoliv a poté se **podvodně** vydávat za někoho, kým není. Aby k tomu nedocházelo, musí mít odesíatel k veřejnému klíči **certifikát**, který ověřuje jeho identitu. Certifikáty vydávají **certifikační autority**, které při tomto procesu ověří žadatelovu identitu a **podepíší jeho veřejný klíč** svým soukromým. Opět by se dalo namítat, že potenciální útočník si může vytvořit i vlastní certifikační autoritu a **certifikovat si falešné digitální podpisy**. Tomu se dá předejít jen tak, že příjemce si ověří nejen identitu vlastníka veřejného klíče, ale i **pravost certifikační autority**. Ta se zajišťuje tak zvaným **řetězcem důvěry**, který funguje na principu, že nadřazené certifikační autority podepisují klíče svým podřazeným certifikačním autoritám.

Tento řetězec končí u **kořenové certifikační autority**, která již nemůže být dále ověřena, ale vzhledem k tomu, že je pouze jedna, nedá se o její pravost pochybovat.

## Ochrana proti přehrání (replay attack)

Přehrání je druh útoku, u kterého útočník zachytí jinak **validní zabezpečenou komunikaci** a snaží se ji použít **opakovat** nebo ji pouze pozdržet. Jako ochrana proti tomuto typu útoku se nejčastěji používají **časová razítka**. Podle časového razítka se určí stáří zprávy a zprávy starší než stanovená hodnota jsou ignorovány. K tomu, aby časová razítka fungovala, musí být zajištěno, že příjemce i odesíatel mají **synchronizovaný čas** a že maximální stáří zprávy zhruba odpovídá době jejího doručení. U synchronizace času mohou být komplikací rozdílná časové pásma, ve kterých se respondenti nacházejí. Proto se obvykle pro časová razítka používá koordinovaný světový čas. Další problém synchronizace času může být v tom, že každý respondent používá jiný zdroj hodin, který je nepřesný a k tomu nepřesný s opačnou fází, nebo že útočník zaútočí přímo při synchronizaci času a podvrhne nesprávný čas. U stanovení maximálního stáří zprávy je problematická proměnlivá doba jejího doručení, která je způsobena především změnami v zatížení internetové sítě. Z toho je patrné, že pokud bude útočník dostatečně rychlý, může se mu povést i přes použití časového razítka zprávu replikovat. Proto může být časové razítko ještě doplněno například o **sekvenční číslo** zprávy nebo o nějakou jednorázovou informaci. Jak sekvenční číslo tak jednorázová informace vyžadují **uchování stavu**, což může být problematické, protože komunikace na internetu bývá často nestavová.

## Unicast, Broadcast, Anycast a Multicast

Pro jistotu, protože to nikde nezaznělo...

### Unicast

Jedná se o komunikace jednoho odesílatele s jedním příjemcem (**one-to-one**). Komunikace je řešena standardním směrováním a jedná se o **nejpoužívanější** druh komunikace. Typická unicast komunikace je např. **HTTP**, **SMTP**, ale i **Video on Demand** (YouTube).

### Broadcast

Jde o komunikaci od jednoho bodu ke všem ostatním bodům v síti (**one-to-all**). Broadcastove IPv4 adresy mají všechny **bity adresy hosta** (host address) **rovny 1**. Pozor **IPv6 nemá broadcastové adresy**, vše je řešeno přes multicast. Na linkové vrstvě je MAC adresa pro broadcast **ff:ff:ff:ff:ff:ff**. Broadcast se používá např. u **ARP** nebo **DHCP**. Lze zasílat pouze přes **UDP**.

### Multicast

Multicast je druh komunikace od jednoho vysílajícího k více příjemcům, kteří se o to přihlásili (**one-to-many**). Odesíatel ale zasílá pouze **jeden datagram**, o jeho duplikaci a šíření se **starají síťové prvky**. Přihlašování do skupin je řešeno:

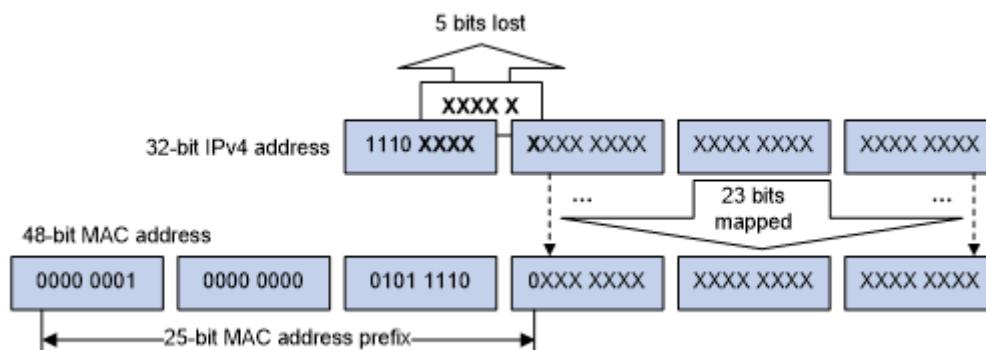
- IPv4 pomocí protokolu **IGMP** a zpráv **Membership Report** (přihlášení se) a **Leave Group** (odhlášení se),
- IPv6 pomocí protokolu **MLD** a zpráv **Multicast Listener Query** (test, jestli někdo ještě naslouchá), **Multicast Listener Report** (přihlášení do skupiny) a **Multicast Listener Done** (odhlášení).

Existují vyhrazené IP adresy (L3) pro multicast:

- IPv4 blok adres **D**, tj. **224.0.0.0** až **239.255.255.255**,
- IPv6 adresy s prefixem **FF00::/8**.

Multicast na L2 je řešen **mapováním IP adres** na **MAC adresy**, mapování ale není přesné:

- IPv4 používá pro mapování 23 bitů MAC adresy, 5 bitů se nemapuje ( $32 - 23 = 9$  (blok D) = 5, tj.  $2^5 = 32$  multicast IP adres je mapováno na stejnou MAC adresu), jedná se o adresy **01:00:5E:00:00:00** až **01:00:5E:7F:FF:FF**.



- IPv6 používá pro mapování 32 bitů MAC adresy a jedná se o adresy **33:33:00:00:00:00** až **33:33:FF:FF:FF:FF** (tedy **33:33:xx:xx:xx:xx**).



Multicast se používá pro **streamování televize a rádia** (IPTV, IP Radio) a zejména u směrovacích protokolů **RIP**, **EIGRP**, **OSPF**, **DVRMP**, ... U IPv6 nahrazuje multicast broadcast, např. při konfiguraci IP adresy pomocí **NDP**.

## Anycast

Na síti existuje **více serverů**, které poskytují **stejnou odpověď** (CDN). Typicky je anycast provoz směrován k nejbližšímu serveru, ale v případě poruchy může být použit jiný (nebo klient si případně může vybrat jaký). Anycast slouží také jako ochrana vůči DoS útokům a umožňuje **rozložení zátěže**.

