Poznámky k štátnej záverečnej skúške na MFF UK

Autor: Vladimír Bedecs (vladimir.bedecs@gmail.com)

Pokryté okruhy

- Složitost a vyčíslitelnost (I2)
- Datové struktury (I2)
- Architektura počítačů a sítí (Systémové architektury)
- Programovací jazyky a překladače (Softwarové inženýrství)
- Analýza a návrh softwarových systémů (Softwarové inženýrství)

Prológ

Tento dokument som vytvoril počas mojej prípravy na štátnice, ktoré sa konali dňa 2. 9. 2010. Osobne som sa učil len z týchto poznámok, čo mi na zvládnutie štátnic stačilo. Niektoré pasáže je treba brať s veľkou rezervou a v žadnom prípade netvrdím, že to čo tu nájdete je správne. Okruh analýza a navrh softwarových systémů je spracovaný veľmi skromne.

Použité zdroje

- Statnicove vycucky v0.5 od Majkla ma inšpirovali k vytvoreniu týchto poznámok a tvorili základnú kostru spoločných okruhov. Z pôvodných Majklových poznámok napokon nezostalo veľa.
- http://wiki.matfyz.cz – v sekcii štátnice sú veľmi užitočné poznámky. Bolo by možno dobré tam nejakým spôsobom tento text (alebo aspoň tie pasáže, ktoré sú správne☺) zmergeovať.
- Slajdy a materiály prednášajúcich – texty, obrázky a niekedy aj celé slajdy... Touto cestou sa za to dotyčným prednášajúcim ospravedlňujem.
- Datové struktury (Vidner, Kotal)
- http://statnice.matfyz.info/ - učebné texty k bakalárkam
- Vyčíslitelnosť od Ladislava Strojila
- http://www.wikipedia.org
- a mnoho ďalších materiálov, ktoré som niekde našiel, použil, ale už som si nedokazal spomenúť na ich pôvod…

Licencia

Používajte, upravujte a šírte bez akéhokoľvek obmedzenia. Poteší ma, ak mi napíšete na mail, že vám tento materiál pomohol k úspešnému zvládnutiu finále na matfyze.

Složitost a vyčíslitelnost

Uvod

Definice (casová složitost): Casovou složitostí algoritmu rozumíme závislost jeho casových nároku na velikosti konkrétních vstupních dat. Analogicky se definuje i pametová složitost. Dobu zpracování úlohy o velikosti n znacíme T(n). Casovou složitost casto zkoumáme z nekolik hledisek:

- **v nejhorším prípade** – maximální pocet operací pro nejaká data,
- **v nejlepším prípade** – minimální pocet operací pro nejaká data,
- **v prumerném (ocekávaném) prípade** – prumer pro všechna možná vstupní data (nekdy též strední hodnota náhodné veliciny T(n)).

Definice: Casova *slozitost problemu* = slozitost asymtoticky nejlepsiho mozneho algoritmu resiciho dany problem (tedy ne nejlepsiho znameho algoritmu).

Asymptotická složitost

Funkce **f(n) je asymptoticky menší nebo rovna než g(n)**, znacíme **f(n)** ∈ **O(g(n))**, práve tehdy, když:

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

Funkce **f(n) je asymptoticky vetší nebo rovna než g(n)**, znacíme **f(n)** ∈ **Ω(g(n))**, práve tehdy, když:

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

Funkce **f(n) je asymptoticky stejná jako g(n)**, znacíme **f(n)** ∈ $\boldsymbol{\theta}$**(g(n))**, práve tehdy, když:

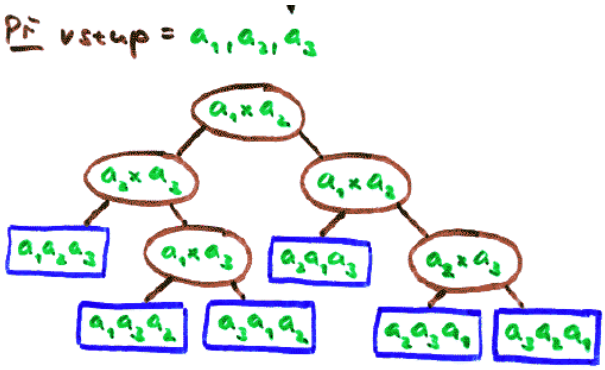
$$\exists c_1, c_2 > 0 \exists n_0 \forall n > n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Dolní odhady pro složitost třídění (rozhodovací stromy)

Třídíme-li s pomocí porovnávání, můžeme si výpočet znázornit binárním stromem. Začínáme v kořeni, v uzlech porovnáváme a větvíme, v listech máme setříděno. Pro každá vstupní data se výpočet někde odliší od ostatních, strom má *n*! listů (tolik je možných uspořádání množiny s *n* prvky.) Výška stromu je počet provedených porovnání při nejdelším výpočtu (v nejhorším případě) je alespoň $\log_2(n!)$. (Úplný binární strom s *k* listy má výšku $\log_2 k$, neúplný je vyšší.)

Veta: Pro kazdy tridici algoritmus zalozeny na porovnavani prvku existuje vstupni posloupnost, pro kterou dany algoritmus provede $\Omega(n \log_2 n)$ porovnani.

Model: Strom porovnani (rozhodovaci strom):



Hloubka stromu = **delka nejdelsi vetve** = **pocet porovnani v nejhorsim pripade**.

Aby bol triediaci algoritmus korektny, musí strom porovnani obsahovat v listoch vsetkych n! permutacii!!!

Veta: Binarny strom s n! listamy ma vysku $\Omega(\mathrm{n} \log_2 \mathrm{n})$.

Dokaz: Uplny binarny strom vysky v ma 2^v listov:

$$2^v \geq n!$$
$$v \geq \log_2 n!$$
$$n! \geq \left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)}$$
$$\log_2 n! \geq \left(\frac{n}{2}\right) \log_2 \left(\frac{n}{2}\right) = \frac{1}{2} n(\log_2 n - 1) \in \Omega(n \log_2 n)$$

Další priklad: Vyber k-teho z n prvkov:

K-tý prvek nelze s porovnanim ziskat driv nez v O(n): musim mit alespon retizek s n-1 hranami.

Rozhodovací stromy formálně

Většina obecných třídicích algoritmů používá jedinou primitivní operaci mezi prvky vstupní posloupnosti, a to jejich vzájemné porovnání. To znamená, že práci takového algoritmu lze popsat binárním stromem, jehož vnitřní vrcholy jsou ohodnoceny porovnáními dvojic prvků vstupní posloupnosti (např. $a_i < a_j$). Bez újmy na obecnosti předpokládejme, že vstupní posloupnost je permutace π množiny $\{1, 2, \ldots, n\}$. Tato permutace prochází stromem takto:

> Začíná v kořeni stromu. Když je ve vnitřním vrcholu v ohodnoceném porovnáním $a_i \leq a_j$, pak když $\pi(i) < \pi(j)$, pokračuje v levém synu vrcholu v, a když $\pi(j) < \pi(i)$, pokračuje v pravém synu vrcholu v. Proces třídění končí, když se dostane do listu.

Aby byl algoritmus korektní, musí platit, že dvě různé permutace skončí v různých listech. Tedy strom popisující korektní algoritmus pro setřídění n-prvkovýh posloupností musí mít alespoň $n!$ listů. Délka cesty z kořene do listu, kde skončila permutace π, reprezentuje počet porovnání, které potřebuje daný algoritmus k setřídění dané posloupnosti π. Protože porovnání vyžaduje alespoň jednotku času, dostáváme tím i dolní odhad na čas potřebný k setřídění této posloupnosti algoritmem odpovídajícím danému stromu. Dolní odhad počtu porovnání i času pro daný algoritmus a všechny n-prvkové posloupnosti je pak délka nejdelší cesty z kořene do listu v odpovídajícím stromu. To nám umožňuje získat obecně platný dolní odhad času potřebného k setřídění $n-$prvkové posloupnosti, kterým je minimum přes všechny binární stromy s alespoň $n!$ listy z jejich maximálních délek cest z kořene do listu. Korektnost těchto úvah plyne z pozorování, že když porovnání je jediná primitivní operace, pak algoritmus není závislý na konkrétních prvcích vstupní posloupnosti, ale jen na jejich vzájemném vztahu. Proto stačí uvažovat pouze permutace n-prvkové množiny, protože zachycují všechny možné vztahy v n-prvkové posloupnosti. Dále je třeba si uvědomit, že vztah mezi stromem pro n-prvkové posloupnosti a stromem pro $(n+1)$-prvkové posloupnosti je dán konkrétním algoritmem a nedá se popsat obecně.

V nevhodném algoritmu se může stát, že v některém listu neskončí žádná permutace. To nastane, když strom pro n-prvkové posloupnosti má více než $n!$ listů, nebo, jinak řečeno, když porovnání dvou stejných prvků se na nějaké cestě vyskytne alespoň dvakrát.

Definice. Mějme třídicí algoritmus **A**, který jako jedinou primitivní operaci s prvky vstupní posloupnosti používá jejich porovnání. Řekneme, že binární strom T, jehož vnitřní vrcholy jsou ohodnoceny porovnáními $a_i \leq a_j$ pro $i, j = 1, 2, \ldots, n$, $i \neq j$, je rozhodovacím stromem algoritmu **A** pro n-prvkové posloupnosti, když pro každou permutaci π n-prvkové množiny platí

> posloupnost porovnání při třídění permutace π algoritmem **A** je stejná jako posloupnost porovnání při průchodu permutace π stromem T.

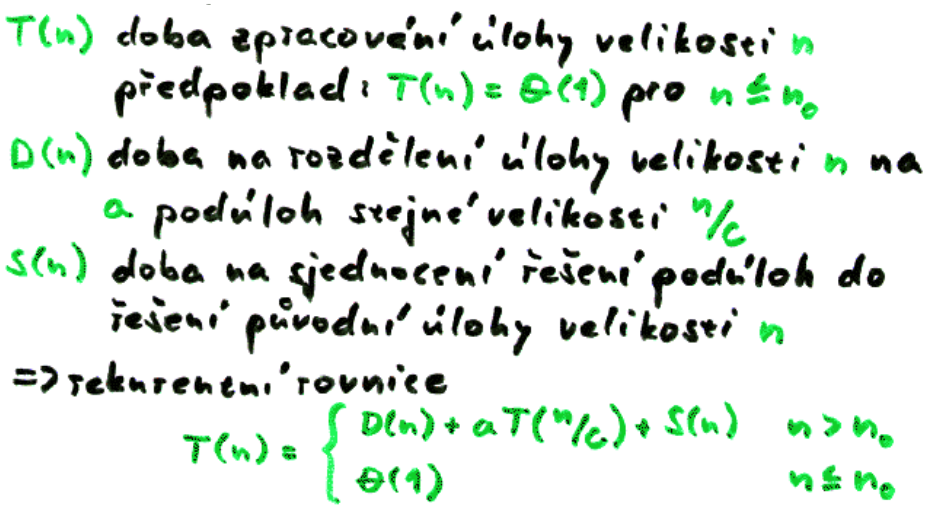
Pak korektnost algoritmu zajišťuje, že dvě různé permutace množiny $\{1, 2, \ldots, n\}$ skončí v různých listech stromu T a dolním odhadem pro čas algoritmu **A** v nejhorším případě je délka nejdelší cesty z kořene do listu. Při rovnoměrném rozdělení vstupních posloupností je očekávaný čas algoritmu **A** roven průměrné délce cesty z kořene do listu.

Metody tvorby algoritmů: Rozděl a panuj.

R&P - metoda návrhu algoritmů o třech krocích:

1. Rozděl úlohu na podúlohy menšího rozsahu a stejného typu.
2. Vyřeš podúlohy buď jako triviální případy nebo dalším dělením.
3. Sjednoť řešení podúloh do řešení původní úlohy.

Analyza casove slozitosti:



Pr.: Mergesort (rozdel na pulky az do delky 1 a pak slevej monotonni posloupnosti):

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Metody řešení rekurentnich rovnic:

1. **Substituční metoda** - uhodneme asymptoticky správné řešení a indukcí jej ověříme (zvlast pro horni a zvlast pro dolni odhad)
2. kuchařka (**Master Theorem**):

Nechť a ≥ 1, c > 1, d ≥ 0 jsou reálná čísla a nechť $T: \mathbb{N} \to \mathbb{N}$ je neklesající funkce taková, že pro $\forall n$ ve tvaru c^k (kde $k \in \mathbb{N}$) platí $T(n) = aT\left(\frac{n}{c}\right) + F(n)$ kde pro funkci $F: \mathbb{N} \to \mathbb{N}$ platí $F(n) = \theta(n^d)$. Označme $x = \log_c a$. Potom:

- je-li x < d, potom platí $T(n) = \theta(n^d)$,
- je-li x = d, potom platí $F(n) = \theta(n^d \log n) = \theta(n^x \log n)$,
- je-li x > d, potom platí $T(n) = \theta(n^x)$.

Příklady:

- **MergeSort:** $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) \to \theta(n \log n)$

Metody tvorby algoritmů: Dynamické programování.

Metoda reseni problemu, ktere v sobe obsahuji prekryvajici se subproblemy. Typickym prikladem je vypocet fibonaciho cisla. Fib. posloupnost je definovana jako:

$f(0) = 1, f(1) = 1$
$f(n+2) = f(n+1) + f(n)$

Vypocet treba 4. cisla by pak byl f(4) = f(3) + f(2) = f(2) + f(1) + f(1) + f(0) = f(1) + f(0) + f(1) +f(1) +f(0) = 5. Vidime, ze jsme f(2) pocitali dvakrat, f(1) trikrat a f(0) dvakrat. Ve vetsim meritku toto zbytecne pocitani vede k exponencialni slozitosti algoritmu. Lepsi cestou je pocitat "od spodu", kdy pomoci f(0) a f(1) spocteme f(2), pak s jeho pomoci f(3) nakonec f(4) s linearni slozitosti.

V dynamickom programovani sa například vytvori mapa fibonanciho cisel a ich hodnot. Před tym, nez zacnem pocitat hodnotu nějakého fibonanciho cisla, pozriem sa najskor do mapy.

V dynamickem programovani vyuzivame:

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

Obvykle se pouziva jeden ze dvou pristupu k dyn. Programovani:

- **Top-down** - problem se rekurzivne deli na podproblemy a po jejich vyreseni se zapamatuji vysledky, ktere se pouziji pri pripadnem opetovnem reseni daneho podproblemu.
- **Bottom-up** - nejdrive se spocitaji vsechny mozne podproblemy (viz pr. s fib. posloupnosti), ktere se potom skladaji do reseni vetsich problemu. Tento pristup je vyhodnejsi z hlediska poctu volani funkci a mista na zasobniku, ale ne vzdy musi byt zrejme, ktere vsechny subproblemy je treba predem spocitat

Příklady použití: Násobení matic, Problém batohu…

Dalsim prikladem je treba **problem batohu** (mame veci ruzne vahy (jednorozmerna varianta pr. batohu) a batoh urcite nosnosti. Ktere veci mame dat do batohu, abychom jej co nejlepe zaplnili?) a jeho specialni pripad - soucet podmnoziny.

Další příklad: **uzávorkování součinu matic** tak, aby počet skalárních součinů byl co nejmenší (dělá se pomocí 2 čtvercových matic (M a K) řádu rovného počtu násobených matic, v pracovních maticích se pracuje pouze nad diagonálou hodnot M_{ij} je možné interpretovat jako minimální počet skalárních součinů při nejlepším uzávorkování matic i až j, hodnota K_{ij} určuje matici, která rozděluje závorkování na dvě podmnožiny matic, závorkování (1. část) x (2.část).

Metody tvorby algoritmů: Hladový algoritmus.

Hladový algoritmus většinou splňuje následující podmínky:

- v každém kroku udělá LOKÁLNĚ optimální rozhodnutí (výběr hodnoty)
- nikdy nemění již udělaná rozhodnutí („nebacktrackuje“)
- pokud vždy zaručeně nalezne GLOBÁLNĚ optimální řešení tak jde o **hladový optimalizační algoritmus**, jinak (většinou) hovoříme o **hladové heuristice**

Problém 1: Je dán souvislý neorientovaný graf G=(V,E) a funkce $d : E \rightarrow R^+$ udávající délky hran. Najděte minimální kostru grafu G, tj. acyklický podgraf grafu G, který má |V|-1 hran, a mezi všemi takovými podgrafy má minimálním součet délek hran.

Problém 2: Je dána množina S={1,2, … ,n} úkolů jednotkové délky. Ke každému úkolu i je dána požadovaná lhůta dokončení $d_i \in N^+$ a pokuta $w_i \in N^+$, kterou je úkol penalizován není-li hotov do své lhůty. Najděte rozvrh (permutaci) úkolů (od času 0 do času n), který minimalizuje celkovou pokutu za úkoly, které nejsou hotovy do jejich lhůty.

Zadefinujeme obecný hladový algoritmus řešící problémy 1 a 2 jako speciální případy.

Definice: Matroid je uspořádaná dvojice M=(S,I) splňující následující tři podmínky :

1. S je konečná a neprázdná množina prvků matroidu M.
2. I je neprázdná množina podmnožin množiny S (nezávislých podmnožin množiny S), která má následující **dědičnou vlastnost**: pokud ($B \in I$) a ($A \subseteq B$) tak ($A \in I$).
3. I má následující **výměnnou vlastnost**: pokud ($A \in I$) a ($B \in I$) takové, že ($|A| < |B|$) tak platí ($\exists x \in B \setminus A : A \cup \{x\} \in I$).

Příklad 1: Maticový matroid.

- T je reálná matice řádu n x n. Pak T definuje matroid $M_T=(S_T,I_T)$ kde
- S_T je množina sloupců matice T a
- $A \subseteq S_T$ je nezávislá (tj. $A \in I_T$) pokud A sestává z lineárně nezávislých vektorů.

Příklad 2: Grafový matroid.

- G=(V,E) je neorientovaný graf. Pak G definuje matroid $M_G=(S_G,I_G)$ kde
- S_G=E a
- $A \subseteq S_G$ je nezávislá (tj. $A \in I_G$) pokud A neobsahuje cyklus (tj. pokud A tvoří les).

Věta: Všechny maximální nezávislé množiny v matroidu mají stejnou velikost.
Dukaz: plyne z vymenne vlastnosti.

Definice: Matroid M=(S,I) je **vážený** pokud je dána váhová funkce $w : S \rightarrow R^+$ a její přirozené rozšíření na podmnožiny S je dáno předpisem $\forall A \subseteq S: \ w(A) = \sum_{x \in A} w(x)$.

Matroidový Problém (MP): Pro daný vážený matroid M najděte nezávislou množinu v M s co největší váhou (taková množina se nazývá **optimální množina matroidu M**). Protože jsou všechny váhy kladné, tak je optimální množina vždy (v inkluzi) maximální nezávislou množinou.

Fakt: Problém 1 je speciální případ MP. Vezměme $M_G=(S_G,I_G)$ a pro každé $e \in S_G$ definujme $w(e) = c - d(e)$, kde c je dostatečně velké číslo (libovolné $c > \max d(e)$ vyhovuje). Nyní platí: optimální množina v M_G (vzhledem k w) = minimální kostra v G (vzhledem k d).

Hladový algoritmus pro MP
Nechť je dán matroid $M=(S,I)$, kde $S=\{x_1, \dots, x_n\}$, a váhová funkce $w : S \rightarrow R^+$.

GREEDY(M,w):
$A := \emptyset$;
Setřiď a přečísluj S sestupně (do nerostoucí posloupnosti) podle vah $\{w(x_1)=\max w(x_i)\}$;
for i := 1 to n do
    if $((A \cup \{x\}) \in I)$ then $A := A \cup \{x\}$;
return A.

Časová složitost: $\Theta(n \log n)$ na setřídění, n krát test na nezávislost (složitost záleží na konkrétním matroidu). Celkem $\Theta(n \log n + n f(n))$ kde f(n) je složitost testu na nezávislost.

Lemma 1: Nechť $x \in S$ takový, že $\{x\} \notin I$. Potom neexistuje $A \in I$ taková, že $x \in A$.
Důsledek 1: Prvky přeskočené před prvním výběrem nemohou být v žádné optimální množině.

Lemma 2: (přípustnost hladového výběru) Nechť je S setříděno podle vah do nerostoucí posloupnosti a nechť je x první prvek v S pro který $\{x\} \in I$. Potom existuje optimální množina $A \in I$ taková, že $x \in A$.
Důsledek 2: První výběr „nezablokuje“ cestu ke konstrukci optimální množiny, což znamená, že v další práci může algoritmus pátrat po optimální množině pouze mezi těmi množinami prvků, které obsahují prvek x.

Lemma 3: (existence optimální podstruktury) Nechť je x první prvek vybraný algoritmem GREEDY(M,w). Pak je nalezení optimální množiny v M obsahující x ekvivalentní s nalezením optimální množiny v matroidu $M'=(S',I')$
; kde $S' = \{y \in S \mid \{x,y\} \in I\}$, $I' = \{B \subseteq S' \setminus \{x\} \mid (B \cup \{x\}) \in I\}$ a váhová funkce w je stejná jako u M (omezená na S'). Matroid M' se nazývá **kontrakce M prvkem x**.

Věta: **GREEDY(M,w) vrací optimální množinu matroidu M.**

Důkaz (pouze zhruba): prvky přeskočené na počátku nemohou být v žádné optimální množině (Lemma 1). Po výběru prvního prvku x Lemma 2 zaručuje existenci optimální množiny obsahující x. Zbývající problém je převeden (díky Lemma 3) na problém stejného typu ale na menší množině prvků → iterujeme dokud existují neprázdné nezávislé množiny.

Amortizovaná složitost

- Typické použití: odhad časové složitosti posloupnosti operací na datové struktuře.
- Počítá amortizovanou cenu (prumerny cas na 1 operaci při provedeni posloupnosti operaci) každé operace v nejhorším případě (v nejhorší možné posloupnosti operací)
- Dává realističtější odhad složitosti než klasický odhad složitosti každé operace nejhorším možným případem
- Nejedná se o složitost v průměrném případě (přes nějakou distribuci vstupních dat).

Amortizovaná analýza

a) **Agregační metoda**
   Spočítáme (nejhorší možný) čas T(n) pro posloupnost n operací.
   Amortizovaná cena 1 operace je pak **T(n)/n**.
b) **Účetní metoda**

Od každé operace „vybereme“ určitý „peněžní obnos“ (amortizovanou cenu operace, která se může lišit podle typu operace), ze kterého „zaplatíme“ za danou operaci. Jedna peněžní jednotka (1 Euro) stačí na zaplacení konstantního množství práce. Pokud po zaplacení za operaci nějaké peníze zbudou, tak jsou uloženy na „účet“. Pokud pro zaplacení za operaci nějaké peníze chybí, tak je lze z „účtu“ čerpat. Zůstatek na účtu musí být stále nezáporný (nic nelze provádět „na dluh“).

Metodu vysvětlíme na dvou jednoduchých “učebnicových” příkladech (v obou je jen jeden typ operace a tím pádem i jediná možná posloupnost operací).

Příklad 1: Inkrementace binárního čítače

Inicializujeme k-bitový binarní čítač hodnotou (0, … ,0) a pak ho n krát inkrementujeme (předpokládáme $k \geq \log_2 n$). **Cíl**: **spočítat bitové operace (bit-flipy) na jednu inkrementaci**.

- Klasicky počítaný nejhorší případ: n incrementací a $\log_2 n$ bit-flipů na jednu incrementataci v nejhorším případě $\rightarrow$ celkem $O(n \log_2 n)$ bit-flipů.
- Ukážeme, že amortizovaná cena jedné inkrementace je $2 = O(1)$ a tím pádem je celkový počet bit-flipů jenom $O(n)$.

Fakt: každá incrementace změní právě jeden bit z 0 na 1 $\rightarrow$ jedno Euro platí za tento flip a druhé Euro je uloženo na účet u takto změněného bitu $\rightarrow$ každý bit jehož hodnota je 1 má na svém účtu jedno Euro, které je později případně použito na flip z 1 na 0

Příklad 2: Vkládání do dynamického pole

Začneme s prázdným polem délky 1 a postupně do něj vložímen n prvků. Pokud je stávající pole plné, tak alokujeme pole dvojnásobné délky a stávající pole (a vkládaný prvek) do něj zkopírujeme. **Cíl**: **spočítat operace s prvky pole na jedno vložení**.

- Klasicky počítaný nejhorší případ: n vložení a O(n) operací s prvky pole na jedno vložení v nejhorším případě → celkem $O(n^2)$ operací s prvky pole
- Ukážeme, že amortizovaná cena jednoho vložení je 3 = O(1) a tím pádem je celkový počet operací s prvky pole jenom O(n)

Amortizovaná složitost formálně

Popíšeme bankovní paradigma pro počítání s amortizovanou složitostí. Předpokládejme, že máme funkci h, která ohodnucuje konfigurace a kvantitativně vystihuje jejich vhodnost pro provedení operace o. Když na konfiguraci D aplikujeme operaci o a dostaneme konfiguraci D', pak amortizovaná složitost $am(o)$ operace o má vystihovat nejen časovou náročnost operace, ale i to, jak se změnila vhodnost konfigurace pro tuto operaci. Proto ji definujme jako $am(o) = t(o) + h(D') - h(D)$, kde $t(o)$ je čas potřebný pro provedení operace o.

Předpokládejme, že chceme provést posloupnost operací $o_1, o_2, \ldots, o_n$ na konfiguraci D_0. Znázorníme si to takto:

$$D_0 \xrightarrow{o_1} D_1 \xrightarrow{o_2} D_2 \xrightarrow{o_3} \ldots \xrightarrow{o_n} D_n.$$

Předpokládejme, že pro každé $i = 1, 2, \ldots, n$ máme odhad $c(o_i)$ amortizované složitosti operace o_i, tj. $am(o_i) \leq c(o_i)$ pro všechna $i = 1, 2, \ldots, n$. Pak

$$\sum_{i=1}^{n} am(o_i) = \sum_{i=1}^{n} \big(t(o_i) + h(D_i) - h(D_{i-1})\big) = h(D_n) - h(D_0) + \sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i).$$

Z toho plyne, že

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n) + h(D_0).$$

Obvykle je $h(D) \geq 0$ pro všechny konfigurace D nebo naopak $h(D) \leq 0$ pro všechny konfigurace D. Když $h(D) \geq 0$, pak

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) + h(D_0),$$

když $h(D) \leq 0$, pak

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n).$$

To znamená, že odhad amortizované složitosti dává také odhad na časovou složitost posloupnosti operací, který bývá lepší než odhad složitosti v nejhorším případě. Tato skutečnost vysvětluje řadu případů, kdy výsledky byly lepší než teoretický výpočet. Ukazuje se, že složitost posloupnosti operací v nejhorším případě je často podstatně menší než součet složitostí v nejhorším případě pro jednotlivé operace.

Agregační metoda

a) binární čítač nechť $\ell = \lfloor \log_2 n \rfloor$

v průběhu n přičtení se

0-ový	bit	překlopí	n	krát
1-ní	-"-		$\lfloor n/2 \rfloor$	-"-
2-hý	-"-		$\lfloor n/4 \rfloor$	-"-
ℓ-ový	-"-		$\lfloor n/2^\ell \rfloor$	-"-

celkový počet překlopení je $\sum_{i=0}^{\ell} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

amortizovaná cena 1 přičtení je $\frac{2n}{n} = 2 = \Theta(1)$

b) dynamické pole

cena i-tého vložení $c_i = \begin{cases} i & \text{pokud } \exists k: i-1 = 2^k \\ 1 & \text{jinak} \end{cases}$

celková cena n vložení:

$$T(n) = \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + 2n = 3n$$

amortizovaná cena 1 vložení je $\frac{3n}{n} = 3 = \Theta(1)$

Účetní metoda

a) binární čítač

Fakt: při každém přičtení je právě jeden bit překlopen z 0 na 1

Od každého přičtení "vyberame" 2 jednotky
- 1 použita na překlopení zmíněného bitu
- 1 dána na účet (u tohoto bitu)

=> každý bit nastavený na 1 ma na účtu 1 jedn.

Překlopení z 1 na 0 hrazena z účtu u daného bitu

Stav účtu = počet 1 v čítači ≥ 0

Amortizovaná cena 1 přičtení je $2 = \Theta(1)$

b) dynamické pole

od každého vložení vyberu 3 jednotky:



- 1 jednotku na vlastní vložení
- 1 jednotku na vlastní přesun při příští expanzi
- 1 jednotku na přesun odpovídajícího prvku v levé části pole při příští expanzi

Platí: Po expanzi je na účtu 0 jinak vždy ≥ 0

Uvod do NP-úplných problémů

Úloha – Pro dané zadání (vstup, instanci úlohy) najít výstup s danými vlastnostmi. (např. najít kostru v grafu, cyklus v grafu, Hamiltonovskou kružnici …)

Optimalizacní úloha – Pro dané zadání najít optimální (vetšinou nejmenší nebo nejvetší) výstup s danými vlastnostmi. výstup s danými vlastnostmi (např. min. kostra, nejkratší cyklus, klika, TSP …)

Rozhodovací problém – Pro dané zadání odpovedet ANO/NE. (např. SAT, SP)

Omezíme se pouze na rozhodovací problémy, což není omezující podmínka, protože ke každé úloze a optimalizační úloze lze „přiřadit“ nejvýše stejně těžký (z hlediska řešitelnosti v polynomiálním čase) rozhodovací problém takový, že:

- polynomiální algoritmus řešící (optimal.) úlohu řeší i přiřazený rozhodovací problém
- pokud je přiřazený rozhodovací problém „těžký“ je „těžká“ i (optimalizační) úloha

Často platí i opačným směrem (ale tento směr nebývá tak snadný).

Definice (rozhodovací problém jako problém rozpoznávání jazyka): Nechť P je rozhodovací problém:

- Předpokládáme, že každé zadání (instance) problému P je zakódována jako řetězec (slovo) v abecedě {0,1}, tj. instance problému = slovo z {0,1}*.
- Kódy všech instancí problému P = jazyk L(P) $\subseteq$ {0,1}* který se dělí na:
  - **$L(P)_Y$ = kódy instancí s odpovědí ANO**
  - **$L(P)_N$ = kódy instancí s odpovědí NE**

Rozhodovací problém P lze nyní formulovat takto: pro daný vstup **$x \in L(P)$ rozhodni zda $x \in L(P)_Y$ nebo $x \in L(P)_N$**.

Předpokládáme, že rozhodnout zda $x \in L(P)$ (tj. rozhodnout zda x kóduje nějakou instanci problému P nebo je to “nesmysl”) je možné v polynomiálním čase vzhledem k |x| (tento test lze považovat za jakýsi „preprocessing“).

Deterministický Turingův Stroj (DTS)



DTS je abstraktní stroj, který se skládá z:

- **řídící jednotky** která může být v konečném počtu stavů
- potenciálně nekonečné (jednostranné) **pracovní pásky** (která sestává z buňek, z nichž každá může obsahovat právě jeden symbol)
- **hlavy** pro čtení a zápis pracující na pásce

Program pro DTS sestává z:

- Konečné množiny Q stavů řídící jednotky, která obsahuje **počáteční stav q_0** and **koncové stavy q_y and q_n**
- Konečné množiny Γ of páskových symbolů, která obsahuje množinu $\Sigma \subseteq \Gamma$ vstupních symbolů a prázdný symbol $* \in \Gamma \setminus \Sigma$
- Přechodové funkce $\delta : (Q \setminus \{q_y,q_n\}) \times \Gamma \rightarrow Q \times \Gamma \times \{ \leftarrow , \bullet , \rightarrow \}$

DTS program začíná v počátečním stavu q_0 s hlavou na prvním buňce pracovní pásky a vstupním slovem $x \in \Sigma^*$ v prvních $|x|$ buňkách pracovní pásky (všechny ostatní buňky obsahují $*$). Každý krok programu spočívá v jednom použití přechodové funkce. Program končí práci v okamžiku, když dosáhne koncového stavu. Program buď **přijme x** ve stavu q_y nebo **odmítne x** ve stavu q_n.

- **Definice**: **Jazyk L(M) přijímaný DTS programem M** sestává ze všech vstupních slov přijímaných strojem M. **DTS program M řeší rozhodovací problém P** pokud M vždy zastaví (pro každý vstup) a navíc platí $L(M) = L(P)_Y$.
- **Definice**: Nechť M je DTS program, který vždy zastaví. **Časová složitost programu M** je dána funkcí $T_M(n) = \max \{ t \mid \exists x \in \Sigma^* , |x| = n$, M skončí na vstupu x po t krocích$\}$. Pokud navíc existuje polynom p takový, že $\forall n : T_M(n) \leq p(n)$, tak se **M nazývá polynomiální DTS program**.

Definice: Rozhodovací **problém P je ve třídě *P*** tehdy a jen tehdy, když existuje polynomiální DTS program M který řeší P, tj. takový, že $L(M) = L(P)_Y$.

Nedeterministický Turingův Stroj (NTS)

Všechny definice stejné jako pro DTS kromě přechodové funkce δ, která je nahrazena tabulkou δ, která přiřazuje každé dvojici z množiny $(Q \setminus \{q_y,q_n\}) \times \Gamma$ množinu možných pokračování, tj. množinu trojic z $Q \times \Gamma \times \{ \leftarrow , \bullet , \rightarrow \}$.

- **Definice**: **NTS s programem M přijímá vstup $x \in \Sigma^*$** tehdy a jen tehdy když existuje přijímající výpočet programu M na vstupu x (výpočet končící v q_y).
- **Definice**: **Jazyk L(M) přijímaný NTS s programem M** sestává ze všech vstupních slov přijímaných strojem M. Čas, ve kterém M přijímá vstup x je definován jako počet kroků v nejkratším přijímacím výpočtu.
- **Definice**: **Časová složitost NTS M** je dána funkcí $T_M(n)$, kde $T_M(n) = 1$ pokud M nepřijímá žádný vstup x s $|x| = n$ a $T_M(n) = \max \{ t \mid \exists x \in \Sigma^* , |x| = n$, M přijímá vstup x v t krocích$\}$ v ostatních případech. Pokud navíc existuje polynom p takový, že $\forall n : T_M(n) \leq p(n)$, tak se **M nazývá polynomiální NTS program**.

(POZOR: na nepřijímaných vstupech nemusí, na rozdíl od DTS, výpočet M vůbec skončit)!

Definice: Rozhodovací problém **P je ve třídě *NP*** tehdy a jen tehdy, když existuje polynomiální NTS program M takový, že $L(M) = L(P)_Y$, a ve třídě ***co-NP*** tehdy a jen tehdy, když existuje polynomiální NTS program M takový, že $L(M) = L(P)_N$.



Alternativní definice NTS: přidáme tzv. **hádací pásku**. Práce NTS pak sestává ze 2 fází:

1) Na hádací pásku je (nedeterministicky) zapsán řetězec $y \in \{1, \ldots, k\}^*$, kde k je maximální počet možných pokračování pro dvojici (stav, symbol) v tabulce δ.

2) Pomocí y je výpočet na vstupu x změněn z nedeterministického na deterministický. Vstup x je přijat pokud existuje (**certifikát**) y kódující přijímající výpočet.

Turingovy stroje, o kterých jsem zatím mluvili jsou tzv. **akceptory** (vstup přijímají nebo nepřijímají). Kromě akceptorů lze definovat také tzv. **transducery**.

Definice: Turingův stroj (deterministický nebo nedeterministický) doplněný o výstupní pásku, na kterou lze pouze zapisovat, a jejíž hlava se pohybuje pouze vpravo, se nazývá **transducer**. Výstupem takového TS je obsah výstupní pásky v okamžiku, kdy TS zastaví. Ostatní pojmy (např. časová složitost) jsou definovány jako u akceptorů.

Úplné problémy pro třídu NP, Cook-Levinova věta.

Definice: Funkce $f : \{0,1\}^* \to \{0,1\}^*$ je **polynomiálně vyčíslitelná** tehdy a jen tehdy, když existuje polynom p a DTS transducer A takový, že pro každý vstup $x \in \{0,1\}^*$ dává DTS transducer A výstup f(x) po vykonání nejvýše p(|x|) kroků.

Definice: Jazyk **L je polynomiálně převoditelný na jazyk L'** (**značíme $L \propto L'$**) tehdy a jen tehdy, když existuje polynomiálně vyčíslitelná funkce f taková, že $x \in \{0,1\}^*$: $(x \in L) \equiv (f(x) \in L')$.

Definice: **Problém P je NP-těžký** tehdy a jen tehdy, když $\forall Q \in$ ***NP*** : $L(Q)_Y \propto L(P)_Y$.
Definice: **Problém P je NP-úplný** tehdy a jen tehdy, když je P NP-těžký a navíc $P \in$ ***NP***.

Definice: **Problém P je co-NP-těžký** tehdy a jen tehdy, když $\forall Q \in$ ***co-NP*** : $L(Q)_N \propto L(P)_N$. **Definice**: **Problém P je co-NP-úplný** tehdy a jen tehdy, když je P co-NP-těžký a navíc
$P \in$ ***co-NP***.

Důsledek: Pokud je Q NP-těžký a $L(Q)_Y \propto L(P)_Y$, pak také P je NP-těžký. Obdobně, pokud je Q co-NP-těžký a $L(Q)_N \propto L(P)_N$, pak také P je co-NP-těžký. Toto je standardní postup jak dokazovat NP-těžkost (resp. co-NP-těžkost), který ovšem vyžaduje existenci alespoň jednoho NP-těžkého (resp. co-NP-těžkého) problému.

Poznámka: Ví se, že **P** ⊆ **NP**. Neví se však, zda $P \neq NP$. Predpokládá se to, ale ještě to nikdo nedokázal.

Poznámka: Pokud existuje polynomiální algoritmus pro nejaký NP-težký problém, pak existují polynomiální algoritmy pro všechny problémy ve tríde NP.

Příklady NP-úplných problémů

Splnitelnost (SAT): KNF F na n Booleovských proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F? Transformace Kachlíkování $\propto$ SAT.

- uvažuji trojice $x_{i,j,k}$, kde i a j určují pozici a k typ kachličky
- vytvoření zápisu v SAT z KACHL pak vypadá tak, že pomocí konjunkcí disjunkcí (a normálních a negovaných tvarů x) reprezentuju tvrzení:
    - na každé pozici je aspoň jedna kachlička
    - na každé pozici je nejvýše jedna kachlička
    - kachličky, které nemohou sousedit vertikálně (horizontálně) spolu nesousedí
    - kachličky, které nemohou být na jednotlivých krajních hranách tam nejsou

3-SAT: Kubická KNF (vždy len 3 premenne v zatvorkach) F na n Booleovských proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F? Transformace SAT $\propto$ 3-SAT.

3-Barvení Grafu (3-BG): Neorientovaný graf G=(V, E). Lze obarvit vrcholy ve V třemi barvami tak, aby žádná hrana v E nebyla monochromatická (jednobarevná)? Transformace
3-SAT $\propto$ 3-BG.

Klika (KL): Neorientovaný graf G=(V, E) a přirozené číslo k. Existuje $V' \subseteq V$, $|V'| = k$, indukující úplný podgraf grafu G? Transformace SAT $\propto$ KL.

Nezávislá Množina (NM): Neorientovaný graf G=(V, E) a přirozené číslo q. Existuje $V' \subseteq V$, $|V'| = q$, taková, že uvnitř V' nejsou žádné hrany? Transformace KL $\propto$ NM.

Vrcholové Pokrytí (VP): Neorientovaný graf G=(V, E) a přirozené číslo r. Existuje $V' \subseteq V$, $|V'| = r$, taková, že každá hrana má ve V' alespoň jeden vrchol? Transformace NM $\propto$ VP.

Hamiltonovská Kružnice (HK): Neorientovaný graf G=(V,E). Obsahuje G Hamiltonovskou kružnici, tj. jednoduchou kružnici, která prochází každým vrcholem právě jednou? Transformace VP $\propto$ HK.

Obchodní Cestující (TSP): Úplný neorientovaný graf G=(V,E), váhy $w : E \rightarrow Z_0^+$, a číslo $k \in Z^+$. Existuje v G Hamiltonovská kružnice s celkovou váhou nejvýše k? Transformace HK $\propto$ TSP.

Součet Podmnožiny (SP): Čísla $a_1, \ldots, a_n, b \in Z^+$. Existuje množina indexů $S \subseteq \{1, \ldots, n\}$, taková, že $\Sigma_{i \in S} a_i = b$? Transformace VP $\propto$ SP.

Algoritmus pro SP: předpokládejme, že platí $a_1 \geq \ldots \geq a_n$, a že A je pole délky b.

for j := 1 to b do {A[j] := 0; a_0 := b+1};

for i := 1 to n do

    A[a_i] := 1;

    for j := b downto a_{i-1} do if (A[j] = 1) and (j+$a_i \leq$ b) then A[j+a_i] := 1;

SP := (A[b] = 1).

Platí: Po i-tém průchodu hlavním cyklem obsahuje pole A jedničky právě u těch indexů, které odpovídají součtům všech neprázdných podmnožin množiny $\{a_1, \ldots, a_i\}$, které jsou nejvýše rovny b. **Časová složitost**: O(nb), což je **exponenciální časová složitost** vzhledem k binárně (ale také ternárně, dekadicky, …) kódovanému vstupu, ale **polynomiální časová složitost** vzhledem k unárně kódovanému vstupu. Algoritmy s těmito vlastnostmi se nazývají **pseudopolynomiální**.

Věta (Cook-Levin 1971): Existuje NP-úplný problém. (ukazujeme pro KACHL)

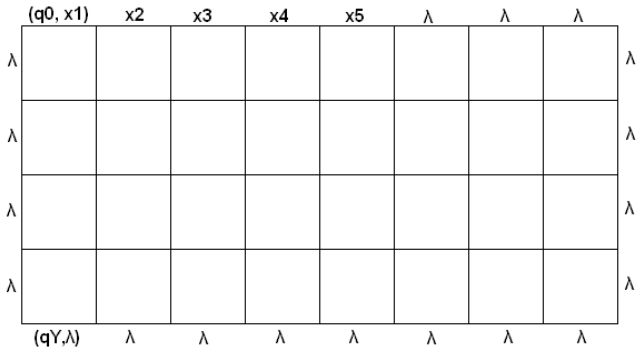
Kachlíkování (KACHL):

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

Otázka: Lze síť S vykachlíkovat pomocí kachlíků z množiny K (stejný typ lze použít libovolně krát, kachlíky ale nelze otáčet) tak, aby a) barvy kachlíků přilehlé k obvodu sítě souhlasily s barvami předepsanými tomto na obvodu sítě a b) každá dvojice barev na dotyku dvou kachlíků byla rovněž shodná?

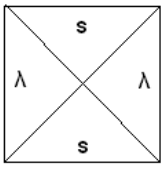
Z definice třídy NP víme, že problém P je v NP tehdy, když existuje polynomiální NTS program M, který řeší P ($L(M)=L(P)_Y$). Abychom dokázali, že KACHL je NP-těžký, musíme jakýkoliv problém P vyřešit pomocí KACHLu, neboli jakýkoliv program M se vstupem $x \in L(M)$ převést na problém KACHL, jehož rešením je také řešení M se vstupem x. Pomocí kachličkování se tedy budeme snažit simulovat běh NTS s programem M a vstupem x. Celý nedeterminismus programu M převedeme na nedeterminismus výběru kachliček. Podaří-li se nám koupelnu vykachličkovat, pak jsme našli i přijímací výpočet programu M se vstupem x.

Stěna, kterou budeme kachličkovat bude dostatečně dlouhá i vysoká (max. počet kroků algoritmu). Na jejím horním okraji bude zleva napsáno slovo x (pro každou hlavičku sloupce kachlíků jeden symbol $x_i \in x$) a zbytek doplníme prázdnými znaky pásky (λ). Přesněji: hlavička prvního sloupce nebude x_1, ale dvojice (q_0, x_1). Takže, na horním okraji stěny je zakódováno vstupní slovo x, které je zapsáno na pásce, a hlava NTS stojí na nejlevějším konci pásky, na symbolu x_1 a je v počátečním stavu q_0.



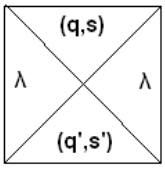
Nyní budeme dlaždičkovat koupelnu shora dolů, řádek po řádku. Každý řádek bude odpovídat jedné konfiguraci turingova stroje. V každém řádku tedy uvidíme aktuální vzhled pásky, dále uvidíme, nad kterým symbolem na pásce je hlava TS a v jakém stavu se nachází. Budeme tedy potřebovat několik druhů kachlíků. Jedna skupina

kachlíků bude reprezentovat pozici na pásce, na které je aktuálně hlava ve stavu q, druhá skupina kachlíků bude reprezentovat pozice pásky, nad kterými hlava není. Je jasné, že když nad danou pozicí pásky není hlava, páska se v tom místě nemění a dvě po sobě jdoucí konfigurace stroje budou tedy mít v tomto místě pásky stejný symbol. Druhá skupina kachlíků tedy bude "kopírovací" - bude kopírovat symbol z předchozí konfigurace (z řádku nad ní) a nebude nic měnit:

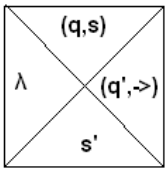


pro všechna s ∈ Σ (abeceda pásky)

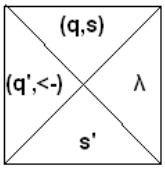
První skupina kachlíků tedy bude reprezentovat pozici pásky se symbolem s, nad kterou je hlava ve stavu q a která přechází přechodovou funkcí δ do stavu q', zapíše na pozici symbol s' a posune se doprava, doleva, nebo zůstane. Vzhledem k tomu, že se jedná o NTS program, není přechodová funkce funkcí, ale zobrazením, kde $\delta(q,s) \subset \{Q \times \Sigma \times \{\leftarrow, \rightarrow, \circ\}\}$. Takže pro dvojici (q,s) může existovat několik dlaždiček:



pro všechna q,s,q',s' taková, že $(q', s', \circ) \in \delta(q,s)$

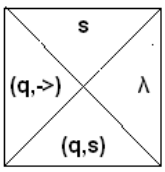


pro všechna q,s,q',s' taková, že $(q', s', \rightarrow) \in \delta(q,s)$

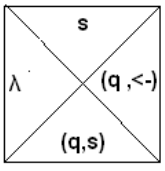


pro všechna q,s,q',s' taková, že $(q', s', \leftarrow) \in \delta(q,s)$

Kromě toho budeme potřebovat ještě další dva "kopírovací" kachlíky, které budou kompatibilní s dvěma předchozími a budou kopírovat symbol s shora a stav q zleva nebo zprava:

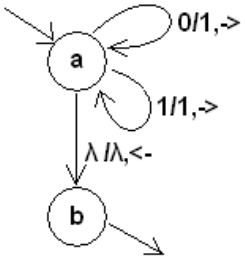


pro všechna q a všechna s

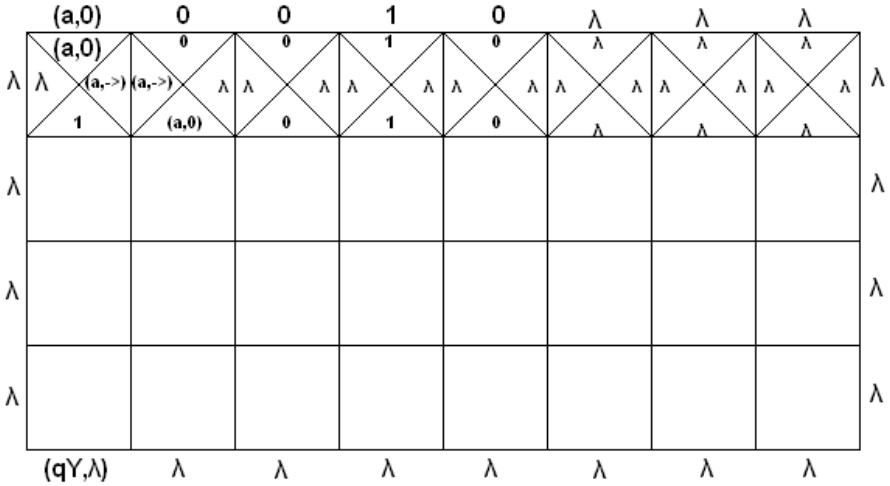


pro všechna q a všechna s

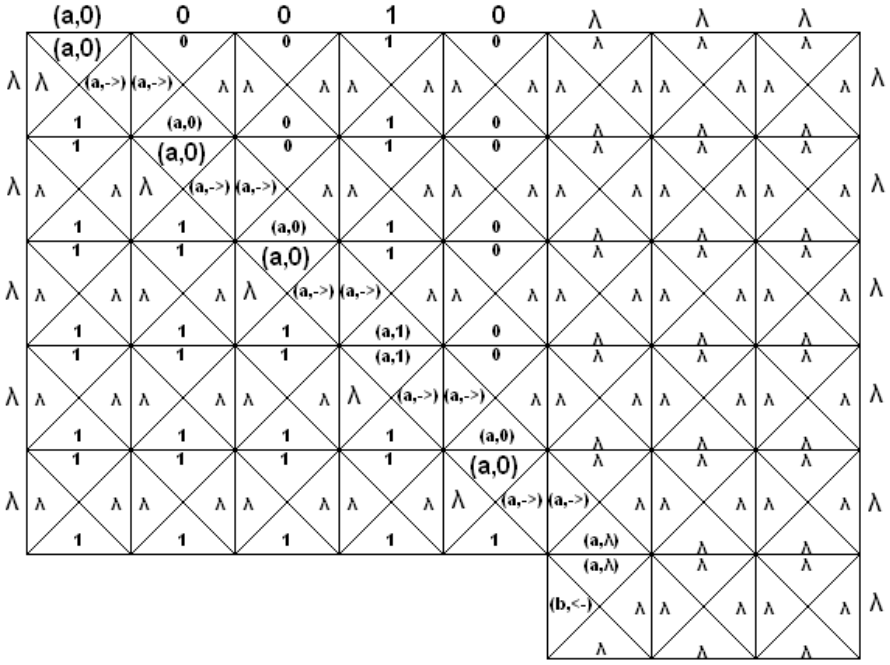
Vše máme připraveno, můžeme kachličkovat. Čistě hypoteticky, pokud by problém zněl "Lze nahradit vstupní slovo x ∈ {0,1}* na pásce stejně dlouhou posloupností jedniček?", vystačili bychom si s jednoduchoučkým DETERMINISTICKÝM programem (zatím neřešíme úklid a koncový stav):



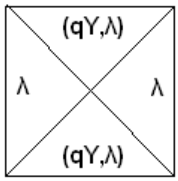
Vzhledem k tomu, že program je deterministický, má pro stejný vstup jen jeden přímočarý výpočet a proto bude i vykachličkování přímočaré, protože se nebudeme muset nikde rozhodovat, kterou z možných cest jít. Na následujícím obrázku vidíme konfiguraci stroje po prvním kroku (spodní okraj prvního řádku kachliček):



Dále budeme pokračovat stejným způsobem:



Tím jsme původní slovo 00010 nahradili slovem 11111. Stroj by po sobě měl uklidit, a skončit v konfiguraci $(q_Y,\lambda),\lambda,\lambda,...$ To ovšem vyžaduje doplnění uvedeného programu o uklízecí část a nový koncový stav q_Y (samozřejmě v návaznosti na změnu δ i nové kachličky). Máme-li koupelnu příliš vysokou, budeme potřebovat ještě poslední druh kachlíků:



Kdybychom měli nedeterministický program, bude i vykachličkování nedeterministické a museli bychom správné vykachličkování nějakým způsobem uhodnout, nebo pomocí "brutal force" zkusit všechny možné kombinace a u každé testovat, zda jsou podmínky barevnosti sousedních stran splněné ($O(k^{2n}n^2)$). Tím jsme ukázali, že pomocí vykachličkování umíme najít přijímací výpočet libovolného NTS programu, a tudíž vyřešit libovolný NP problém, a tudíž

$$\forall P \in NP : L(P)_Y < L(KACHL)_Y$$

Pseudopolynomiální algoritmy, silná NP úplnost.

Nechť je dán rozhodovací problém Q a jeho instance X. Definujme:

- kód(X) = délka zápisu (počet bitů) instance X v binárním (či „vyšším“) kódováním
- max(X) = největší číslo v X (velikost čísla, **NE** délka jeho binárního zápisu)

Definice: Algoritmus řešící Q se nazývá **pseudopolynomiální**, pokud je jeho časová složitost při spuštění na vstupu X omezena polynomem v proměnných kód(X) a max(X).

Poznámka: každý polynomiální algoritmus je samozřejmě také pseudopolynomiální.

Pozorování: Pokud je Q takový, že pro každou jeho instanci X platí $max(X) \leq p(kód(X))$ pro nějaký polynom p, tak pro Q pojem polynomyálního a pseudopolynomiálního algoritmu splývá. Problémy, kde toto **ne**nastává (tj. nesplývají pro ně ty pojmy), budeme nazývat **číselné problémy**.

Definice: Rozhodovací problém Q se nazývá **číselný**, pokud neexistuje polynom p takový, že pro každou instanci X problému Q platí $max(X) \leq p(kód(X))$.

Věta: Nechť Q je NP-úplný problém, který není číselný. Potom pokud ***P*** $\neq$ ***NP***, tak Q nemůže být řešen pseudopolynomiálním algoritmem.

Otázka: Je každý číslený problém řešitelný nějakým pseudopolynomiálním algoritmem?

Odpověď: NE (a typickým představitelům takových problémů se říká silně NP-těžké).

Nechť je Q rozhodovací problém a p polynom. Symbolem Q_p označíme množinu instancí problému Q (tj. podproblém problému Q), pro které platí $max(X) \leq p(kód(X))$, tj. $Q_p = \{X \in Q \mid max(X) \leq p(kód(X))\}$.

Věta: Nechť je A pseudopolynomiální algoritmus řešící Q. Potom pro každý polynom p je A polynomiálním algoritmem řešícím Q_p.

Definice: Rozhodovací problém Q se nazývá **silně NP-úplný**, pokud $Q \in$ ***NP*** a existuje polynom p takový, že podproblém Q_p je NP-úplný.

Věta: Nechť Q je silně NP-úplný problém. Potom pokud ***P*** $\neq$ ***NP***, tak Q nemůže být řešen pseudopolynomiálním algoritmem.

Příklady číselných silně NP-úplných problémů

Obchodní Cestující (TSP) - je to číselný problém (váhy na hranách mohou být libovolně velké). Je silně NP-úplný neboť zůstává NP-úplný i když váhy omezíme (malou) konstantou.

3-partition (3-P) - toto je „čistě“ číselný problém.
Zadání: $a_1, \ldots, a_{3m}, b \in N$, taková že $\forall j : ¼\, b < a_j < ½\, b$ a platí $\Sigma_{j=1}^{3m} a_j = mb$
Otázka: $\exists S_1, \ldots, S_m$ disjunktní rozdělení množiny $\{1, \ldots, 3m\}$ takové, že $\forall i : \Sigma_{j \in Si} a_j = b$?

Aproximační algoritmy a schémata.

Aproximační algoritmy jsou typicky používány na řešení "velkých" instancí NP-těžkých optimalizačních problémů, pro které je nalezení optimálního řešení "beznadějné", tj. časově příliš náročné (pro "malé" instance lze nalézt optimální řešení "hrubou silou" v exponenciálním čase). Aproximační algoritmus má následující tři vlastnosti:

- Vrací (většinou) suboptimální řešení (někdy ale může vrátit i optimum).
- Dává odhad kvality vráceného řešení vzhledem k optimu.
- Běží v polynomiálním čase vzhledem k velikosti zadání.

Značení: **OPT** = optimální řešení
APR = řešení vrácené aproximačním algoritmem
f (Z) = hodnota řešení Z (předpokládáme, že je **vždy nezáporná**)

Definice: Aproximační algoritmus A řeší optimalizační problém X s **poměrovou chybou** r(n), pokud pro všechna zadání problému X velikosti n platí:
$\max \{ f(APR) / f(OPT) , f(OPT) / f(APR) \} \leq r(n)$.

Definice: Aproximační algoritmus A řeší optimalizační problém X s **relativní chybou** e(n), pokud pro všechna zadání problému X velikosti n platí:
$|f(APR) - f(OPT)| / f(OPT) \leq e(n)$.

Příklad maximalizačního problému (optimalizační verze Kliky) - Pro daný neorientovaný graf najít největší (měřeno počtem vrcholů) kliku v daném grafu. Aproximační algoritmus by musel poskytovat odhad (záruku kvality) tohoto typu:

$f(APR) \geq ¾ f(OPT)$, kde f(X) je v tomto případě počet vrcholů (tj. velikost kliky) v řešení X.

Příklad minimalizačního problému (optimalizační verze Obchodního cestujícího) - Pro daný úplný vážený neorientovaný graf najít nejkratší Hamiltonovskou kružnici (měřeno součtem délek hran) v daném grafu. Aproximační algoritmus by musel poskytovat odhad (záruku kvality) tohoto typu:

$f(APR) \leq 2 f(OPT)$, kde f(X) je v tomto případě délka Hamiltonovské kružnice v řešení X.

Příklady aproximačních algoritmů

Úloha vrcholového pokrytí (optimalizační verze) - Neorientovaný graf G = (V,E). Úlohou je najít vrcholové pokrytí minimální velikosti, tj. najít $V' \subseteq V$ takové, že pro každé $(u, v) \in E$ platí $u \in V'$ nebo $v \in V'$ (nebo oboje), a navíc V' má minimální možnou kardinalitu.

Algoritmus A: opakovaně vyber v grafu vrchol nejvyššího stupně, přidej ho do postupně konstruovaného vrcholového pokrytí a odstraň ho z grafu spolu se všemi incidentními (a tedy pokrytými) hranami dokud nezbývá v grafu žádná hrana.

Algoritmus B: opakovaně vyber v grafu libovolnou hranu (u,v) dej jak u tak v do postupně konstruovaného vrcholového pokrytí a odstraň jak u tak v z grafu spolu se všemi incidentními (a tedy pokrytými) hranami dokud nezbývá v grafu žádná hrana. Algoritmus B má konstantní poměrovou chybu r(n) = 2.

Úloha obchodního cestujícího (optimalizační verze) - Úplný vážený neorientovaný graf
G = (V,E) a váhová funkce c : E$\rightarrow Z^+ \cup \{0\}$. Najít v G Hamiltonovskou kružnici nejmenší celkové váhy (délky).

Obchodní cestující s trojúhelníkovou nerovností - Platí:
$\forall$ u,v,w $\in$ V : c(u,w) ≤ c(u,v) + c(v,w)

Napřed nutno zjistit: Je tento podproblém vůbec NP-těžký (a má tedy vůbec cenu uvažovat o aproximačních algoritmech)??

Algoritmus C:

a) Najdi minimální kostru grafu G.
b) Vyber libovolný vrchol grafu G a spusť z něj na nalezené kostře DFS, které očísluje vrcholy v preorder pořadí
c) Výsledná Hamiltonovská kružnice je dána pořadím (permutací) z bodu b)

Pokud je v bodě a) použit Primův (Jarníkův) algoritmus, tak celý algoritmus běží v čase
O(|E|) = O($|V|^2$). Algoritmus C má konstantní poměrovou chybu r(n) ≤ 2.

Obchodní cestující bez trojúhelníkové nerovností

Nechť R ≥ 1 je libovolná konstanta. Potom pokud ***P ≠ NP***, tak neexistuje polynomiální aproximační algoritmus řešící obecný případ obchodního cestujícího s poměrovou chybou nejvýše R.

Důsledek (o existenci neaproximovatelných úloh): Existují NP-těžké optimalizační úlohy, pro které neexistují polynomiální aproximační algoritmy s konstantní poměrovou chybou (pokud ***P ≠ NP***).

Opačný případ: Existují NP-těžké optimalizační úkoly, které lze aproximovat s libovolně malou relativní chybou (poměrovou chybou libovolně blízko 1) s tím, že čím menší je požadovaná chyba, tím vyšší je časová složitost aproximačního algoritmu.

Definice: **Aproximační schéma (AS)** pro optimalizační úlohu X je algoritmus, jehož vstupem je zadání Y úlohy X a (racionální) číslo e>0, který pro libovolné pevné e pracuje jako aproximační algoritmus pro úlohu X s relativní chybou e. (Doba běhu může být exponenciální jak ve velikosti zadání Y tak v 1/e).

Definice: **Polynomiální aproximační schéma (PAS)** pro optimalizační úlohu X je AS, jehož časová složitost je polynomiální vzhledem k velikosti zadání Y úlohy X. (Doba běhu může být stále ještě exponenciální vzhledem k 1/e).

Definice: **Úplně polynomiální aproximační schéma (ÚPAS)** pro optimalizační úlohu X je PAS, jehož časová složitost je polynomiální také vzhledem k k 1/e.

Úloha součtu podmnožiny (optimalizační verze) - Množina přirozených čísel $A = \{x_1, \dots, x_n\}$ a přirozené číslo t. Úlohou je najít množinu indexů $S \subseteq \{1, \dots, n\}$ takovou, že $sum = \sum_{i \in S} x_i$ je co největší při platnosti podmínky $sum \leq t$.

Pseudopolynomiální algoritmus pro SP

Značení: Nechť L je uspořádaný seznam přirozených čísel $a_1, \dots, a_n$. Pak L+x, kde x je přirozené číslo je uspořádaný seznam přirozených čísel $a_1+x, \dots, a_n+x$.

SOUČET(A,t)

begin L_0 := (0); { seznam délky 1 obsahující číslo 0 }

for i := 1 **to** n **do** L_i := MERGE(L_{i-1}, $L_{i-1} + x_i$);

{ MERGE slije oba seznamy, čísla větší než t zahodí }

řešení := největší prvek v L_n

end.

Věta: Seznam L_i pro $1 \leq i \leq n$ je uspořádaný seznam obsahující součty všech podmnožin množiny $A = \{x_1, \dots, x_n\}$, které jsou menší nebo rovny číslu t.

Časová složitost:

- v každém případě $O(|L_1| + \dots + |L_n|)$
- pokud jsou v seznamech drženy duplicitní hodnoty tak (v nejhorším případě) $\Omega(2^n)$
- ale pokud jsou duplicity v MERGE vyházeny, tak $O(n \cdot t)$
- algoritmus je polynomiální pokud $t \leq p(n)$ nebo $\forall i: x_i \leq p(n)$ pro nějaký polynom p

ÚPAS pro SP

Vstup: Množina přiroz. čísel $A = \{x_1, \dots, x_n\}$, přirozené číslo t a aproximační parametr e.

APPROX-SP(A,t,e)

begin L_0 := (0); { seznam délky 1 obsahující číslo 0 }

for i := 1 to n do

begin L_i := MERGE(L_{i-1}, $L_{i-1} + x_i$);

{ MERGE slije oba seznamy, čísla větší než t zahodí }

L_i := PROŘEŽ (L_i, e/n);

end;

řešení := největší prvek v L_n

end.

Časová složitost $\Theta(|L_1| + \dots + |L_n|)$

Uvod do vycislitelnosti

Co to je funkce ve vyčíslitelnosti?

Funkcí se myslí vždy tzv. aritmetická funkce, která k-tici čísel přiřazuje jedno číslo, tedy $f : \mathbb{N}^k \to \mathbb{N}$. Tato funkce navíc nemusí být totální (česky úplná), tedy může být parciální (česky částečná), což znamená, že nemusí být "všude definovaná". V matematice se zápisem $f : A \to B$ obvykle myslí, že definiční obor *f* je *A* (zapisujeme *dom*(*f*) = *A*). Ve vyčíslitelnosti, ale platí $dom(f) \subseteq A$. Ještě jinými slovy: pro některé k-tice nemusí být hodnota *f* definována a tyto k-tice pak nepatří do definičního oboru *f*.

Funkce lze chápat jako programy a naopak

Funkci $f : \mathbb{N}^k \to \mathbb{N}$ si můžeme představit jako program, který na vstupu dostane *k* čísel a vrátí jedno číslo. Tedy schematicky:

program_pro_f(x_1,...,x_k) { ...; return(y); }

Každý konečný matematický objekt (např. celé číslo, slovo nad nějakou abecedou, dvojice čísel,...) lze zakódovat jako jedno (přirozené) číslo (konkrétní metody kódování budou popsány dále). Vezmeme-li si tedy program (který nemá žádné side-efekty, jako že by něco tiskl atd., a jeho návratová hodnota závisí jen na vstupních parametrech), tak ho můžeme chápat jako funkci $\mathbb{N}^k \to \mathbb{N}$.

Co znamená, že program či funkce konverguje či diverguje?

Jistě znáte nějaký program, který se pro nějaký vstup zacyklí a nikdy se nezastaví (while (true) do {}). Pokud se program zacyklí pro všechny vstupy, říkáme, že je to *prázdný program*, kterému odpovídá *prázdná funkce*, což jest funkce *f* taková, že $dom(f) = \emptyset$. Aby to znělo učeněji, tak místo *program se zacyklí* (resp. *funkce není definována*) říkáme *program diverguje* (resp. *funkce diverguje*) a píšeme $f(x_1, ..., x_n) \uparrow$. Obdobně místo *program se zastaví (po konečném počtu kroků)* (resp. *funkce je definována*) říkáme, že *program konverguje* (resp. *funkce konverguje*) a píšeme $f(x_1, ..., x_n) \downarrow$. Například pro funkci *f* jedné proměnné tedy máme $dom(f) = \{x | f(x) \downarrow\}$.

Co znamená $\downarrow =$?

Zápis $f(x) \downarrow = y$ je zkratkou za $f(x) \downarrow \ \wedge \ f(x) = y$, tedy *f*(*x*) konverguje a rovná se *y*. Exaktně vzato by stačilo psát *f*(*x*) = *y*, protože z toho už vyplývá, že funkce konverguje. Zápis $f(x) \downarrow = y$ se ale používá v situacích, kdy nás chce jeho autor upozornit, že to není tak samozřejmé, že *f*(*x*) konverguje. Případně to může znamenat, že se v důkazu nejprve musí dokázat ta konvergence a až poté výsledná hodnota.

Predikát jako funkce

Predikát k proměnných $P(x_1,...,x_k)$ je nějaké tvrzení o těchto proměnných, které může být pravdivé či nepravdivé. Např. $P(x_1,x_2)$="(x_1 je prvočíslo) a ($x_2 > x_1$)". Ve vyčíslitelnosti budou proměnnými vždy čísla a pravdu reprezentujeme jako 1, nepravdu jako 0. Predikáty lze tedy chápat jako funkce jejichž obor hodnot je {0,1} (zapisujeme $rng(f) = \{0,1\}$).

Množina jako predikát

Množinu lze chápat jako soubor prvků, které mají nějakou vlastnost P, tedy množina $M = \{x \mid P(x)$ platí$\}$. Pokud nebude uvedeno jinak, bude v následujícím textu pojem *množina* znamenat vždy *množina (nějakých) přirozených čísel*. Každý predikát jedné proměnné odpovídá nějaké množině a naopak. Jelikož už víme, že predikáty odpovídají funkcím, tak nás nepřekvapí, že množiny odpovídají funkcím jedné proměnné s oborem hodnot {0,1}. Takovouto funkci nazýváme *charakteristická funkce množiny* a pro množinu M píšeme $C_M : \mathbb{N} \to \{0,1\}$ a

$$C_M(x) = \begin{cases} 1 & \text{pro } x \in M \\ 0 & \text{pro } x \notin M \end{cases}$$

Všimněme si, že charakteristická funkce je z definice totální.

Relace jako predikát

Relace na množinách $A_1,...,A_n$ je podmnožina jejich kartézského součinu. Ve vyčíslitelnosti jako množiny A_i bereme (kupodivu opět:-)) $\mathbb{N}$. Relace na n množinách je v důsledku to samé jako predikát n proměnných. Nenechte se tedy zmást, že se občas místo predikátu používá relace. (Zajímavé, že zatímco v teorii množin se odvozuje funkce z relace, zde bereme funkci jako základní pojem a relace převádíme na funkce).

Problém jako množina

V teorii složitosti se *instance problému* definuje jako slovo nad abecedou {0,1} (tedy pro nás číslo) a *problém Q* je pak definován jako úloha rozhodnout, zda daná instance problému patří do jazyka $L(Q)_Y$, tedy do množiny kladných instancí. Zjednodušeně řečeno: problémy jsou zase jen množiny.

Definice: Problém je **rozhodnutelný** (též se říká **efektivně rozhodnutelný** či **rekurzivní**), jestliže existuje TS, který se zastaví na každém vstupu a v přijímacím stavu skončí právě pro ty vstupy, které jsou řešením daného problému. O množině kladných instancí pak říkáme, že je to **rekurzivní množina**, případně že je efektivně rozhodnutelná (o každém prvku můžeme efektivně zjistit, zda do ní patří, nebo nepatří).

Definice: Problém je **částečně rozhodnutelný** (též se říká **vyčíslitelný** či **rekurzivně spočetný**), jestliže existuje TS, který se zastaví (v přijímacím stavu) na vstupech, pro které je odpověď ANO (na vstupech, pro které je odpověď NE, se zastavit nemusí). O množině kladných instancí pak říkáme, že je to **rekurzivně spočetná množina** - existuje pro ni program, který se na vstupu *x* zastaví, pokud

prvek *x* patří do té množiny. Dokud se nezastaví, nevíme nic. *Příklad ze života: když se hraje hokej, prodloužení, je to nerozhodně, vy to nesledujete a jen slyšíte případný řev zvenku, tak pokud se tento řev ozve, víte, že jsme vyhráli, ale pokud je ticho, nevíte nic. Takže hokejový řev je rekurzivně spočetný.*

Definice: Problém je **nerozhodnutelný**, jestliže není rozhodnutelný. Existují nerozhodnutelné problémy, které jsou částečně rozhodnutelné (Halting problem), ale existují také problémy, které nejsou ani částečně rozhodnutelné.

ČRF (částečně rekurzivní funkce)

Lze dokázat (viz dále), že ke každému programu (v libovolném prog. jazyce, či abstraktním modelu jakým je např. Turingův stroj – dále jen TS) jde zkonstruovat ekvivalentní ČRF. Slovem *ekvivalentní* se zde myslí to, že program i funkce dojdou pro libovolný vstup ke stejnému výsledku, délka výpočtu (počet kroků, ať už to znamená cokoli) ovšem může být řádově delší. To je ale ve vyčíslitelnosti běžné: zajímá nás jen co (ne)jde spočítat a jakými prostředky, nikoli, jak dlouho to bude trvat, od toho tu je teorie složitosti. Obecný program se může pro nějaký vstup i zacyklit, tudíž i ČRF nemusí být totální.

Téměř jako synonymum k "částečně rekurzivní"=ČR se používá "rekurzivně spočetné"=RS, každý z těchto dvou přívlastků se ovšem pro zmatení nepřítele (jak jinak než z řad studentstva) tradičně spojuje s jinými podstatnými jmény. O funkci se říká že je to ČRF, kdežto o predikátu, že je to RSP (rekurzivně spočetný predikát), a o množině, že je to RSM (rekurzivně spočetná množina).

Možná už znáte z Automatů a gramatik rekurzivně spočetné jazyky definované jako třídu jazyků přijímaných nějakým TS. Tedy jazyk L je RS, existuje-li TS, jenž se zastaví v koncovém přijímacím stavu právě na všech slovech daného jazyka. Pokud nějaké slovo v daném jazyce není, tak se buď TS zastaví v nekoncovém stavu nebo se zacyklí (tj. nikdy se nezastaví). Chceme-li zjistit, zda nějaké slovo patří do L, pak pustíme TS a čekáme, ale dokud se stroj nezastaví, tak nic nevíme.

TS přijímající L lze jednoduše upravit tak, aby se zastavil právě na slovech z L: pokud by se mohl zastavit v nepřijímacím stavu, tak přidáme instrukce, které výpočet zacyklí. Máme-li ČRF *f*, která je ekvivalentní s takto upraveným TS, tak platí, že $L = dom(f)$. Vysvětlení: každému slovu lze přiřadit jeho kód, tedy číslo; *f* bude funkce jedné proměnné; $dom(f)$ je množina kódů slov jazyka L. Množina $dom(f)$ je RSM.

Z částečně rekurzivních funkcí lze odvodit rekurzivně spočetné množiny a rekurzivně spočetné predikáty.

ORF = obecně rekurzivní funkce

ORFce jsou ty ČRFce, které jsou totální. Z ORFcí lze odvodit ORP=Obecně rekurzivní predikáty a (obecně) rekurzivní množiny – u množin se slovo *obecně* vynechává.

PRF = primitivně rekurzivní funkce

Jak si později ukážeme, PRFce jsou vlastní podmnožinou ORFcí. Jsou to takové hezké funkce, které vždy konvergují. Lze pomocí nich vyjádřit většina známých

operací jako např.: sčítání, násobení, negace, mocnění, test na prvočíselnost… Naopak se těžko hledá úkol, který by nezvládly (tím je např. Ackermannova funkce). Zatím naprosto neformálně můžeme říct, že PRF odpovídají programům, které mohou používat jen for-cykly (to jest for i := 1 to x do..., nikoli takové ty céčkovské, co simulují while-cyklus), ale nikoli while-cykly. Programy odpovídající ČRF mohou navíc používat právě onen while-cyklus (který ale může běžet do nekonečna a tak vznikne divergence).

Přehled zkratek a terminologie

	funkce	**predikát**	**množina**	**problém**
primitivně rekurzivní	PRF	PRP	nezavádí se	nezavádí se
obecně rekurzivní	ORF	ORP	rekurzivní množina	rozhodnutelný
částečně rekurzivní / rekurzivně spočetný	ČRF	RSP	RSM	částečně rozhodnutelný

Efektivní vyčíslitelnost se definuje jako to, co je vyčíslitelné (řešitelné) pomocí ČRF. Vzhledem k již zmíněné ekvivalenci TS a ČRF (kterou ale dokážeme až později), jsou následující výrazy synonyma:

efektivně vyčíslitelné = turingovsky vyčíslitelné = algoritmicky vyčíslitelné = algoritmicky řešitelné atd.

Základné finty

Množiny jsou stejně velké (mají stejnou kardinalitu), pokud mezi nimi existuje bijekce. **Spočetná množina** je množina, kterou lze vzájemně jednoznačně zobrazit na některou podmnožinu množiny přirozených čísel, tedy že existuje bijekce $f : A \to \mathbb{N}$.

Není těžké dokázat, že množina $\mathbb{N} \cup \{a\}$ (přirozená čísla a ještě jeden nový prvek k tomu) je spočetná. Bijekcí na $\mathbb{N}$ je zobrazení $f(a) = 0$ a $f(x) = x + 1 \text{ pro } x \in \mathbb{N}$. Obdobně snadné je najít bijekci mezi $\mathbb{Z}$ a $\mathbb{N}$.

O něco těžší je dokázat, že i množina $\mathbb{N} \times \mathbb{N}$ je spočetná. Jde tedy o to, umět zakódovat dvojici čísel jako jedno číslo, tedy najít funkci $f(a, b) = \langle a, b \rangle$, kde $\langle a, b \rangle$ značí kód dvojice čísel.

Kdyby stačilo jen najít prosté zobrazení do $\mathbb{N}$, tak můžeme použít $f(a, b) = 2^a 3^b$ (místo 2 a 3 mohou být i jiná prvočísla). Pokud požadujeme zobrazení bijektivní (protože z výsledku nejsme schopni říct hodnoty “a“ a „b“), můžeme použít například Cantorovo diagonální uspořádání.

Cantorovo diagonální uspořádání

$$f(a,b) = \langle a,b \rangle = \frac{(a+b)(a+b+1)}{2} + a$$

a\b	0	1	2	3
0	0	1	3	6
1	2	4	7	…
2	5	8	…	…

Kódování n-tic

Právě jsme si ukázali, že pomocí Cantorova diagonálního uspořádání můžeme zakódovat dvojici čísel do jednoho čísla. Můžeme sestrojit i funkci l (jako *left*), resp. r (jako right), která dostane kód dvojice a vrátí její levý (resp. pravý) člen. To lze udělat různě chytře (rychle), ale vyčíslitelníkům stačí, že to jde: např. vyzkoušet všechny dvojice čísel menších než zadaný kód – to jsou dva for-cykly plus krát výpočet kódu (k čemuž opět stačí jen for-cykly). Netřeba se namáhat s něčím sofistikovanějším, rychlejším – ve vyčíslitelnosti to vyjde nastejno.

Když už umíme kódovat dvojice, není těžké vymyslet způsob, jak kódovat n-tice:

$$\langle x_1, ..., x_{n+1} \rangle_{n+1} = \langle x_1, \langle x_2, ..., x_{n+1} \rangle_n \rangle$$

(Index n u právé závorky značí kód n-tice. Závorky bez indexu značí kód dvojice, jak jsme si ho zavedli.)

Můžeme si též zavést funkci pro výběr j-tého z n prvků zakódovaných do jednoho čísla. Na příkladech:

$$l(\langle 42,3 \rangle) = l(1038) = (\langle 42,3 \rangle)_{2,1} = (1038)_{2,1} = 42$$

$$z := \langle 1,2,3,4,5 \rangle_5$$

$$(z)_{5,4} = 4$$

Cantorova diagonální metoda

Jestli se vám začíná zdát, že Cantor a diagonála mají něco společného, tak jste na dobré stopě :-). Diagonální metoda je obecný dokazovací postup, který poprvé využil Georg Cantor k důkazu nespočetnosti množiny reálných čísel. My ji zde použijeme k důkazu nespočetnosti potenční množiny přirozených čísel. V různých obměnách se ovšem bude používat celý semestr pro mnoho dalších důkazů.

$\mathcal{P}(\mathbb{N})$ je nespočetná.

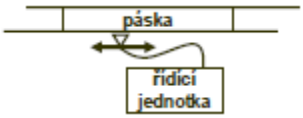
Důkaz sporem: Předpokládejme, že $\{A_n\}_{n \in \mathbb{N}}$ je posloupnost **všech** podmnožin $\mathbb{N}$. Zkonstruujme množinu $B = \{n | n \in \mathbb{N} \wedge n \notin A_n\}$. Platí, že $B \subseteq \mathbb{N}$ (triv.), ale zároveň vidíme, že pro $\forall n \in \mathbb{N} : A_n \neq B$ (index *n* je totiž obsažen v právě

jedné z množin *B* a A_n), a tedy $\{A_n\}_{n\in\mathbb{N}}$ nebyla posloupnost všech podmnožin $\mathbb{N}$, což je spor s predpokladem, že byla.

Turingův stroj

Turingovým strojem nazýváme pětici T = (Q,X,δ,q0,F), kde:

- **Q** - neprázdná konečná množina stavů.
- **X** - neprázdná konečná množina symbolů obsahuje symbol **ε** pro prázdné políčko.
- **δ** - přechodová funkce **δ : (Q - F) × X → Q × X × {-1,0,1}** popisuje změnu stavu, zápis na pásku a posun hlavy.
- **q0** ∈ Q - počáteční stav.
- **F** ⊆ Q - množina koncových stavů.



Výpočet začíná ve stavu q0. V každém taktu dojde ke změně stavu, k přepisu políčka na pásce, k posunu hlavy. Výpočet končí, když není definována žádná instrukce (speciálně platí pro koncové stavy).

Konfigurace Turingova stroje

Je souhrn údajů přesně popisující stav výpočtu. Obsahuje:

- nejmenší souvislou část pásky, která obsahuje všechny neprázdné buňky a čtenou buňku.
- vnitřní stav
- polohu čtené buňky (hlavy)

Modifikace TS

Buďto mohou být omezující (jakoby se "ubírá síla"):

- "neposedný TS": hlava smí jen doleva či doprava, nesmí zůstat na místě
- v každém taktu lze provést jen dvě činnosti
- abeceda se zredukuje na dva symboly (lambda a nelambda - zvýší se počet stavů) …

Nebo mohou modifikace jakoby "přidávat sílu":

- více pásek, více hlav
- nedeterministický TS (na rozdíl od zásobníkových automatů se nedeterministický TS dá simulovat pomocí deterministického)

Univerzální TS

Pro připomenutí (namísto definice): **Univerzální Turingův stroj** U umí simulovat libovolný jiný TS T nad libovolným vstupem V. U dostane na vstupu kód stroje T a vstup V a vrátí stejný výsledek, jako by vrátil T na vsupu V. To můžeme zapsat takto $U(kód(T),V) \simeq T(V)$ (Pokud se T na vstupu V zacyklí, musí se zacyklit i U(kód(T),V)).

Kódování TS

Pokud uvažujeme kanonický TS s jedinou páskou a chceme mu předat na vstup dva parametry, můžeme je oddělit na pásce nějakým speciálním symbolem (na přednášce se k tomuto účelu používá hvězdička, tedy U(kód(T),V) je totéž co U(kód(T) * V)).

Druhý technický detail, který musíme vyřešit, je: jak zakódovat stroj T jako slovo nad nějakou abecedou. Abychom si zjednodušili práci, předpokládáme, že stroj T pracuje nad abecedou s pouze dvěma symboly (λ a 1), a jako správní matematici uchylujeme se k tomuto předpokladu BÚNO (viz předchozí část o variantách TS). Podobně předpokládáme, že koncový stav je jen jeden a přechodová funkce je (až na ten jeden koncový stav) úplná.

Mějme stavy Q_0, Q_1,..., Q_n stroje T, přičemž Q_0 je onen koncový stav. Přechodovou funkci f stroje T pak můžeme zakódovat jako posloupnost: $f(Q_1,\lambda),f(Q_1,1),...,f(Q_n,\lambda),f(Q_n,1)$ (jednotlivé členy této posloupnosti samozřejmě oddělujeme vhodnými symboly). Každý člen posloupnosti je trojice: kód symbolu, který se má zapsat na pásku (λ nebo 1), kód pohybu (doleva, doprava, stát) a kód nového stavu. Kódy stavů můžeme kódovat různě, ale pro popis univerzálního TS je nejjednodušší použít unární kódování: Q_j se zapíše jako j+1 čárek (to +1 je tam proto, aby i kód stavu Q_0 měl aspoň jednu čárku, ale to je detail).

Godelovo číslo a alternativní kódování TS

Očíslujeme stavy z **Q** a symboly z **X** binárními čísly. Popíšeme přechodovou funkci pomocí abecedy A = {0, 1, δ, (,), =, R, N, L, “,”} zřetězením zápisů typu:
δ(q, x) = (q’, y, Z) ;kde Z je jeden z možných posunů {R, N, L}.

„Přeložíme“ každý znak z abecedy A na znak z binární abecedy {0, 1}. Potom každý Turingův stroj jednoznačně odpovídá přirozenému číslu, tzv. **Godelovmu číslu daného turingova stroje.**

Páska univerzálního stroje

Páska UTS pak vypadá následovně (může obsahovat i jiné symboly, než vstupní λ a 1 ze simulovaného stroje, takže má navíc $Y, \Delta, M, \times$ a možná další):

$$Y[blok1]Y[blok2]\Delta[blok3] \times O_1 \times O_2 \times O_3 \ldots O_m \times$$

Kde:

- "blok1" je konfigurace *(trojice <stav, obsah pásky vlevo, obsah pásky vpravo>)* původního stroje, jen obsah právě čteného pole (nad nímž má simulovaný stroj hlavu) je nahrazen pomocným symbolem M
- "blok2" kóduje aktuální číslo stavu stroje (takže to je příslušný počet čárek)
- "blok3" je jen jedna buňka, v níž je uložen obsah právě čteného pole (λ nebo 1)
- O_i jsou kódy přechodových funkcí z jednotlivých stavů (pro oba dva znaky na vstupu), tj. $O_i \equiv f(Q_i, \lambda)f(Q_i, 1)$

Jak funguje simulace T

Práce UTS pak sestává z:

1. odsimulování jednoho kroku práce původního stroje
2. test na to, zda blok2 obsahuje koncový stav
3. procedura vyčištění pásky a vydání výsledku

Takže se vždycky odkrokuje jeden stav původního TS a otestuje se, zda se v blok2 neobjevil koncový stav, a pokud ano, spustí se čištění a konec, jinak se pokračuje simulací dalšího kroku.

Krok původního TS se simuluje následovně:

1. najde se relevantní blok O_i -- stav *i* si nelze pamatovat přímo, proto musím z blok2 postupně umazávat čárky a posouvat nějaký speciální symbol "zarážku" doprava
2. zarážka se posune na konkrétní instrukci podle blok3 (čteného znaku)
3. provede se instrukce (po kouskách se nový stav přenese do blok2, mám 6 možností zápisu písmene a pohybu hlavy; při mazání a pohybu se musí dávat pozor na okraje pásky původního stroje -- pokud někde něco chybí či přebývá, celá páska původního stroje se po pásce UTS musí posouvat).

Algoritmicky vyčíslitelné funkce a jejich vlastnosti.

Efektivně vyčíslitelné = turingovsky vyčíslitelné = algoritmicky vyčíslitelné = algoritmicky řešitelné atd.

Definice: Funkce je (turingovsky) vyčíslitelná, pokud existuje Turingův stroj, který ji (při zvoleném kódování na přirozených čísel na pásce) vyčísluje.

Funkce budeme konstruovat induktivně: zavedeme si tři základní funkce (nulová, následník, vydělení) a pak tři operátory (substituce, primitivní rekurze, minimalizace). Operátor je taková "metafunkce" — na vstupu dostane jednu či více funkcí a na výstupu vrátí nějakou novou funkci.

Ke každé funkci, kterou takto zkonstruujeme, pak bude existovat její **odvození**, což je posloupnost funkcí, z nichž každá je buď základní funkcí nebo vznikla z předchozích funkcí pomocí nějakého operátoru.

Pokud vám to připomíná induktivní výstavbu důkazů v logice, tak si můžete připsat malé nevýznamné plus. Ano, základní funkce jsou paralelou k axiomům, operátory jsou paralelou k odvozovacím pravidlům (modus ponens,...) a odvození funkce je paralelou k odvození důkazu.
Pokud vám to začne připomínat funkcionální programování (nebo dokonce lambda kalkulus), tak si můžete připsat o něco větší (leč stále nevýznamné) plus. Ano, principem je funkcionální programování (ale bez určené implementace).

Pro šťouraly: Ve vyčíslitelnosti se často volně zaměňují termíny *základní funkce* a *schéma základní funkce* (podobně jako v logice se zaměňuje axiom a schéma axiomu). Nulová funkce i následník jsou základní funkce. Ale vydělení je schéma, které definuje nekonečně mnoho základních funkcí.

Základní funkce

- **Nulová funkce**

$$\mathbf{o}(x) = 0 \quad (\forall x)$$

```
function nulova(x) { return 0; }
```

- **Následník**

$$\mathbf{s}(x) = x + 1 \quad (\forall x)$$

```
function succ(x) { return x+1; }
```

- **Vydělení j-té složky**

$$\mathbf{I_n^j}(x_1, ..., x_n) = x_j \quad (\forall j, n : 1 \leq j \leq n)$$

```
function selekce<j>(x1,...,xn) { return xj; }
```

Operátory

Všechny tři operátory mají v dolním indexu n, což značí, že výsledná funkce $\mathbf{h}$ bude funkcí n proměnných.

- **Substituce**

$S_n^m(\mathbf{f}, \mathbf{g_1}, ..., \mathbf{g_m}) = \mathbf{h}$, kde
$\mathbf{h}(x_1, ..., x_n) \simeq \mathbf{f}(\mathbf{g_1}(x_1, ..., x_n), ..., \mathbf{g_m}(x_1, ..., x_n))$

Z uvedené definice je zřejmé, že $\mathbf{f}$ je funkce m proměnných a všechny $\mathbf{g_i}$ jsou funkce n proměnných. Pokud zavedeme konvenci, že $\vec{x} = (x_1, ..., x_n)$, tak můžeme zjednodušeně psát $\mathbf{h}(\vec{x}) \simeq \mathbf{f}(\mathbf{g_1}(\vec{x}), ..., \mathbf{g_m}(\vec{x}))$.

Operátor substituce je základní prvek programování (v lib. prog. jazyce, co znám) — jednoduše úlohu, kterou mám řešit, vyřeším pomocí funkcí, které už naprogramoval někdo dřív.

- **Primitivní rekurze**

$R_n(\mathbf{f}, \mathbf{g}) = \mathbf{h}$, kde
$\mathbf{h}(0, x_2, ..., x_n) \simeq \mathbf{f}(x_2, ..., x_n)$
$\mathbf{h}(i+1, x_2, ..., x_n) \simeq \mathbf{g}(i, \mathbf{h}(i, x_2, ..., x_n), x_2, ..., x_n)$

Je vidět, že operátor primitivní rekurze má sílu for-cyklu. Proměnná x_1 má zvláštní význam — slouží jako čítač. Na přednášce se uváděl vysvětlující příklad na primitivní rekurzi pro n=1. Pak h(0)=konstanta a h(i+1)=g(i,h(i)). Malý problém je v tom, že jsme si nedefinovali nulární funkce a f by v tomto případě musela mít n-1=0 proměnných.

- **Minimalizace**

$M_n(\mathbf{f}) = \mathbf{h}$, kde
$\mathbf{h}(x_1, ..., x_n) \downarrow = z \iff (\mathbf{f}(x_1, ..., x_n, z) \downarrow = 0 \ \ \& \ \ \forall j < z : \mathbf{f}(x_1, ..., x_n, j) \downarrow \neq 0)$

Je vidět, že operátor minimalizace má sílu while-cyklu. Poslední proměnná ve funkci f (v pseudokódu označená jako `i`) má zvláštní význam. Snažíme se ji minimalizovat, to jest najít nejmenší `i` takové, aby f vracela nulu.

Pokud se jako f předá funkce, která nikdy nevrací nulu, tak operátor minimalizace vytvoří funkci, která nebude nikde definovaná (výše uvedený pseudokód se zacyklí). To se u předchozích operátorů stát nemohlo: pokud dostali na vstupu totální funkce, vrátili také totální funkci. Pokud nechceme zavádět nulární funkce, je nutné definici operátoru minimalizace doplnit o podmínku, že f je funkce alespoň dvou proměnných.

Definicia: Odvozeni funkce = posloupnost funkcí f_0 .. f_n = f, kde kazda funkce je bud základní funkce nebo vznikne použitím operátoru na predchozi funkce.

Definicia: Totální funkce = funkce je totální, pokud funkce (program) skončí na každem vstupu.

Definicia: Částečne rekurzivní funkce (ČRF) = funkce je částečně rekurzivní (ČRF), pokud má odvození.

Definicia: Obecně rekurzivní funkce (ORF) = pokud je ČRF a je totální.

Definicia: Primitivně rekurzivní (PRF) = pokud ma odvození pouze s použitím operatorů a R_n (bez minimalizace, tj bez while cyklu). Napr. funkcie na test prvočíselnosti, prvočíselné roklady, zbytky…

Ke každé ČRF (i ORF a PRF) funkci existuje její odvození. Toto odvození lze chápat jako program vyčíslující danou funkci. Například funkce přičtení dvojky má odvození: $S_1^1(\mathbf{s}, \mathbf{s})$. Je dobré si uvědomit, že jedna funkce může mít více programů. To znamená, že ty programy jsou různé (mají jiný zdroják = odvození), ale dělají to samé. Ve skutečnosti má každá ČRF nekonečně mnoho odvození (čili programů).

Například funkci přičtení jedničky můžeme snadno odvodit jako $\mathbf{s}$, ale můžeme si odvození (uměle) prodloužit: $S_1^1(\mathbf{s}, \mathbf{I_1^1})$, nebo dokonce $S_1^1(\mathbf{s}, S_1^1(\mathbf{I_1^1}, \mathbf{I_1^1}))$ atd.

ČRFunkcí je spočetně mnoho (jak si za chvíli ukážeme), a klidně by se místo *třída* ČRFcí mohlo říkat *množina* ČRFcí. Jenže vyčíslitelníci jsou zvyklí, že množinou se vždy myslí nějaká podmnožina přirozených čísel, a tak když chtějí mluvit o množině něčeho jiného, tak to radši i nazvou jinak (třída, systém,...). Obdobně pro ORF a PRF.

Definicia: Predikát P je primitivně rekurzivní, resp. **obecně rekurzivní**, jestliže jeho charakteristická funkce je PRF, resp. ORF.

Definicia: Predikát P je rekurzivně spočetný, jestliže jeho obor pravdivosti je definičním oborem nějaké ČRF.

Definicia: Množina M je rekurzivní, jestliže její charakteristická funkce je ORF.

Definicia: Množina M je rekurzivně spočetná, jestliže M je definičním oborem nějaké ČRF.

Ve vyčíslitelnosti nemáme záporná čísla, a proto si musíme definovat *aritmetické mínus* (značí se $\dot{-}$), které vrátí buď rozdíl, nebo nulu, pokud by rozdíl měl být záporný. PRF funkce **sg** budeme používat ve významu negace.

Věta: Spojky výrokového počtu zachovávají PR, resp. OR. Tzn.

.

Důkaz:

$C_P \,.\, C_Q$

Věta: Omezená kvantifikace zachovává PR, rsp. OR.

Tzn.

Důkaz:

je omezená disjunkce délky x+1:

0 …

je omezená konjunkce délky x+1:

0 …

Věta: P ORP => P i $\overline{P}$ jsou rekurzivně spočetné predikáty.
Tzn. M rekurzivní množina => M i $\overline{M}$ jsou rekurzivně spočetné.

Důkaz:

If P then HALT else NEHALT … P je rekurzivně spočetný

If P then NEHALT else HALT … P je rekurzivně spočetný

Toto tvrzení platí i opačne:

Postova věta

Množina M je rekurzivní právě, když M i $\overline{M}$ jsou rekurzivně spočetné.
Predikát Q je ORP právě, když Q i ¬Q jsou RSP.

Inkluze mezi ČRF, ORF a PRF

Neostré inkluze

Je zřejmé, že každá ORF je ČRF a že i každá PRF je ČRF. Základní funkce jsou totální a operátory substituce a primitivní rekurze totálnost zachovávají. PRF tedy musí být totální. Tím jsme dokázali, že $\mathrm{PRF} \subseteq \mathrm{ORF} \subseteq \mathrm{ČRF}$.

Ostré inkluze

Platí však dokonce: $\mathrm{PRF} \subsetneq \mathrm{ORF} \subsetneq \mathrm{ČRF}$.

ČRF, která není ORF

Chceme najít nějakou netotální ČRF. Zkonstruujeme tedy například funkci f, která není nikde definovaná (tzv. prázdná funkce). K tomu budeme muset zjevně použít operátor minimalizace: f:=M(g), kde g je nějaká funkce, která nikdy nevrací nulu. Nejjednodušší by bylo vzít jako g rovnou následníka, tedy $f := M_0(\mathbf{s})$. Problém je, že jsme si nezavedli nulární funkce (a tou by f byla).

Zvolme tedy $g := S_2^1(\mathbf{s}, \mathbf{I_2^2})$, což znamená, že *g*(*x*,*y*) = *y* + 1 (na první proměnné nezáleží). Pak máme *f*: = M_1(*g*), dohromady tedy $f := M_1(S_2^1(\mathbf{s}, \mathbf{I_2^2}))$.

ORF, která není PRF

Příkladem takové funkce je **Ackermannova funkce** nebo univerzální funkce pro třídu PRF:

```
A(0, n) = n + 1
A(m, 0) = A(m-1, 1)
A(m, n) = A(m-1, A(m, n-1)); m > 0 && n > 0
```

Důkaz: (resp. myšlenka důkazu), že Ackermannova fce je ORF, ale není PRF (TODO nejsem si jistej, že tomu rozumim správně)

1. A(x,y) je ORF
    1. A je totální (je všude definovaná na všech dvojicích ordinálních čísel omega^2)
    2. A je efektivně vyčíslitelná - ?? zastaví se program na vstupních datech?? zobrazíme vstupní data na dobré uspořádání (TODO def) a v průběhu programu klesáme, v dobrém uspořádání neexistuje nekonečný klesající řetězec a tedy program vždy skončí
    3. je A ČRF? (nějaký čachry s minimalizací)

2. A není PRF
    1. definuje se "strukturální" složitost PR programů (nechť PR programy jsou PR termy - tj. základní fce, substituce, primitivní rekurze)
    2. definuje se hloubka PR-termů - maximální počet do sebe zanořených for-cyklů (operátorů rekurze)
        1. $hl(s) = hl(o) = hl(I_n^j) = 0$
        2. $hl(S_n^m(f, g_1, g_2, ..., g_m)) = max(hl(f), hl(g_1), hl(g_2), ...hl(g_m))$
        3. $hl(R_n^n(f, g)) = max(hl(f), hl(g) + 1))$
    3. definuje se třída funkcí R_i obsahující PR-programy hloubky nejvýše *i*
    4. fce f_i je z ($R_i \setminus R_{i-1}$) - je na ní potřeba aspoň *i* vnořených for-cyklů (s méně to nejde)
    5. sjednocení všech R_i je PRF
    6. teď se na $A(x,y)$ nahlídne jako na množinu funkcí $f_x(y)$ a dokáže se, že $f_x(x)$ není z žádného R_i
    7. $A(x,y)$ je rostoucí na obou proměnných
    8. jakákoliv fce z R_i < jakákoliv fce z R_{i+1} (síla *i* vnořených cyklů je menší než síla *i* + 1 vnořených cyklů)
    9. sporem, nechť $A(x,x)$ je z PRF, pak je z nějakého R_i, tedy musí být $A(x,x) < f_{i+1}(x) < f_x(x)$, pro $x > i + 1$ SPOR

Algoritmicky vyčíslitelné funkce a ekvivalence jejich ruznych matematickych definic.

Matematické modely

TS (Turing), ČRF (Kleene), Postovy systémy, Lambda kalkul, RAM Programy (Cook and Reckhow).

Turingův stroj, definice

Turingův stroj je šestice *M* = (*Q*,Γ,*s*,*b*,*F*,δ), kde:

- *Q* je konečná množina vnitřních stavů
- Γ je konečná abeceda symbolů a znaků
- $s \in Q$ je počáteční stav
- $b \in \Gamma$ je symbol reprezentující prázdný symbol (*b* není součástí vstupní abecedy přijímaného řetězce)
- $F \subseteq Q$ je množina koncových stavů
- $\delta : \{Q - q_f \in F\} \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$ přechodová funkce, kde:
    - ***L*** znamená posun čtecí hlavy vlevo, ***R*** znamená posun čtecí hlavy vpravo, ***N*** znamená neposunutí čtecí hlavy

ČRF, definice

Viz výš.

Postovy systémy, definice

Postův systém je trojice (*m*, *A*, *P*), kde

- *m* je kladné celé číslo (*deletion number*).
- *A* je konečná abeceda symbolů, jeden ze symbolů je koncový symbol (*halting symbol*) *halting symbol*. *Slova* jsou konečné posloupnosti symbolů z A.
- *P* je množina přepisovacích pravidel (*production rules*), přiřazujících každému symbolu *x* z *A* slovo *P(x)* (*production*). Produkce (*P(ʜ)*). Zastavovacímu symbolu (kro transparentnost) je přiřazeno slovo následovně: *P(ʜ)* = `'ʜ'`, ikdyž to nehraje roli.

Lambda kalkulu, definice

Lambda výraz se skládá z

- Proměnných v_1, v_2, . . . v_n
- abstrahujícího symbolu λ and .
- závorek ( )

Množina lambda výrazů, Λ, je definována rekursivně:

1. Pokud x je proměnná, pak x ∈ Λ
2. Pokud x je proměnná a M ∈ Λ, pak (λ x . M) ∈ Λ
3. Pokud M, N ∈ Λ, pak (M N) ∈ Λ

RAM programy, definice

RAM stroj se sklálá z registrů, ke kterým máme přímý přístup. RAM program se pak skládá z instrukcí čtyř typů:

1. $LR_i <- R_i + 1$
2. $LR_i <- R_i - 1$
3. LR_i jump (M,N)
4. continue,

kde L je návěští (má tvar "k:", k přirozené, je nepovinné), R_i je registr s adresou i, M,N jsou konkrétní návěstí použitá v programu.

Práce RAM stroje probíhá instrukce v pořadí v jakém jsou zapsané (s výjimkou instrukce 4 - tedy jump). Význam jednotlivých instrukcí je tento:

1. zvýší se hodnota v poli R_i o jedničku
2. sníží se hodnota v poli R_i o jedničku
3. test na hodnotu v poli R_i; pro kladnou hodnotu skoč na instrukci s návěstím M, pro 0 instrukce s návěstím N.
4. continue nic nedělá, pouze oznamuje konec práce.

RAM porgram vyčísluje fci n proměnných takto: Vstupem je n-tice zadaná v registrech R_1,...,R_n. Pokud je fce pro tuto n-tici definovaná, RAM program skončí a v registru R_1 bude funkční hodnota.

Ekvivalence TS a ČRF

ČRF => TS (každá ČRF je turingovsky vyčíslitelná)

Důkaz indukcí podle konstrukce ČRF

- Základní f-ce jsou turingovsky vyčíslitelné.
- Operátory zachovávají T-vyčíslitelnost:
    - Substituce: Vyčíslím podfunkce g_i (pomocí m pásek), dám to dohromady a vyčíslím na tom *f*.
    - Primitivní rekurze: Vyhodnotím *f* a pak y-krát "otočím" *g* za použití vstupních parametrů, počítadla cyklů a výsledku z predcházejícího výpočtu.
    - Minimalizace: Opakovaně vyčísluji *f* na vstupních parametrech a zvětšujícím se počítadle cyklů. Když *f* poprvé vrátí 0 skončím a vrátím hodnotu počítadla.

TS => ČRF

Výpočet TS se dá chápat jako posloupnost konfigurací (obsah pásky, stav řídící jednotky, pozice hlavy). Tyto konfigurace jde zakódovat do kódu, a výpočet TS chápat jako posloupnost těchto kódů. Kódujeme tzv. Postova slova UqsV do $\mathbb{N}$.

Jeden krok TS představuje lokální změnu v Postově slově a jistě tedy existuje funkce na kódech, která charakterizuje. Tato funkce vlastně obsahuje jenom rozhodovací strom odpovídající přechodové funkci našeho TS a na to nám stačí prostředky PRF (na if podmínku použijeme operátor primitivní rekurze).

Pomocí for cyklu (primitivní rekurze) pak můžeme sestavit funkci *Comp*(*x*,*s*), která vypočte kód stavu stroje startujíciho ze stavu odpovídajícímu kódu *x* po *s* krocích. Pak už jen pomocí while cyklu přes to celé nalezneme posloupnost kroků do konečného stavu, tj. $Comp(x, \mu_s(\text{stroj je za s kroku v konfiguraci } q_f)$.
Formálně by to šlo zapsat tak, že pro každý turingův stroj TS existuje ČRF *g* taková, že pro libovolnou konfiguraci *k* platí $kod(TS(k)) \simeq g(kod(k))$.

Částečně rekurzivní funkce

Numerace ČRF

Každá ČRF má z definice nějaké své odvození a každé takové odvození lze převést na přirozené číslo (a zpět, tedy převod je injektivní = prostý). Způsobů, jak tento převod počítat je mnoho, například bychom mohli vzít posloupnost ASCII kódů odvození a chápat to jako jedno číslo, nejčastěji se však používá standardní numerace definovaná v následující větě (Kleene).

- Převodu se říká *numerace* a výslednému číslu *kód funkce* nebo *číslo programu* a vetšinou ho budeme značit jako *e*.
- Jen tak na okraj: Podobně jako ČRF, jdou i logické formule (RSP) převádět na čísla, převodu se říká gödelizace a výsledku gödelova čísla.
- Jedna funkce může mít víc odvození a tedy i víc kódů. Různé funkce mají ale zaručeně různá odvození a tedy i kódy.
- Důsledkem je, že množina ČRF je spočetná.

Kleenova věta

Toto je jedna z nejdůležitějších vět vyčíslitelnosti, protože většina ostatních je na ní založená.

Znění věty (Kleene, o normální formě, o univerzální funkci):

Pro každé $k \geq 1$ existují

- ČRF $\Psi_k(e,x_1,...,x_k)$ k+1 proměnných (univerzální funkce)
- PRP T_k k+2 proměnných (Kleeneho T predikát)
- PRF U jedné proměnné (PRF "obal" univerzální funkce)
- PRF s_k k+1 proměnných (fixování prvních k proměnných)

takové, že:

1. Ψ_k je univerzální funkcí pro třídu k proměnných. Jde efektivně získat *e* z odvození (normalizace) a naopak z *e* jde získat efektivně odvození.
2. $\Psi_k(e, x_1, ..., x_k) \simeq U(\mu_y(T_k(e, x_1, x_2, ..., x_k, y)))$
3. s_k jsou prosté funkce rostoucí ve všech proměnných
4. $\Psi_{m+n}(e, y_1, ..., y_m, x_1, ..., x_n) \simeq \Psi_n(s_m(e, y_1, ..., y_m), x_1, ..., x_n)$ (s-m-n věta)
5. $T_k(e, x_1, ..., x_k, y) \ \wedge \ T_k(e, x_1, ..., x_k, z) \Rightarrow y = z$ (standardnost univerzální funkce).

Náznak důkazu

Cože zase mám něco dokazovat? A on už někdo dokázal existenci univerzálního Turingova stroje, že? A on už někdo dokázal, že TS má stejnou sílu jako ČRF, že? Tak to by pro ty dva už neměl být takový problém dokázat tohle, ne?

Jestli čekáte lepší důkaz než tento monolog, tak hledejte, ale o moc exaktnější to asi nebude a rozhodně to nebude elegantnější. Celý důkaz se nedělá ani na přednášce. Důležitější než přesný důkaz je ale pochopit, co znamenají jednotlivé funkce a predikát *T*.

Co znamená predikát T?

Kleeneho predikát $T_1(e,x,y)$ platí, právě když $y = < y_0,y_1 >$ (dvojice zakódovaná jako jedno číslo) a e je kód funkce jedné proměnné, která se při simulaci na TS nad vstupem x zastaví (přesně) po y_0 krocích s výsledkem y_1. Obecně můžeme zavést predikát $T_k(e,x_1,...,x_k,y)$ pro funkce k proměnných.

- Funkce odpovídající predikátu T_k je primitivně rekurzivní. Přesný důkaz je pracný, ale základní myšlenkou je simulace výpočtu TS, přičemž víme, kolik nejvýše kroků je třeba simulovat, takže nám stačí for-cyklus a nepotřebujeme while.
- Pro konkrétní TS a zadaný vstup x buďto existuje právě jedna dvojice $< y_0,y_1 >$ taková, že po y_0 krocích se stroj zastaví s výsledkem y_1, a nebo neexistuje žádná taková dvojice, pokud se stroj nikdy nezastaví. (Tím jsme dokázali bod 5.)

Co znamená μ_y?

$\mu_y T_1(e,x,y)$ je funkce dvou proměnných (e a x), která vrátí nejmenší y takové, aby platil predikát $T_1(e,x,y)$. Když už máme sestrojen predikát T (včetně odvození), tak tuto funkci vytvoříme jednoduše pomocí operátoru minimalizace.

Co znamená funkce *U*?

U je jednoduchá funkce, která vydělí druhou složku z kódu dvojice. Mimochodem při zavádění kódování dvojic jsme si přesně tuto funkci nazvali r (jako right), ale dnes je v módě U. Speciálně funkce $U(\mu_y T_1(e,x,y))$ vrátí výsledek e-tého programu (e-té ČRFunkce) jedné proměnné spuštěného na vstupu x. Protože to je hodně významná funkce, tzv. *univerzální pro ČRF 1 proměnné*, zavedeme si pro ni značení $\Psi_1(e,x)$, obdobně $\Psi_k(e,x_1,...,x_k)$ pro k proměnných. A aby každý věděl, že jak moc hodně významná ta funkce je, tak pro ni zavedeme ještě další značení, φ_e, ve kterém se pro stručnost explicitně neuvádí k, ale pozná se to z počtu parametrů, protože $\Psi_k(e,x_1,...,x_k) \equiv \varphi_e(x_1,...,x_k)$.

Co znamená univerzální funkce?

Obecně mějme třídu $\mathcal{A}$ funkcí n proměnných, které mají nějakou vlastnost (např. jsou to PRF, ČRF,...). *Univerzální funkce* pro třídu $\mathcal{A}$ je funkce *univ*, která dostane jako vstup číslo e a parametry $x_1,...,x_n$ a vrátí to samé, co by vrátila e-tá funkce z třídy $\mathcal{A}$. Důležité je, že takto dostaneme všechny funkce z té třídy, tedy $\forall f \in \mathcal{A} \quad \exists e \in \mathbb{N} : f(x_1,...,x_n) \simeq univ(e,x_1,...,x_n)$. To mimochodem znamená, že třída funkcí $\mathcal{A}$ musí být spočetná.

Kleenova věta říká důležitou informaci: Existuje **univerzální funkce pro ČRF *k*-proměnných**, kterou budeme nazývat Ψ_k. Tato univerzální funkce je navíc sama také ČRF, z čehož plynou zajímavé důsledky.

Co říká s-m-n věta?

Mám číslo e, tedy e-tý program pro $m + n$ proměnných. Rozhodnu se prvních m proměnných zafixovat na hodnotách $y_1,...,y_m$ a vytvořit tak novou funkci (program) n proměnných. Schematicky zapsáno:

```
nova_funkce(x_1,...,x_n){
  return program_s_cislem_e(y_1, ..., y_m, x_1, ..., x_n);
}
```

S-m-n věta říká, že můžu snadno zjistit číslo této nové funkce (programu). Existuje totiž sada PRFunkcí $s_1,s_2,s_3,...$ a my použijeme konkrétně $s_m(e,y_1,...,y_m)$ a získáme tak číslo té nové funkce.

Univerzální PRF a ORF

Předchozí věta dokazuje existenci univerzální ČRFunkce pro třídu ČRF. Přirozeně nás může napadnout, zda by to nefungovalo i pro PRF a ORF. Inu, nefungovalo.

Neexistuje univerzální PRF pro třídu PRF

Důkaz provedeme klasicky "diagonalizačním" sporem. Nechť existuje univerzální funkce *f* pro třídu PRF jedné proměnné a nechť je *f* sama PRF. Zkonstruujeme funkci *g* tak, aby $g(x) = f(x, x) + 1$. Funkce *g* je PRF, a proto musí existovat index *e* takový, že $f(e,x) = g(x)$ (protože *f* je univerzální funkce pro PRF jedné proměnné). Spor dostaneme, pokud dosadíme za *x* index *e*, tedy $f(e,e) = g(e) = f(e,e) + 1$.

Neexistuje univerzální ORF pro třídu ORF

Důkaz je přesně stejný jako pro PRF.

Několik postřehů na závěr

- Univerzální funkce pro ORF sice existuje, ale je to ČRF, konkrétně to je Ψ_k z Kleenovy věty.
- Univerzální funkce pro PRF sice existuje, ale je to ORF.
- Ψ_k nelze "ztotálnit" (rozšířit do ORF), prostě nelze dodefinovat nějaké hodnoty pro ty vstupy, na kterých diverguje.

Věta o selektoru

Pro každý RSP *Q* k+1 proměnných existuje ČRF φ k proměnných, pro kterou platí

$$\varphi(x_1,...,x_k)\downarrow \Leftrightarrow \exists y Q(x_1,...,x_k,y)$$
$$\varphi(x_1,...,x_k)\downarrow \Rightarrow Q\Big(x_1,...,x_k,\varphi(x_1,...,x_k)\Big).$$

Druhá řádka říká, že pro každý RSP *Q* umíme zkonstruovat takovou ČRF φ, která lze dosadit za poslední proměnnou *Q* (respektive její výsledek y) a pokud φ konverguje, tak *Q* platí. První řádka říká, že φ "se snaží seč může", tedy pokud nějaké y existuje, tak ho najde. (Nebýt této podmínky, šlo by za φ zvolit třeba prázdnou funkci.)

Predikát *Q* k+1 proměnných si můžeme představit jako relaci či graf (grafem se zde rozumí množina bodů, žádné hrany a vrcholy) v k+1 rozměrném prostoru (přirozených čísel ovšem, jak jinak). Predikát $Q(x_1,...,x_k,x_{k+1})$ určuje, zda bod

(x_1,...,x_k,x_{k+1}) je v grafu, či nikoli. φ můžeme chápat jako funkci na k rozměrném prostoru takovou, že v grafu *vybere* v poslední proměnné právě jeden bod (pokud existuje). Celý tento odstavec je tu jen proto, aby vysvětlil název *věta (či lemma) o selektoru* a proč se říká, že ČRF mají RS graf. Klidně na něj zatím zapomeňte.

Důkaz:

Pro RSP *Q* existuje TS *T*, který se zastaví v přijímacím stavu, pokud $Q(x_1,...,x_k,y)$. Definujme ORF

$$\beta(x_1,...,x_k,y,j) = 0 \Leftrightarrow T \text{ přijme vstup } x_1,...,x_k,y \text{ za } j \text{ kroků}$$

Požadovaná ČRF je pak: $\varphi(x_1,...,x_k) = \mu_{<y,j>}\beta(x_1,...,x_k,y,j)$

Pokud totiž pro $x_1,...,x_k$ existuje nějaké *y* takové, že platí $Q(x_1,...,x_k,y)$, tak se odpovídající TS zastaví nejvýše po *j* krocích, a my to zjistíme v $< y,j >$ -té iteraci minimalizační funkce $\mu_{<y,j>}$.

Rekurzivní a rekurzivně spočetné množiny a jejich vlastnosti.

Predikát je **obecně rekurzivní** (ORP), resp. **primitivně rekurzivní** (PRP), jestliže jeho charakteristická funkce je taková (ORF, resp. PRF) . ORP jsou právě ty, které jsou efektivně rozhodnutelné.

Predikát je **rekurzivně spočetný** (RSP), jestliže jeho obor pravdivosti je definičním oborem nějaké ČRF. Tam, kde P neplatí, tam program není definován.

Množina M je **rekurzivní**, jestliže její charakteristická fce je ORF. Rekurzivní množina je taková, že o každém prvku dokážeme efektivně zjistit, zda do množiny patří či nikoliv. Rekurzivní množina je právě efektivně (algoritmicky) rozhodnutelná.

Množina M je **rekurzivně spočetná**, jestliže je definičním oborem nějaké ČRF. Rekurzivně spočetné množiny se nějaký program pouze zastaví, když prvek do množiny patří, jinak nic nevíme.

W_x (x-tá rekurzivně spočetná množina)

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

- Ty *y*, pro která konverguje *x* tá univerzální funkce.

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

- *K* je množina (indexů) funkcí, co konvergují na tom svém indexu.

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

Důkaz: Z definicde je $K = \{x \in W_x\} = \{y : \varphi_y(y) \downarrow\} = \{y : \Psi_1(y,y) \downarrow\}$. Univerzální funkce je ČRF, y je v jejím definičním oboru a tedy je rekursivně spočetná. Kdyby *K* byla rekursivní, tak bychom mohli sestavit ORF α takovou, že $\alpha(x) \downarrow \Leftrightarrow \Psi_1(x,x) \uparrow$. Tato α by měla gödelovo číslo *a* a podle Kleeneho věty o NF by $\alpha(x) \simeq \Psi_1(a,x)$. Speciálně by bylo $\alpha(a) \simeq \Psi_1(a,a)$. A to je spor, protože podle definice je buď $\Psi_1(a,a)$ nezkonverguje, pak ale α(*a*) zkonverguje. Druhá možnost je, že $\Psi_1(a,a)$ zkonverguje ale α(*a*) pak nekonverguje. Kdyby $\overline{K}$ byla rekurzivně spočetná, měla by index x_0 ($\overline{K} = W_{x_0}$), což vede ke sporu $x_0 \in \overline{K} \Leftrightarrow x_0 \in W_{x_0} \Leftrightarrow x_0 \in K$

Věta (Postova): Množina *M* je rekurzivní právě když *M* i $\overline{M}$ jsou rekurzivně spočetné. Predikát *Q* je ORP právě když *Q* i $\neg Q$ jsou RSP.

Důkaz: Zleva doprava triviální – přímo z definice. Zprava doleva intuitivně: $M = dom(P_1), \overline{M} = dom(P_2)$. Pustíme oba programy současně a čekáme, který se zastaví. Formálně pomocí věty o selektoru: $Q(x,y) = (x \in M \wedge y = 1) \vee (x \notin M \wedge y = 0)$. Selektor je ČRF, která vždy konverguje (vždy existuje *y*, je buď jedna, nebo nula). A máme charakteristickou funkci.

Věta:Každá rekurzivně spočetná množina je oborem hodnot nějaké ČRF.

Důkaz: Máme dánu x-tou RSM. Vytvoříme množinu dvojic $R = \{\langle z, y\rangle : z \in W_x \ \& \ y = z\}$ (Predikát $Q(z, y) = z \in W_x \ \& \ y = z$). Množina R je rekurzivně spočetná, tedy má ČRF selektor φ, platí $dom(\varphi) = rng(\varphi) = W_x$. Myšlenka důkazu je, že body, kde φ_x konverguje, vyneseme na diagonálu a vytvoříme selektor. Jeho definiční obor bude současně oborem hodnot.

Věta:Každý obor hodnot ČRF je rekurzivně spočetná množina.

Důkaz: Myšlenka: zkonstruujeme pseudoinverzní funkci k ČRF φ. Dána $M = rng(\varphi_x)$, kde φ_x je ČRF. Definujme predikát $Q(z, y) = \varphi_x(y) \simeq z$. Pak Q je RSP a jeho selektor je ČRF jejíž je M doménou.

Věta: Množina je rekursivní <=> je oborem hodnot nějaké rostoucí úsekové ČRF. (úseková funkce je taková, jejímž def. oborem je počáteční úsek $\mathbb{N}$)

Důkaz:

"=>": U ČRF jsme použili selektor, jehož doménou byla doména charakteristické funkce množiny. Pokud víme, že je tato množina rekursivní, můžeme prvky seřadit do posloupnosti (až do vyčerpání množiny bude tato posloupnost všude definovaná). A dál použijeme minimalizaci: $\varphi(0) \simeq \mu_y(y \in M), \varphi(n+1) \simeq \mu_y(y > \varphi(n) \wedge y \in M)$. Pak $M = rng(\varphi)$.

"<=": Nechť φ je rostouci useková ČRF, $M = rng(\varphi)$. Patří-li y do M, pak $\exists x$ takové, že $\varphi(x) = y$, protože y je z oboru hodnot φ. Navíc víme, že $y \leq x$, to proto, že x, y jsou $\in \mathbb{N}$ a funkce je rostoucí, a v $\mathbb{N}$ "stoupání musí být větší, než diagonála". Při dotazu, jestli $y \in M$ stačí projít všechny $x \leq y$, pak víme, že mezi těmi většími ho už nenajdeme a tak máme definovanou ČRF charakteristickou funkci.

Věta: Množina je r.s. <=> je oborem hodnot prosté úsekové ČRF.

Důkaz:

"<=" Obor jakékoli ČRF je r.s. množina z definice.

"=>" Z předchozí věty víme, že RM jsou generovány rostoucími úsekovými funkcemi. Vezměme množinu $B = \{< x, s >: \varphi(x) \downarrow za\ presne\ s\ kroku\}$, $\varphi(x)$ je ČRF pro kterou $dom(\varphi) = M$. Tato množina je rukursivní, pro každou dvojici lze rozhodnout, jestli $\varphi(x)$ zkonverguje po s krocích. Funkce generující B je přitom rostoucí (z předchozí věty, B je rekursivní) a ze l(f) je v M. Formálně můžeme definovat $g(x) = (f(x))_{2,1}$, pak g je ČRF, $rng(g) = M$ a g je prostá (růst záležel na kódování dvojice <x,s>, f však nemůže vygenerovat x dvakrát, bylo řečeno *přesně s kroků*).

1-převeditelnost, m-převeditelnost

- Množina **A je 1-převedilná na B** (značíme $A \leq_1 B$), jestliže existuje prostá ORF f taková, že $x \in A$ právě tehdy když $f(x) \in B$.
- Množina **A je m-převedilná na B** (značíme $A \leq_m B$), jestliže existuje ORF (ne nutně prostá) f taková, že $x \in A$ právě tehdy když $f(x) \in B$.
- Množina A je **1-úplná**, jestliže je rekurzivně spočetná a každá rekurzivně spočetná množina je na ní 1-převeditelná.
- Množina A je **m-úplná**, jestliže je rekurzivně spočetná a každá rekurzivně spočetná množina je na ní m-převeditelná.

Věta: K je 1-úplná.

Důkaz:

K je rekurzivně spočetná viz definice, dále vezmu libovolnou rek. spočetnou množinu W_x, vezmu ČRF fci α(*y*,*x*,*w*), která ji popisuje (pro každou rek.spoč. množinu existuje ČRF, jejímž definičním oborem je daná množina, proměnná w je fiktivní proměnná – běžný trik). Nyní α(*y*,*x*,*w*) je definována <=> $y \in W_x$

Vyjádřím si alfu pomocí univerzální funkce $\psi_3(a,y,x,w)$, díky s-m-n větě vím, že ta se podmíněně rovná $\psi_1(s_2(a,y,x),w)$, kde s_2 je z Kleeneho věty PRF, tedy i ORF (s_k je prostá fce – dle Kleeneho věty).

Pro libovolné y z W_x platí, že α(*y*,*x*,*w*) je definována a to je <=> $\varphi_h(y,x)(w)$ <=> $\varphi_h(y,x)(h(y,x))$ <=> $h(y,x) \in K$ ($h(y,x)$ vyjadřuje $s_2(a,y,x)$)

Tedy W_x je 1-převeditelná na K pomocí funkce h(y,x).

Věta: $K_0 = \{< y,x > \mid y \in W_x\}$ je 1-úplná (důkaz převodem *K* (je 1-úplná) na K_0).

Další vztahy

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

Definice: Permutace (bijekce) na , která je ORF, se nazývá *rekurzivní permutace*.

Definice: Množiny *A a B jsou rekurzivně isomorfní*, jestliže existuje rekurzivní permutace p taková, že p(A) = B. Značíme A B.

Definice:

$A \equiv_1 B$, jestliže $A \leq_1 B \ \& \ B \leq_1 A$
$A \equiv_m B$, jestliže $A \leq_m B \ \& \ B \leq_m A$

Myhillova věta

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

Důkaz: je ve Strojilovi na straně 5, zprava doleva spočívá v postupném generování grafu *h* tak, aby v kroku *n* platilo $\{0,...,n\} \subseteq dom(h)$ a $\{0,...,n\} \subseteq rng(h)$.

Důsledek: $K \equiv K_0$, neboť $K \equiv_1 K_0$.

Rekurzivně spočetné predikáty

Věta: Je-li *Q* ORF, pak $\exists y Q$ je rekurzivně spočetný predikát. (nema Q v predpokladech byt spis predikat, nez funkce?)

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

Věta: Predikát $\exists y T_k(e, x_1, ..., x_k, y)$ je univerzální RSP pro třídu RSP *k* proměnných.

T_k je predikát z Kleeneho věty o normální formě, co říká že funkce *e* má při y_0 krocích od konce mezivýsledek y_1. ($y = \langle y_0, y_1 \rangle$). Důkaz tvrzení plyne z Kleeneho věty.

Věta: Doplněk nezachovává rekursivní spočenost (důkaz už byl)

Věta: Konjunkce a disjunkce zachovávají rekurzivní spočetnost. (Průnik a sjednocení rekurzivně spočetných množin je rekurzivně spočetná množina. Stejně pro predikáty.)

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

Formální důkaz pro průnik: $x \in A, x \in B$ jsou RSP. Pomocí Kleeneho T Predikátu jde každý predikát zapsat v univerzálním tvaru $\exists z, T_k(a, x_1, .., x_k, z)$, kde *a* je gödelovo číslo predikátu. Tak zapíšeme $x \in A, x \in B$ pomocí obecného T predikátu: $\exists z_1, T_k(a, x, z_1) \wedge \exists z_2, T_k(b, x, z_2)$. Nyní z_1, z_2 zakódujeme do dvojice: $\exists w, T_k(a, x, l(w)) \wedge T_k(b, x, r(w))$. Jelikož T predikár je PRP, můžeme to přepsat jako $\exists z, T_{k+2}(e, a, b, x, z)$. A teď se použije s-m-n věta. $\exists z, T_k(s(e, a, b), x, z)$. Obdobně pro sjednocení.

Věta: Omezená kvantifikace $(\forall y)_{y \leq t}$ a existenční kvantifikace (pro $k \geq 2$) zachovávají rekurzivní spočetnost.

Důkaz: Neformálně: omezený kvantifikátor lze zkontrolovat for cyklem, existenční kvantifikátor je rozveden ve Strojilovi na straně 6.

Algoritmicky nerozhodnutelné problémy (halting problem).

Uvažujme rozhodovací problémy - odpověď je buď *ANO* nebo *NE*. Instance tohoto problému nechť jsou kódována v abecedě Σ = {0,1}. Definujme jazyk kladných a záporných instancí problému. O jazyku řekneme, že je *rekursivní*, pokud ho řeší nějaký (deterministický) turingův stroj. Slovem *řešit problém* budeme popisovat situaci, kdy se stroj pro všechny kladné instance zastaví v koncovém stavu, kde přijímá a pro všechny ostatní instance problému se zastaví ve stavu, kde odmítá.

Definice: Problém je *algoritmicky rozhodnutelný*, pokud je jazyk jeho kladných instancí rekursivní. Jazyk, který není algoritmicky rozhodnutelný je *algoritmicky nerozhodnutelný*.

- **Halting problém, z Johančiných pohádek**

Připomenu definici: neexistuje algoritmus, který by pro daný turináč a daný vstup dokázal rozhodnout, zda se turingáč na tomto vstupu zastaví. Pro důkaz použijeme pár myšlenek, které je dobré si zapamatovat, protože se s nimi nevidíme naposled. První z nich je (překvapivě) myšlenka UTS, druhou je otázka zastavení algoritmu na jeho vlastním kódu a třetí myšlenkou je přiřadit výsledky něčeho úmyslně opačně - nelogicky: když se něco povede, 0, když nepovede, 1. Z tohohle všeho dohromady kouká nějaký slibný spor, tak teď si to konečně aplikujeme na náš konkrétní úkol.

Důkaz (pomocí UTS): provedeme sporem, předpokládáme, že existuje algoritmus H(M,K), co mu předhodíme turingáč M a vstup K a on řekne, jestli tenhle vztah má budoucnost :). Tedy vždy se zastaví a řekne 0 (nemá), nebo 1 (má). Pomocí něj sestrojíme kapku brutálnější algoritmus Alg(K), který se nad daným kódem programu zastaví právě tehdy, když se tento program sám se na svém vlastním kódu nezastaví. Tedy Alg(K) zastaví <=> U(K,K) nezastaví. Prostě algoritmus Alg říká: mně se líbí program, co nesežere svůj vlastní vstup, a tak se na něm zastavím, naopak program, co svůj vlastní vstup sežere, se mi nelíbí, takže se na něm třeba zacyklím a prostě trucuju :). Alg(K) definujeme podle výsledku výše zmíněného algoritmu H (který se vždy zastaví a něco řekne, takže my ho provedem a podle toho, co řekl, se rozhodnem): H(U(K,K),K) = 0 => Alg(K) zastaví, H(U(K,K),K) = 1 => Alg(K) zacyklí. Celá tahle prasárna jde provést právě proto, co jsme si na začátku definovali, tedy že algoritmus H existuje a vždy se zastaví. No a teď už asi nikoho nepřekvapí, že po tak ďábelské definici algoritmu Alg snadno vykouzlíme spor - pustíme ho na jeho vlastním kódu. Potom Alg(Alg) zastaví <=> U(Alg,Alg) nezastaví <=> Alg(Alg) nezastaví. Tohle ukazuje, že původní algoritmus H (který rozhoduje o zastavení) je zvrácený a nemůže existovat :).

Důkaz (pomocí ČRF): Z kleeneho věty víme, že existuje efektivní očíslování pro třídu ČRF a její univerzální funkce. vezměme množinu $K = \{x : \varphi_x(x) \downarrow\} = \{x : \Psi(x,x) \downarrow\}$. Ta není rekursivní (pokud by byla, tak bychom mohli dodefinovat funkci α takovou, že $\alpha \downarrow \Leftrightarrow \varphi_x(x) \uparrow$. Ta by byla r.s. a měla gödelovo číslo e, takže $\varphi_e(e) \simeq \alpha(e)$ takže $\alpha(e) \downarrow \Leftrightarrow \varphi_e(e) \uparrow \Leftrightarrow \alpha(e) \uparrow$). No a tím spíš není rekursivní $K_0 = \{< x,y >: \varphi_x(y) \downarrow\}$, protože $K \leq_1 K_0$.

Ostatní nerozhodnutelné problémy

Věta 3.3
Problém totálnosti vyčislitelných funkcí je nerozhodnutelný. Jinými slovy neexistuje algoritmus, který pro daný index i rozhodne, zda je vyčislitelná funkce $\varphi_i : \mathbf{N} \longrightarrow \mathbf{N}$ totální.

Věta 3.4
Problém verifikace je nerozhodnutelný. Jinými slovy neumím rozhodnout, zda daný program počítá danou funkci.

Věta 3.5
Problém ekvivalence je nerozhodnutelný. Jinými slovy neexistuje program, který pro i a j rozhodne, zda $\varphi_i = \varphi_j$.

Věty o rekurzi a jejich aplikace: priklady, Riceova veta.

Pouziti v logice, godelovy vety, paradox lhare.

Aplikace vět o rekurzi: Důkaz Riceovy věty, nerozhodnutelnost minimality Turingova stroje, ekvivalenci dvou programů nelze rozhodnout algoritimicky (důsledek věty Rice).

Věty o rekurzi

Věta (o rekursi): Jestliže *f* je ČRF jedné proměnné, potom existuje *a* takové, ze $\varphi_{f(a)}(x) \simeq \varphi_a(x)$ pro každé *x*.

Říká, že existuje pevný bod zobrazení *f*. (φ je zřejmě univerzální funkce jedné proměnné)

Důkaz:
$\varphi_{f(s_1(z,z))}(x) \simeq \Psi_2(s_1(z,z),z,x) \simeq \Psi_2(e,z,x) \simeq^{smn} \Psi_1(s_1(e,z),x) \simeq \varphi_{s_1(e,z)}(x)$

- $s_1(z,z)$ je funkce se s-m-n věty, která dělá totéž co z, ale má první parametr fixovaný na hodnotu z
- e je index programu, který počítá výraz vlevo (tj. $s_1(z,z)$, pak $f(s_1(z,z))$, a vzniklou funkci aplikuje na x)

Pak položíme $z = e$ a $a = s_1(e,e)$ a máme ten pevný bod.

Na přednášce bylo rozebíráno, jak vypadají jednotlivé programy pro *a* a pro *f(a)*, hlavně, který počítá rychleji. A to je pěkná otázka ke státnici. Z odvození v důkazu je vidět, že program s pořadovým číslem *a* spustí univerzální funkci na vstup (a, x). Tím univerzální funkce nejprve spočte $s_1(e,e)$, tedy a, následně spočte f(a) a potom simuluje hlavní program.

Výpočet jde tedy v pořadí:
$\Psi(a,x) \simeq \Psi(s_1(e,e)) \simeq \Psi(e,e,x) \simeq \Psi(f(s,(e,e)))e,x)$. Program s pořadovým číslem a tedy počítá déle než ten s f(a).

Věta (o generování pevných bodů): Pro každou částečně rekurzivní funkci *f* existuje prostá rostoucí PRF *g* taková, že platí $\varphi_{f(g(j))}(x) \simeq \varphi_{g(j)}(x)$.

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

Důkaz: $\varphi_{f(s_2(z,z,j))}(x) \simeq \Psi_3(e,z,j,x) \simeq \Psi_1(s_2(e,z,j),x) \simeq \varphi_{s_2(e,z,j)}(x)$

- *e* podobně jako u věty o rekurzi schválně počítá levou stranu

Zvolíme $g(j) = s_2(e,e,j)$.

Věta (o rekursi v závislosti na parametrech): Nechť $f(x,y_1,y_2,...y_m)$ je ČRF m+1 proměnných. Pak existuje PRF $g(y_1,y_2,...y_m)$ m proměnných taková, že
$\varphi_{g(y_1,y_2,...y_m)}(x) \simeq \varphi_{f(g(y_1,y_2,...y_m),y_1,y_2,...y_m)}(x)$

Důkaz: Obdobně jako základní věta o rekursi.

$\varphi_{f(s_{m+1}(z,z,\overrightarrow{y}),\overrightarrow{y})}(x) \simeq \Psi_{m+2}(e,z,\overrightarrow{y},x) \simeq^{smn} \Psi_1(s_{m+1}(e,z,\overrightarrow{y}),x) \simeq \varphi_{s_{m+1}(e,z,\overrightarrow{y})}(x)$

A teď položíme $g(\overrightarrow{y}) = s_{m+1}(e,e,\overrightarrow{y})$.

V předchozích větách jsme hledali pro ČRF zobrazení jeden pevný bod, pak pro vícerozměrné zobrazení parametrizovaný pevný bod. Tahle věta říká, že pro dané zobrazení můžou pevné body generovat prosto funkci.

Věta (o programu co vypíše sám sebe): Nechť *f*(*x*,y_1,y_2,...,y_m) je ČRF. Pak existuje a takové, že $\varphi_a(y_1,y_2,..,y_m) \simeq f(a,y_1,y_2,...,y_m)$.

Důkaz: Jelikož f je ČRF, platí

$f(x,y_1,y_2,...,y_m) \simeq \Psi(e,x,y_1,y_2,...,y_m) \simeq^{s-m-n} \Psi(s_1(e,x),y_1,y_2,...,y_m)$

. Pou6ijeme větu o rekursi. $\varphi_{s(e,x)}(x) \simeq \varphi_s(e,a_2)$, pro nějaký pevný bod a_2. Potom ale a = s_1(e, a_2) je hledaný pevný bod.

Pomocí této věty jde snadno najít program, co vypíše sám sebe. Vezměme funkci $f(x,y) \simeq x$. Pak podle předchozí věty najdeme a takové, že $\varphi_a(y) \simeq f(a,y)$, jenže z definice je $f(a,y) \simeq a$, tedy máme $\varphi_a(y) \simeq a$. To je program, který pro libovolný vstup vypíše svůj kód.

Riceova věta

Věta (Riceova): Jestliže $\mathcal{A}$ je třída ČRF (jedné proměnné), ktera je netriviální (není prázdná, ani neobsahuje všechny ČRF), potom $A_{\mathcal{A}} = \{x : \varphi_x \in \mathcal{A}\}$ není rekurzivní.

Důkaz: Dokazuje se sporem. Nechť $A_{\mathcal{A}}$ je rekurzivní. Potom lze vytvořit ORF *f* takovou, že všechny prvky z *A* zobrazí na nějaký prvek $b \notin A$ a všechny prvky mimo *A* zobrazí na nějaký prvek $a \in A$. Existence *a* a *b* je zaručena požadavkem na neprázdnost a neúplnost. Podle věty o rekurzi existuje pro *f* (je ORF) nějaký pevný bod u_0, že platí $\varphi_{u_0} = \varphi_{f(u_0)}$.

Tedy $u_0 \in A \Rightarrow f(u_0) = b \notin A$ a $u_0 \notin A \Rightarrow f(u_0) = a \in A$. Což je ovšem spor s tím, že čísla u_0 a $f(u_0)$ buď v *A* leží obě nebo jsou obě mimo – jsou to indexy stejné funkce.

Nejedná se o třídu programů, ale funkcí. Tedy i pro jednoprvkovou $\mathcal{A}$ bude $A_{\mathcal{A}}$ nekonečná a nerekurzivní (každá funkce je vyčíslována nekonečně mnoha programy a rozhodnout o jejich ekvivalenci efektivně nelze). Tato věta vlastně říká, že každá netriviální vlastnost ČRF je algoritmicky nerozhodnutelná.

Důsledek (Riceovi věty): ekvivalence programů.

"Ekvivalenci programů nelze rozhodnout algoritmicky."

Mějme nějaký program s indexem e. Zvolíme $\mathcal{A} = \{\varphi_e\}$, čili $B = \{x | \varphi_x = \varphi_e\}$ je množina všech programů (jejich indexů), které jsou ekvivalentní s programem e. Dle Riceovy věty je množina B nerekurzivní.

Datové struktury

Uvod

Dynamická množina je množina prvku (datová struktura), menící se v case. Každý její prvek je prístupný pres ukazatel a obsahuje:

- **klíc** (jednu položku, typicky hodnotu z lin. usporádané množiny)
- **ukazatel(e)** na další prvky
- prípadne další data

Na takové množine jsou definovány tyto operace:

- **find** - nalezení prvku podle klíce
- **insert** - pridání dalšího prvku
- **delete** - odstranení prvku
- **min, max** - nalezení nejvetšího / nejmenšího prvku
- **succ, pred** - nalezení následujícího / predcházejícího prvku k nejakému predem danému

Stromové vyhledávací struktury: binární stromy a jejich vyvažování.

Binární strom je dynamická množina, kde každý prvek (uzel, node) má krome klíce a príp. dalších dat tri ukazatele na levého a pravého syna a rodice. Speciální uzel je koren, který má NULLový ukazatel na rodice. Ten je v binárním strome jeden. Uzly, které mají NULLové ukazatele na pravého i levého syna, se nazývají listy.

Podstrom je cást stromu (vybrané prvky), která je sama stromem - napr. pokud se jako koren urcí jeden z prvku. Levý(pravý) podstrom nejakého prvku je strom, ve kterém je korenem levý(pravý) syn tohoto prvku. Výška stromu je délka nejdelší cesty od korenu k listu.

Binární strom je vyvážený, pokud max. 1 uzel má jednoho syna (tj. všechny vnitrní uzly krome až na jeden mají oba syny, listy z definice nemají žádného). Výška vyváženého stromu roste logaritmicky vzhledem k poctu uzlu. Výška nevyváženého stromu muže rust až lineárne vzhledem k poctu prvku (i „spojový seznamÿ je platný bin. strom).

Binarni vyhledávací stromy

Pro binarni vyhledavaci strom (BVS) plati, ze kazdy prvek v levem podstromu ma mensi a kazdy prvek v pravem podstrmu vetsi hodnotu nez korenovy prvek toho podstromu.

Obecne operace member, insert, delete, succ, predec, min, max. Vsetky operacie vyzaduju cas O(vyska(T)); T = bvs strom.

Member: podle hodnoty hledaneho jdu do leveho nebo praveho podstromu do doby nez najdu nebo je potomek NULL a hodnota neni ve stromu.

Insert: Najdi "vetev", kde by mel byt novy prvek stejne jako member a tam ho pridej. Pokud je to potreba, uprav strom, at splnuje integritni omezeni sveho druhu.

Delete: Najdi prvek. Kdyz nema zadneho syna, tak ho zrus, kdyz jednoho, tak ho zrus a syna prepoj na otce. Kdyz dva, tak najdi nejvetsi mensi prvek (1x doleva, a potom dokud to jde tak doprava a mam nejvetsiho mensiho potomka) a tyhle dva prehod. Na novem miste ma ruseny prvek max. jednoho syna. Pokud je to potreba, uprav strom, at splnuje integritni omezeni sveho druhu.

Min: najdi prvok s minimalnou hodnotou. (nejlevjejsi prvek v strome).

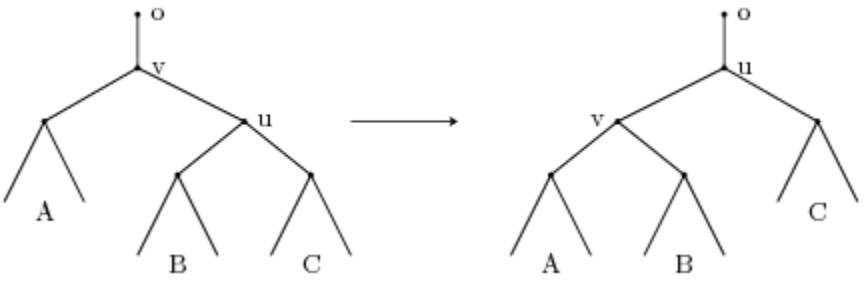
Max: najdi prvok s maximalnou hodnotou. (nejpravjejsi prvek v strome).

Vyvazené vyhledávací stromy

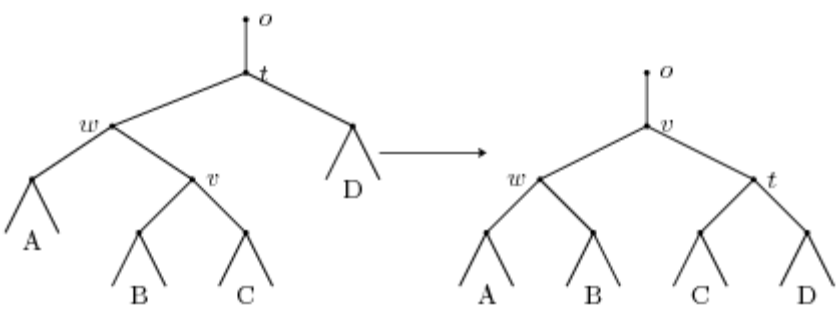
Kvuli zajištení vetší rychlosti (menší asymptotické casové složitosti) operací byly vytvoreny speciální druhy binárních vyhledávacích stromu, které jsou prubežne vyvažovány, aby mely max. výšku menší než $c \cdot \log n$, kde n je pocet uzlu a $c > 0$ nejaká konstanta. Samovyvazovaci stromy, maji nejaka integritni omezeni na svuj "tvar". Za cenu slozitejsich insert/deletu garantuji, ze strom se nikdy nedeformuje na seznam a tudiz bude stale "rychle" hledat. Dela se pomoci normalnich insert/deletu a nasledneho vyvazovani. Jednotlive struktury se lisi tim, podle ceho vyvazuji, a slozitosti operaci INSERT/DELETE/MEMBER.

Vyvazovani se dela pomoci rotaci Rotace(u,v), Rotace(v,u), DvojitaRotace(w,v,t), DvojitaRotace(t,w,v). Rotace(u,v) a Rotace(v,u) jsou jednoduche, DvojitaRotace(w,v,t) a DvojitaRotace(t,w,v) dvojite. Prislusne dvojice rotacii su symetricke.

Rotace(v,u)



DvojitaRotace(w,v,t)



AVL stromy

Zarucují max. logaritmický nárust výšky vzhledem k poctu prvku. Pro každý uzel x se v AVL stromu definuje **faktor vyvážení** jako rozdíl výšky jeho levého a pravého podstromu. AVL stromy jsou BVS navic obsahujici pravidlo, ze hloubka leveho a praveho podstromu se lisi max o 1.

Implementacia

Pro ucely vyvazovani maji ve vrcholech hodnotu -1, 0 , 1 znamenajici, zda aktualne je kratsi leva vetev, jsou stejne, nebo je kratsi prava vetev. Prvek, ktery bude po vlozeni treba vyvazovat ma hodnotu -1 a pridavame do levého podstromu nebo 1 a pridavame do praveho. Ve vsech ostatnich pripadech neni treba vyvazovat. Pri vkládání a odebírání je navíc nutné prubežne (nejhure až ke koreni) upravovat indikaci faktoru vyvážení jednotlivých uzlu.

Věta 7.4.1. *AVL-strom o n vrcholech má výšku nejvýše* $2 \log n$.

Důkaz. Označme $N(h)$ minimální počet vrcholů AVL stromu výšky h. Můžeme tedy $N(h)$ definovat takto:

$$N(0) = 1$$
$$N(1) = 2$$
$$N(h) = 1 + N(h-1) + N(h-2)$$

Je zřejmé, že platí $N(h) \geq 2N(h-2)$ nebo $N(h-2) \leq N(h-1)$.

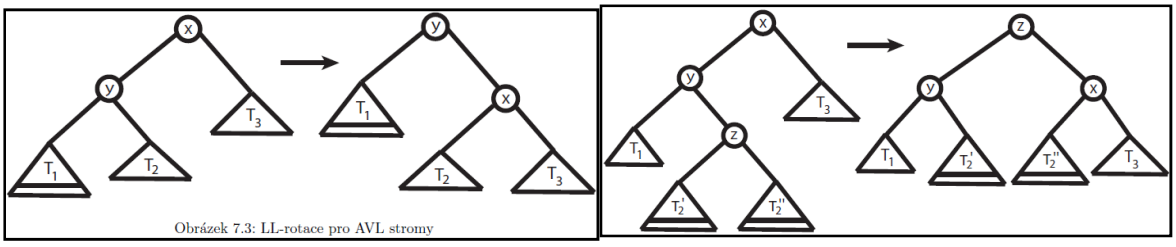
Odtud dostaneme, že $N(h) \geq 2^{h/2}$ a tedy $h \leq 2 \log N(h)$. Nerovnost $N(h) \geq 2^{h/2}$ dokážeme indukcí:

1. $N(0) = 1 \geq 2^{0/2} = 1$
$N(1) = 2 \geq 2^{1/2}$

2. $N(h) \geq 2N(h-2) \geq 2 \cdot 2^{h/(2-1)} = 2^{h/2}$

Operace

Insert - je treba maximálně jedno vyvazovani. Rotace bezi v case O(1).



Obrázek 7.3: LL-rotace pro AVL stromy

Member probiha standardne.

Delete muze zpusobit az **log n vyvazovani**, protoze je mozne, ze vyvazenim po ubrani prvku se snizi vyska podstromu a koren o patro vyse se stane nevyvazenym. Vyvazovani u Delete probiha kombinacemi rotaci Rotace(u,v), Rotace(v,u), DvojitaRotace(w,v,t), DvojitaRotace(t,w,v) stejne jako u insertu.

Složitost všech operaci v AVL stromech je O(log N).

Červeno-černé (RedBlack) stromy

Cerveno-cerné stromy jsou binární vyhledávací stromy s garantovanou max. výškou O(log n), kde n je pocet uzlu, tj. operace na nich mohou mít asymptotickou casovou složitost O(log n). Každý uzel ma priradenou barvu - **cerveny** nebo **cerny**.

Definice: Binární vyhledávací strom T se nazývá **červeno-černý**, jestliže každý vrchol je obarven červeně nebo černě a platí následující podmínky:

1. Listy jsou černé (NILL se považuje za tzv. **externí uzlel**, který je vždy černý).
2. Pokud má červený vrchol otce, je otec černý.
3. Všechny cesty z kořene do listu mají stejný počet černých vrcholů.

Nejprve ukážeme, že pro červeno-černý strom T reprezentující množinu S platí $\mathrm{vyska}(T) = O(\log(|S|))$. Odtud plyne, že červeno-černé stromy patří mezi vyvážené binární vyhledávací stromy. Předpokládejme, že T je červeno-černý strom, který má na cestě z kořene do listu právě k černých vrcholů. Pak pro počet vrcholů $\#T$ stromu T platí

$$2^k - 1 \leq \#T \leq 2^{2k} - 1.$$

Nejmenší takový strom má všechny vrcholy obarvené černě a je to úplný pravidelný binární strom o vysce $k - 1$, což dává dolní odhad. Největší takový strom má všechny vrcholy v sudých hladinách obarveny červeně a v lichých hladinách černě, je to úplný pravidelný binární strom o vysce $2k - 1$ a tím je dán horní odhad. Tedy $k \leq 1 + \log \#T$. Protože velikost množiny S je počet vnitřních vrcholů, dostáváme, že $\#T = 2|S| + 1$. Z vlastností červeno-černých stromů plyne, že

$$k \leq \mathrm{vyska}(T) \leq 2k.$$

Shrneme tato fakta.

Tvrzení. *Červeno-černý strom T reprezentující množinu S splňuje* $\mathrm{vyska}(T) \leq 4 + 2 \log |S|$.

Operace

Insert probiha klasicky jako u bvs, s pripadnym přebarvením, rotaci nebo dvojitou rotaci. Vklada sa vždy uzol s nativnou cervenou farbou.

U **deletu** je zaruceno, ze se spotrebuji maximalne dve rotace nebo rotace+dvojrotace (su aj pripady, kde sa to cele vyriesi iba prefarbenim uzlov porusujucich nejaku podmínku rb stromu) oproti AVL, ktere pri delete potrebuje az h (tedy log n) rotaci.

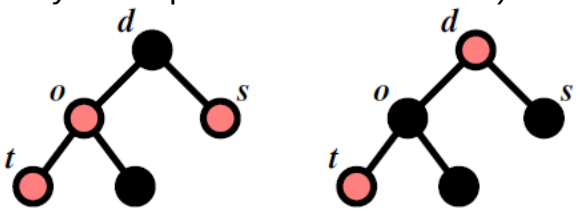
Red black tree maji **insert/delete i member O(log n)**, ale jsou konstatntne krat lepsi nez AVL; coz se taky hodi.

Implementacia

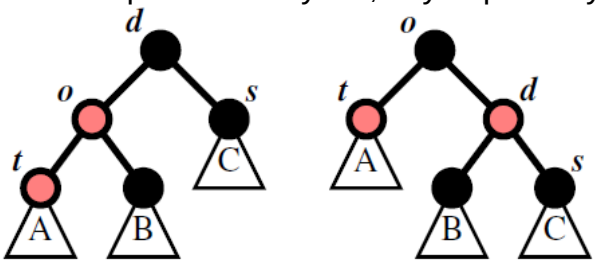
Obe operace mají diky garantované max. výšky garantovanou cas. složitost O(log n). Protože bez porušení vlastností cerveno-cerných stromu lze koren vždy prebarvit nacerno, mužeme pro ne predpokládat, že koren stromu je vždy cerný.

Insert(t) vypadá následovne:

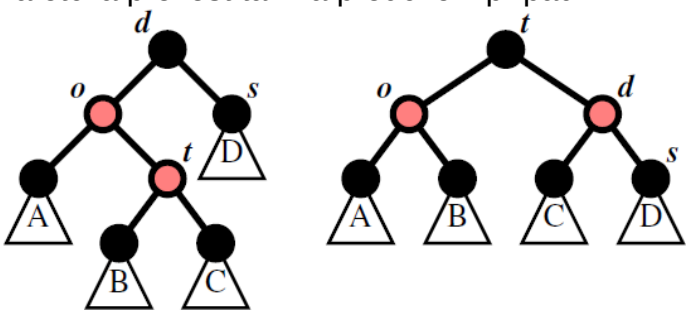
- Nalezení místa pro vložení a pridání nového prvku jako v obecných bin. vyhl. stromech, nový prvek je cerveny.
- Pokud je jeho otec cerný, mužeme skoncit – vlastnosti stromu jsou splnené.
- Pokud je cervený, musíme strom upravovat (tady predpokládám, že otec pridávaného uzlu je levým synem, opacný pripad je symetrický):
  - Je-li i strýc cervený, prebarvit otce a strýce nacerno a prenést chybu o patro výš (je-li ded cerný, koncím, jinak mužu pokracovat až do korene, který už lze prebarvovat beztrestne).



  - Je-li strýc cerný a pridaný uzel je levým synem, udelat pravou rotaci na dedovi a prebarvit uzly tak, aby odpovídaly vlastnostem stromu.



  - Je-li strýc cerný a pridaný uzel je pravým synem, udelat levou dvojrotaci na otci a prevést tak na predchozí prípad.

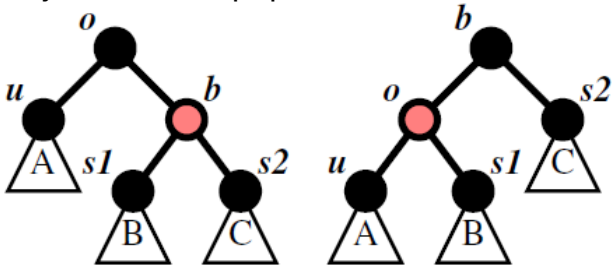


Delete(x) se provádí takto:

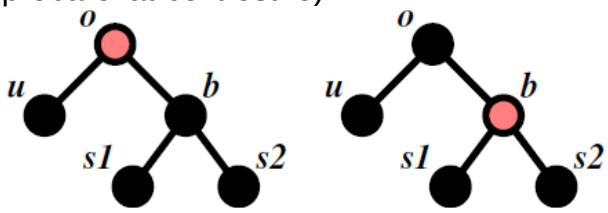
- Odstraním uzel stejne jako v predchozím prípade. Opravdu odstranený uzel má max. jednoho syna. Pokud odstranovaný uzel byl cervený, neporuším vlastnosti stromu, stejne tak pokud jeho syn byl cervený – to reším jeho prebarvením nacerno.
- V opacném prípade (tj. **syn odebíraného – u – je cerný**) musím udelat násl. úpravy (předp. že x je levým synem svého nového otce, v op. prípade

postupuji symetricky). U prohlásím za „**dvojite cerný**“ a této vlastnosti se pokouším zbavit.

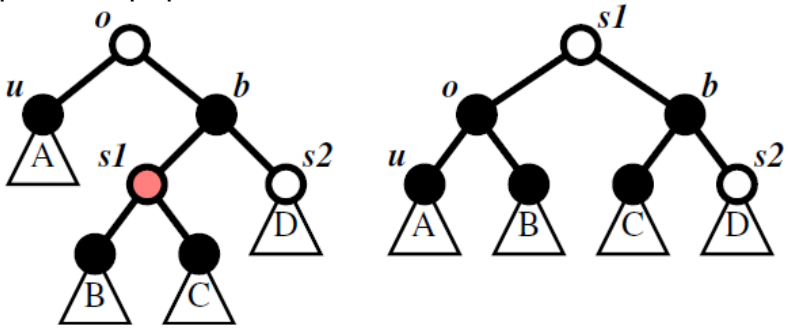
- Pokud je bratr u (bud b) cervený, pak má 2 cerné syny – provedu levou rotaci na rodici u, prohodím barvy rodice o a uzlu b. Prevedu tak situaci na jeden z násl. prípadu:



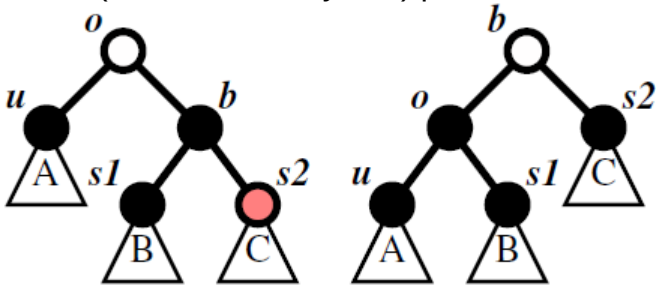
- Je-li b cerný a má-li 2 cerné syny, prohlásím u za cerný a prebarvím b nacerveno, rodice prebarvím bud na cerno (a koncím) nebo na „dvojite cernou“ a propaguji chybu (mohu dojít až do korene, který lze prebarovat beztrestne).



- Je-li b cerný, jeho levý syn cervený a pravý cerný, vymením barvy b s jeho levým synem a na b použiji pravou dvojrotaci, címž dostanu poslední prípad:



- Je-li b cerný a jeho pravý syn cervený, prebarvím pravého syna nacerno, odstraním dvojite cernou z u, provedu levou rotaci na b a pokud mel původně b (a u) cerveného otce, prebarvím b nacerveno a tohoto (ted už levého syna b) prebarvím nacerno.



Haldy

Halda je dynamická množina se stromovou strukturou, u ktereho pro vsechny prvky plati ze **hodnota (otec) <= hodnota(syn)**. Haldy s touto nerovností jsou tzv. **min-heapy**, pokud je nerovnost opacná, jde o **max-heap**.

Obvykle operace na haldach:

- **Min** - vrati nejmensi prvek (koren) - O(1).
- **DeleteMin** - (vrati a) smaze nejmensi prvek - slozitost zavisi na typu haldy.
- **Insert** - vlozi novy prvek.

Podoperace v ramci jinych operaci:

- **UP** (probublava prvek smerem nahoru (meni s otcem, dokud je hodnota otcovského prvku vetsi) - O(h) = O($\log_d$ (n)).
- **Down** (probublava prvek smerem dolu (meni s nejmensim synem, dokud je syn vetsi) - O(dh) = O(d.$\log_d$(n)).

D-regularni haldy - d-arni strom ($d \geq 2$), v kazdem patre se listy vytvari od leva do prava az do zaplneni patra. Plne patro ma d^k vrcholu (k = cislo patra od korene; koren = 1). Otec ma mensi hodnotu nez synove, pri ocislovani vrcholu dle vytvareni (priechod stromu od korena do sirky; root ma cislo = 1) lze jasne rici, jaka cisla budou mit synove, jake ma otec (t.z.v. prirodzene ocislovanie). Pro 2-regularni haldy lze ty souradnice pocitat i bitovym posunem.

Implementace

Pro d-regulární haldy jednoduše proveditelná jejich datová reprezentace polem (bez pointeru), kde pri indexování od 1 (BFS od kořene) má uzel na indexu i:

- Jeho synové mají index: d(i − 1) + 2, d(i − 1) + 3, ..., di + 1
- Rodice na indexu $\left\lceil \frac{i-1}{d} \right\rceil$

Operace

MakeHeap: mame-li nejake prvky predem, muzeme haldu sestavit jinak nez posloupnosti insertu a usetrit tak cas az na O(n). To se dela tak, ze postavime strom (tvar zname z poctu vrcholu a d) a naplnime jej libovolne danymi hodnotami. Pak prochazime odzadu a volame down na vsechny prvky. Slozitost vyjde o(n). (Slozitost makeheap <= n/2.c.1 + n/4.c.2 + … + n/2^h.c.log2n <= c.n, kde c je doba potrebna pro probublani prvku o jedno patro a n je pocet prvku)

Insert: O($\log_d$(n)) - Pridame prvek do posledni hladiny co nejvice vlevo (pripadne nova hladina) a UP.

DeleteMin: O(d.$\log_d$(n)) - Prehodime koren a posledni prvek (nejpravejsi v posledni hladine), ten smazem a na novy koren potom DOWN.

DecreaseKey: Snizit hodnotu prvku - nejde obecne v halde udelat, protoze neumi hledat prvky. Pokud ziskame pozici prvku jinak, tak O($\log_d$n) - UP.

IncreaseKey: Viz. DekreaseKey, DOWN - O(d. log(n)).

Delete: Prvek nahradime hodnotou mensi nez koren, UP, DeleteMin - O(d. logd(n)).

Leftist Haldy

Binarni strom se nazývá **leftist**, pokud pro něj platí:

- Pro kazdeho syna je urceno, zda je levy nebo pravy.
- Pokud ma prvek jednoho syna, je levy.
- Pokud ma dva syny, pak npl(leveho) >= npl(praveho), kde npl(x) je delka nejkratsi cesty z x do vrcholu v podstromu x, ktery ma max jednoho syna.

Letist halda je leftist strom, kde navyše platí: ∀ prvek v stromu T: otec(v) ≤ v.

Prava cesta – cesta ze startovniho vrcholu pouze po pravych synech, posledni prvek nema praveho syna. Délka pravé cesty z vrcholu v je ≤ log (počet vrcholů v podstromu vrcholu v).

MERGE(T1,T2) , složitost: O(log(n1 + n2)); n1 = |T1|, n2 = |T2|:

- když T1 je prazdna return T2
- když T2 je prazdna return T1
- když koren T2 < koren T1 -> prohod T1 a T2; pravy syn T1 -> MERGE(T2, podstrom praveho syna korene T1); když npl(levého syna korene T1) < npl(praveho syna korene T1) -> prohod syny; npl(korene T1) := npl(praveho syna korene T1) + 1; return T1.

Insert: udelej jednoprvkovou haldu a pak MERGE - O(log n).

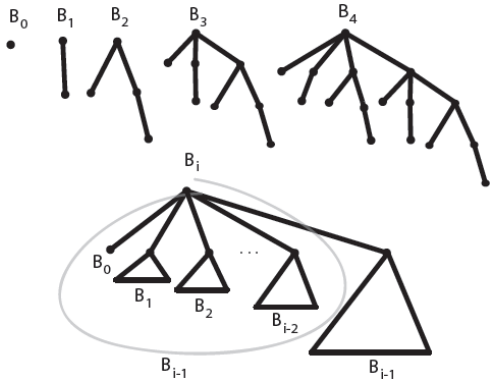
DeleteMin: MERGE(strom leveho syna a strom praveho syna) - O(log(n1 + n2)).

DecreaseKey(x) : odtrhni podstrom T1 vrcholu x, zmensi ohodnoceni vrcholu x, uprav cast T2=T-T1 aby splnovala kriteria pro Leftist haldu, MERGE(T1,T2) – složitost O(log(n))

IncreaseKey a **Delete** O(log(n)).

Binomialni haldy

Binomiální strom Bi je rekurzivně definován jako strom sestávající se z kořene a jeho dětí B0,B1, ...,Bi-1. Každý strom má vlastnost haldy, tj. pro každou stromovou hranu platí klíč otce < klíč syna.



Tvrzeni (pozorovani):

- Bi obsahuje 2^i prvku.
- Koren stromu Bi ma i synu.
- Delka nejdelší cesty z korene do listu ve stromu Bi je i (tj. vyska stromu Bi je i).
- Podstromy urcene syny korene Bi jsou izomorfni se stromy B0, B1, …, Bi-1.

Binomiální halda je soubor binomialnich stromů takových, že:

- každý strom je izomorfní s nějakým Bi
- žádné dva stromy v halde nejsou izomorfní
- existuje jednoznačná korespondence mezi vrcholy reprezentované množiny a vrcholy stromů taková, že prvek odpovídající otci je menší než prvek odpovídající vrcholu.

Nejčastěji je binomialni halda implementována jako pole ukazatelů, kde i-tý ukazatel ukazuje na kořen stromu Bi nebo je NIL. Muze byt doplnena o ukazatel MIN ukazující na strom s nejmenším klicem v koreni.

MERGE(T1, T2): O(log n) - jako binarni scitani - dva stromy B_i se sleji do stromu B_{i+1}.

(T1, T2 binom. haldy velikosti n1, n2)

P = 0, i = 0, T = 0

while i <= log(n1 + n2) **do**
S1 = strom v T1 izomorfní s Bi (pokud neexistuje, tak S1 = 0)
S2 = strom v T2 izomorfní s Bi (pokud neexistuje, tak S2 = 0)

if S1, S2, P = 0 **then** i = i + 1; continue; **end if**

if jeden strom z S1, S2, P je neprázdný **then**
  vložím tento strom do T, P = 0
end if

if dva stromy z S1, S2, P jsou neprázdné **then**
  spojím tyto stromy a výsledek vložím do P
end if

if všechny stromy z S1, S2, P jsou neprázdné **then**
  P vložím do T, spojení S1, S2 vložím do P
end if

i = i + 1

end while

Min - O(1) - je drzen ukazatel na minimum, ktery se aktualizuje pri operacich merge, delete_min.

Insert - O(log n) (amortizovane pro hodne insertu O(1)) - vyrobi se jednoprvkova halda a ta se merguje do binomialni haldy.
Delete se nedá rozumně provést, museli bychom přebudovat celý strom.

DeleteMin - O(log n) - "utrhneme" koren stromu obsahujiciho minimum , ze vzniklych stromu určených podstromy synu co prisli o otce vyrobime haldu a zavolame merge dvou hald.

DecreaseKey - O(log n) - zmensime a nechame probublat (musime mit ukazatel na prvek, nejde vyhledavat).

Lina implementace binomialnich hald

Prinasi MERGE v O(1). Změníme definici - vynecháme podmínku 2 z definice binomiálních hald, tj. ted' v naší binomiální haldě mohou být izomorfní stromy (i když jen dočasně). Další změna spočívá ve změně reprezentace binomiální haldy - haldu reprezentujeme dvojitým kruhovým spojovým seznamem přes kořeny stromů (kruhový spojový seznam umožňuje přidávání a odebírnání prvků v čase O(1)). Operaci MERGE(T1, T2) pak můžeme provést konkatenací seznamů T1 a T2. Jenom to by nefungovalo, musíme ještě změnit operace MIN, DELETEMIN.

Idea: „vyvažování" staci provádět jen při (po) operacich MIN a DELETEMIN, kdy je steje za potrebi prohledat vsechny stromy (ak si neudrzujem ukazatel na MIN). Izomorfní stromy se odstrani procesem zvanym konsolidace.

Složitost operací MIN / DELETEMIN je O(log n). MERGE (a tedy i INSERT) je tedy pouha konkatenace seznamu T1 a T2 – O(1).

DELETEMIN / MIN:

Při prohledávání prvků reprezentovaných kořeny stromů seřadíme stromy do množin Q_i, i = 0, ..., n , kde Q_i je množina všech stromů v T izomorfních s B_i. Pak přichází **konsolidace**:

```
i = 0, T = 0;
while ∃Q_i ≠ 0 do
   while |Q_i| > 1 do
      vezmeme dva stromy z Qi, spojíme je, výsledek dáme do Q_{i+1};
   end while
   if Q_i ≠ 0 then
      strom z Q_i dám do T;
   end if
   i = i + 1;
end while
```

DELETEMIN / MIN - umístí stromy po odtržení nejmenšího prvku do odpovídajících množin Q_i. (v množině Q_i jsou stromy izomorfní s B_i) Poté provede konsolidaci - upraví haldu do podoby, kdy je každý řád zastoupen nejvýše jedním stromem. Konsolidace běží v O(logn) plus vyčerpá účty stromů, které zaniknou při slévání.

Konsolidace probíhá takto:

Vytvořím pole délky log n, které je prázdné O(log n). Procházím spojový seznam vrcholů stromů v haldě a jeden strom za druhým vyjmu a dávám do pole vytvořeného v kroku 1, přičemž se vždy provede příslušné slití. Pokud strom zanikne, tak práci zaplatíme z jeho účtu. Pokud strom nezanikne, tak práci platíme z účtu konsolidace! O(logn). Z pole vytvoříme spojový seznam! O(logn). DELETEMIN tedy potřebuje O(log n) do účtů nově vytvořených stromů. O(logn) na jejich zavedení do spojového seznamu. O(logn) na konsolidaci.

Při konsolidaci vždy zároveň vyhledáme nové minimum.

Fibonacciho haldy

Fibonacciho haldy vycházejí z binomiálních hald, formálně se liší v podstatě pouze tím, že v haldě povolíme i jiné stromy než binomiální. Toto nám umožní implementaci operace DECREASEKEY, která nebyla v binomiálních haldách při zachování složitosti ostatních operací možná. Řád uzlu a stromu je definován jako u binomiálních hald. Slévání se provádí pouze mezi stromy stejného řádu.

Fibonacciho haldy mají nízkou casovou složitost operací – amortizovane O(1) pro INSERT a DECREASEKEY. Operace DELETEMIN je O(log n).

Implementace operací **MERGE(h1,h2)**, **INSERT(h,i)**, **MIN / DELETEMIN(h)** je stejná jako u binomiálních hald v ”líné” verzi.

DECREASEKEY(h,i,d):

1. Snížím klíč prvku i o d.
2. Podstrom i s kořenem i odřízneme a jako samostatný strom ho zavedeme do haldy (tj. zařadím do spojového seznamu kořenů stromů v haldě)) - O(1).
3. Abychom udrželi stromy dostatečně ”košaté” tak od každého vrcholu x mohou být odříznuti nejvýše 2 synové. Po odříznutí 2. syna je odříznut i sám vrchol x.

Přestože jedna operace DECREASEKEY může vyvolat kaskádu řezů, je jeji amortizovaná složitost O(1). Pomocí účetní metody dokážeme, že to platí. Při odřezávání syna vrcholu x zaplatí operace DECREASEKEY:

- 2 jednotky na účet x
- 1 jednotku na účet vzniklého stromu
- 1 jednotku za práci (odříznutí a zařazení)

Při odříznutí druhého syna jsou na účtu vrcholu x 4 jednotky -> mohu zopakovat body 1) - 3).

DeleteMin(T) - prohledame seznam a najdeme strom s nejmensim vrcholem. Ten odtrhneme a syny odznacime. Z podstromů synu nam vznikly nove stromy. Ty pridame do mnoziny T a provedeme konsolidaci (ze stromu se stejnym stupnem i korene udelame strom se stupnem i+1 spojenim korenu. Provadime dokud je v nejake kategorii vice nez jeden zastupce). **Slozitost**: O(pocet stromu v halde + a), kde a je pocet synu smazaneho prvku.

DecreaseKey(x,T) - odtrhneme podstrom x a zmensime ohodnoceni. Pokud byl otec x oznacen, tak jej (s podstromem) taky odtrhnem, rekurzivne az ke korenu (kdyz uzlu utrhneme jednoho syna, tak ho oznacime, kdyz chceme oznacenemu uzlu utrhnout syna, misto toho utrhneme podstrom i s danym uzlem). **Slozitost**: O(pocet odznacovanych vrcholu).

IncreaseKey a **Delete** analogicky ke dvema predeslym (odtrhnem prvek, ze synu stromy, pripadne propagujeme odtrhavani az ke korenu. **Slozitost**: O (pocet odznacovanych vrcholu + b), kde b je pocet synu smazaneho prvku.

Trie

Trie (z angl. retrieval) jsou stromova datova struktura urcena k reprezentaci "slovnikove" mnoziny, rovinna implementace slovniku. Jejich hlavni vyhodou je rychla operace member O(l). Dale zabiraji malo mista - pro ulozeni jednoho klice je treba jen amortizovane konstatni prostor. Taky se hodi pro hledani klice, ktery ma s hledanym (ale neobsazenym) klicem nejdelsi spolecny prefix (hledam ve slovniku "auto" a najde mi to "automat").

Universum U, jehoz nejakou podmnozinu S chceme reprezentovat tvori vsechna slova delky l nad mnozinou znaku Σ (abeceda). Kratsi slova, ktera bychom chteli vlozit doplnime nejakym vhodnym znakem z abecedy (napr. mezerou).

Definice: Trie nad Σ je konecny strom, jehoz kazdy vnitrni vrchlo ma prave k synu, ktere jsou jednoznacne ohodnoceny prvky Σ. Kazdemu vnitrnimu vrcholu trie odpovida slovo nad Σ delky nejvyse l:

- korenu odpovida prazdne slovo
- kdyz vrcholu v odpovida slovo α, pak v[a] (syn vrcholu v ohodnoceny pismenem a) odpovida slovo αa.

Definice: Trie nad Σ reprezentuje mnozinu S ⊆ U, jestlize:

- Pro každý list t je definovana boolean funkce rep(t), vracejici true, jestlize slovo reprezentovane t ∈ S a false, jestlize slovo reprezentovane t ∉ S.
- (prefixova podminka) Pokud vnirtrni vrchol p odpovida slovu α, tak ∃ slovo $\beta \in$ S takove, ze je α prefixem β.
- Pro kazde slovo $\alpha \in$ S existuje v trie list odpovidajici α.

Vrchol neobsahuje klic který reprezentuje (nemusi byt fyzicky ohodnoceny ziadnym pismenom), ale jeho pozice ve stromě udava ten klic. Např. slovník spisovatelů, kde listy ve trie by odpovídaly jménům jednotlivých spisovatelů a data uložená v nich by obsahovala seznam jejich děl.

MEMBER(x = x_1, …, x_l**):**
t := kořen; i := 1;
while t není list **do**
    t := t[x_i] // sestup podle znaku x_i;
    i := i + 1;
end while
return rep(t);

Pouzivame jednotlive znaky k indexaci a tim k hledani dalsi cesty ve stromu. Tedy složitost operace MEMBER je O(l), protože musíme projít nejvýše l vrcholů než dosáhneme listu (délka slov je nejvýše l).

INSERT(x):

t := kořen stromu, i := 1;
while t není list **do**
  a := i-té písmeno x, t := a-tý syn t, i := i + 1;
end while
while i <= l **do**
  vrcholu t přidej k listů ohodnocených písmeny ze Σ; Jejich rep := false;
  t := t[x_i];
  i := i + 1;
end while
rep(t) := true;

Pokud tak dlouhe slovo jěšte v slovniku není, tak pridavej vnitrni vrcholy az do hloubky l.

DELETE(x):

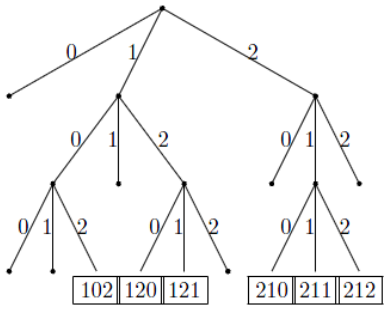
t := kořen stromu, i := 1;
while t není list **do**
  a := i-té písmeno x, t := a-tý syn t, i := i + 1;
end while
rep(t) := false; t := otec t;
while všichni synové t jsou listy s rep = false **do**
  zruš listy t;
  t := otec t;
end while

Analogicky k operaci INSERT dochází k rušení hladin v rámci jedné větve v případě že všechny listy v hladině mají hodnotu rep = false. Použili jsme obrat t := otec t. To lze provést bud' tak, že se vrchol kromě svých synů odkazuje i na svého otce a spotřebuje tak pamět' navíc, nebo se cesta z kořene do aktuálního vrcholu během sestupu ve stromu pamatuje na zásobníku. Tento trik se používá u všech stromových struktur.

Slozitosti:

MEMBER - O(l), **INSERT/DELETE** - O(lk). Pametova slozitost klesa od O(|S|lk) pro malo prvku (zadny spolecny prefix) po O(|S|), je-li reprezentovano cele U.

Příklad: Nechť Σ = {0, 1, 2}, l = 3 a S = {102, 120, 121, 210, 211, 212}.



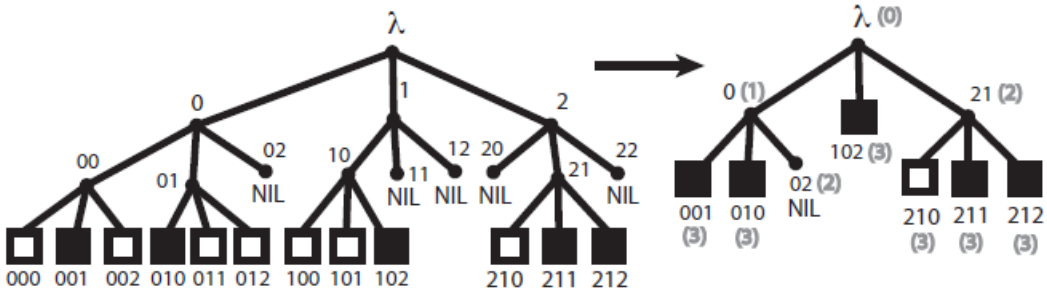
Komprimovane trie

Idea - v pripade, ze ma vnitrni vrchol nekomprimovaneho trie jen jednoho syna, neprinasi zadnou "pridanou hodnotu", protoze mnoziny prvku S reprezentovanych vrcholy v podstromu otce i syna jsou stejne. Rozsirime puvodni trie o funkce:

- **uroven(v)** - cislo urovne vrcholu v odpovidajicim nekomprimovanem trie
- **slovo(v)** - slovo odpovidajici vrcholu v

Komprese

Nyni lze vynechavat vrcholy, ktere maji jen jednoho syna, a ktery neni list s rep(x) = false a tohoto jednoho syna presunout na misto vynechaneho otce. To opakujeme, dokud takovy existuje.



MEMBER(x):
t := kořen;
while t není list **do**
    i := uroven(t) + 1;
    t := t[x_i]; // xi-tý list
end while
return rep(t) && slovo(t) = x;

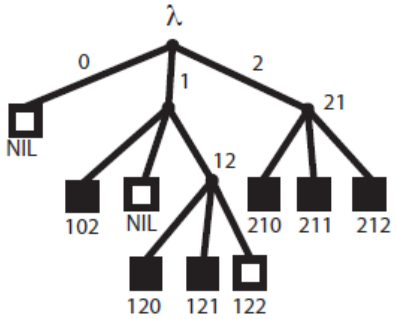
Upravene operace INSERT/DELETE musi pripadne rozsirovat/kratit strom. Slozitosti v komprimovanych trie je v nejhorším případě: Member - O(l), INSERT/DELETE - O(l+k) (může být nutné přidat/odebrat jeden vnitřní vrchol). V průměrném případě (za předpokladu rovnoměrného rozložení vstupních dat) je to očekávaná hloubka trie: MEMBER - O($\log_k(n)$), INSERT/DELETE O($\log_k(n) + k$).

Pametova složitost komprimovaných trie: O(nk) - maximalne n (|S|=n) vnitnich vrcholu, kazdy s k syny.

Jeste komprimovanejsi trie

Reprezentuji se maticemi. Maji pametovou narocnost O(n) a nepodporuji INSERT/DELETE.

Příklad (na komprimovanom trie):



	0	1	2
root	NIL	a	b
a	102	NIL	c
b	210	211	212
c	120	121	NIL

Chceme se zbavit položek NIL v matici reprezentující trie. Další komprese dosáhneme pomocí vektorů **hod** (vektor hodnot) a **rd**. Tyto vektory budou reprezentovat původní matici.

Algoritmus (O(rsm)) pro hledání rd a hod

Nechť M je matice typu r x s, má m významných míst ≠ NIL.

- pro každý řádek nalezneme počet míst ≠ NIL.
- setřídíme řádky Bucketsortem, tak že řádky s větším počtem míst ≠ NIL předcházejí řádky s menším počtem míst ≠ NIL.
- procházíme řádky v daném setřídění a pro každý řádek i nalezneme nejmenší číslo rd(i), že nedochází ke kolizi s předchozími řádky (tj. když $M_{i',j'} \neq NIL \neq M_{i,j}$) a řádek i' byl zařazen, pak rd(i) + j ≠ rd(i') + j'. Pak $M_{i,j} \neq$ NIL je uloženo ve vektoru hod na místě rd(i) + j.

Reprezentacia pomocou vektorov rd a hod

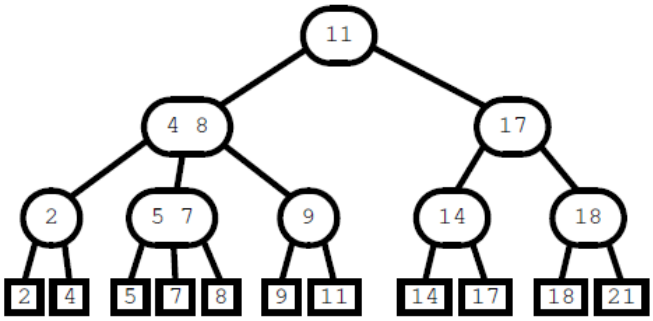
hod	210	211	212	120	121	a	b	102	NIL	c

	root	a	b	c
rd	4	7	0	3

B-stromy a jejich varianty.

Nechť a, b ∈ ℕ, a>=2, b>= 2a-1. Strom je **(a, b) strom**, když platí:

- Každý vnitřní vrchol kromě kořene má alespoň a a nejvýše b synů.
- Kořen má nejvýše b synů. Má alespoň 2 syny, nebo je listem.
- Všechny cesty z kořene do listu jsou stejně dlouhé.



Obrázek 8.1: Příklad (a, b) stromu

Věta: Mějme přirozená čísla a a b taková, že a ≥ 2 a b ≥ 2a−1. Pak pro každé kladné přirozené číslo n existuje (a, b)-strom, který má přesně n listů (tj. může být použit na reprezentaci libovolné množiny |S| = n). Když (a, b)-strom má přesně n listů, pak jeho výška je alespoň $\log_b n$ a nejvýše $1 + \log_a(\frac{n}{2})$. Tedy výška (a, b)-stromu je O(log n).

Důkaz: Zřejmě strom o výšce 0 má jediný list. Strom o výšce 1 má alespoň dva a nejvýše b listů. Strom o výšce 2 má alespoň 2a listů a nejvýše b^2 listů. Indukcí podle výšky h dostaneme, že pro každé n takově, že $\mathbf{2a^{h-1} \leq n \leq b^h}$, existuje (a, b)-strom o výšce h s n listy.
Přesněji, když (a, b)-strom o výšce h − 1 má k listů, tak přidáním další (h-té) hladiny dostaneme (a, b)-strom o výšce h s n listy pro každé n takové, že ka ≤ n ≤ kb.
Pro a ≥ 2 a pro b ≥ 2a – 1, k ≥ 2 platí kb ≥ (k + 1)a a odtud dostáváme požadované tvrzení. Tím je první část věty dokázána. Druhou část dostaneme zlogaritmováním nerovnosti mezi počtem listů a výškou stromu.

Struktura vniřního vrcholu v:

- $\mathbf{p_v}$ je počet synů
- $\mathbf{S_v[1, \ldots, p_v]}$ je pole ukazatelů na syny.
- $\mathbf{H_v[1, \ldots, p_v - 1]}$: $H_v[i]$ je maximální prvek v podstromu $S_v[i]$.

MEMBER(x):
t := kořen;
while t není list **do**
    i := 1;
    while $H_t[i] < x \wedge i < p_t$ **do**
        i := i + 1;
    end while
    t := $S_t[i]$;
end while
return t;

INSERT(x):
t := MEMBER(x);
if t nereprezentuje x **then**
    o := otec t
    vrcholu o přidej nového syna t' reprezentujícího x
    zařaď t' na správné místo mezi jeho bratry a uprav p_o, S_o a H_o
    t := o
    while p_t > b **do** {Štěpení}
        rozděl t na t_1 a t_2:
        do t_1 dej prvních $\lfloor b(b+1)/2 \rfloor$ synů t
        do t_2 dej prvních $\lceil b(b+1)/2 \rceil$ synů t
        o := otec t
        uprav p_o, S_o a H_o
        {při štěpení kořene ještě musíme vytvořit nový kořen}
        t := o
    end while
end if

DELETE(x):
t := MEMBER(x); navíc si zapamatuj vrchol u, v jehož poli H_u je x;
if t reprezentuje x **then**
    o := otec t
    odstraň t
    uprav H_o, H_u {...} a uprav S_o a p_o
    t := o
    while p_t < a ∧ t není kořen **do**
        v := bezprostřední bratr t
        if p_v = a **then** {smíme spojit}
            o := otec t
            sluč v a t do t
            uprav p_o, S_o a H_o
            t := o
        else {p_v > a, spojení by mohlo mít více než b synů}
            {Přesun}
            přesuň krajního syna v do t
            uprav H_{otec} t
        end if
    end while
    if t je kořen a má jen jednoho syna **then**
        smaž t
    end if
end if

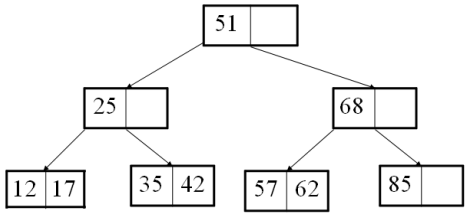
Jak volit parametry a, b?

Pro vnitřní paměť je vhodně a = 2 nebo a = 3, b = 2a. Pro vnější paměť je vhodné a = cca. 100, b = 2a. Pro minimalizaci paměťových nároků je výhodné b = 2a−1, pro minimalizaci časových nároků je výhodné b = 2a.

B-strom rádu m je výškove vyvážený strom, který má násl. vlastnosti:

1. Koren má minimálne 2 syny, pokud není sám listem.
2. Každý uzel krome korene a listu má nejméne $\left\lceil \frac{m}{2} \right\rceil$ a nejvíce m synu
3. Každý uzel ma nejmene $\left\lceil \frac{m}{2} \right\rceil - 1$ a nejvice m-1 datovych zaznamu
4. Všechny vetve (cesty od korene k listu) jsou stejne dlouhé.
5. Data v uzlu (datove zaznamy) jsou organizovana nasledovne:
   p_0, (k_1,p_1,d_1), (k_2,p_2,d_2),…, (k_n,p_n,d_n), u
   ;kde **p_0, p_1,…, p_n** jsou ukazatele na potomky, **k_1, k_2,…, k_n** jsou klice, **d_1, d_2,…, d_n** jsou asociovana data, **u** je nevyuzity prostor a zaznamy **(k_i,p_i,d_i)** jsou usporadany vzestupne podle klicu; $\left\lceil \frac{m}{2} \right\rceil - 1 \leq n \leq m - 1$
6. Odpovídá-li ukazateli p_i podstrom $U(p_i)$, potom plati:
   (i) Pro kazde k v $U(p_{i-1})$ je $k < k_i$ **(neredundantni)**, $k \leq k_i$ **(redundantni)**
   (ii) Pro kazde k v $U(p_i)$ je $k > k_i$, kde $i \in \langle 1,n \rangle$

Variantou jsou redundantní B-stromy, kdy všechna data jsou umístena v listech, vnitrní uzly obsahují pouze vyhledávací klíce. Jiná možnost je použití pouze klíce a odkazu na celý záznam, místo vkládání kompletních záznamu do stromu.



Neredundantní B-strom (2-3-strom)

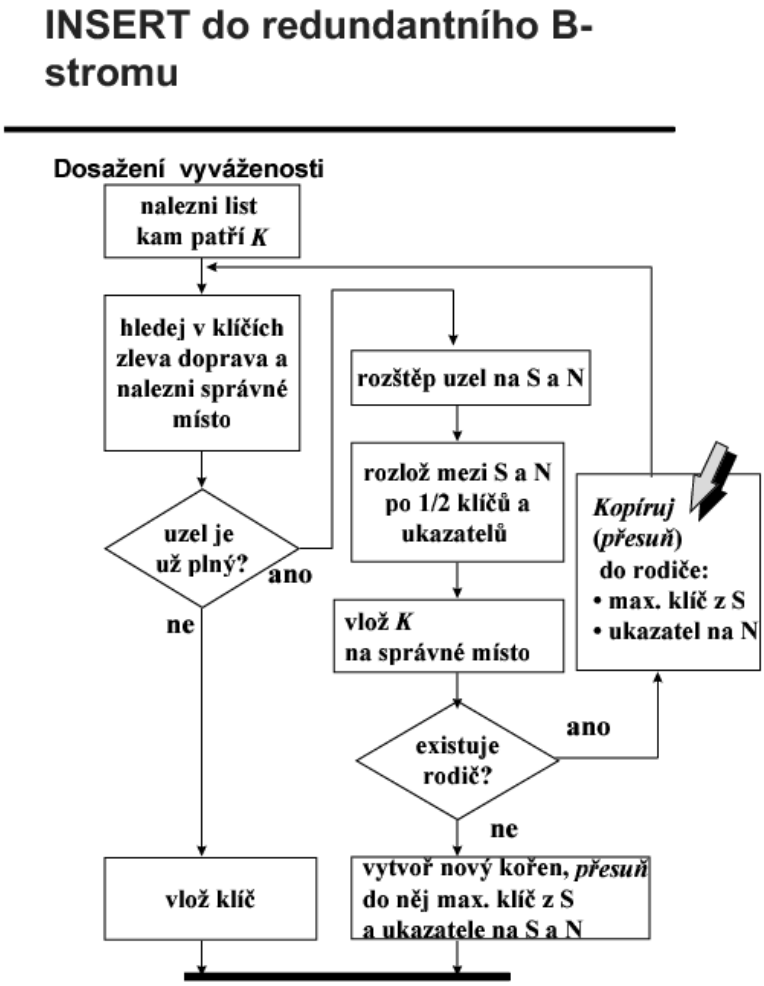
Operace

Member: Jdu od korene a v kazdem prvku porovnam hledanou hodnotu s maximalnimi prvky v podstromech a podle toho vyberu kam jit dal. Opakuju nez jsem v listu. Pokud se list rovna hledanemu prvku nasel, jinak strom neobsahuje prvek.

Insert: Vkládání se deje nalezením místa pro vložení. Jestlize po vlozeni zaznamu dojde k přeplněni stranky, potom se jeji obsah rozdeli do dvou stranek – první polovina do puvodni stranky a druha do nove; to ovšem vede k potrebe aktualizace rodičovské stranky:

- Opravi se hranicni klic v ridicím zaznamu puvodni stranky (na maximalni klic z opravene puvodni stranky)
- Vlozi se ridici zaznam nove stranky

Tento proces se muze opakovat az do korene, při jehož stepeni vznika nova korenova stranka. Proto se cesta od kořene do listu drží v bufferech.



Delete: Algoritmus ruseni prvku je zalozen na mechanizmu spojovani nezplněných stranek, který je dualni k mechanizmu vkladani. Tato dualita ovšem neznamena symetrii. Spojovani stranek muze probihat podle různých strategii, takze reverzni posloupnost operaci DELETE k posloupnosti operaci INSERT nemusi vest k temuz výchozímu stromu. Analogicky se zacina na listove urovni a muze se pokracovat az do korene. Vysledkem potom muze byt nejen zmensovani poctu obsazenych stranek, ale i snizovani vysky stromu. V pripade podteceni stranky (v uzle zustane v nem méne než $\left\lceil \frac{m}{2} \right\rceil$ synu) zkousi praveho sourozence a jde-li to provede slevani obou stranek (MERGE). V opacnem pripade zustava uzel (stranka) podhodnocen(a).

V redundantních B-stromech není nutné pri mazání odstranovat vyhledávací klíc z vnitrních uzlu – prvek s touto hodnotou se ve strome už nebude nacházet, ale vyhledávat podle jeho klíce je dál možné.

Merge: Provadi slevani stranek - (dane stranky a jejiho praveho/leveho sourozence). Musí zajistit odstraneni jednoho ridiciho zaznamu v rodicovskem uzlu. Další varianty jsou dve:

- Nelze-li slevat, provede se přerozděleni zaznamu mezi obema uzly.
- Zkousi se levi soused a bud se sleva, nebu se přerozděluje.

Join: Operace JOIN provede spojení dvou (a,b)-stromů T1 a T2 do jednoho (a,b)-stromu za předpokladu, že všechny prvky, které reprezentuje strom T1 jsou menší než prvky reprezentované stromem T2. JOIN vyžaduje čas O(rozdíl hloubek stromů) $\leq$ O(log($|S_1| + |S_2|$)). Pro operaci JOIN je vhodné spolu se stromem uchovávat také největší prvek reprezentované množiny.

Split: Operace SPLIT(x, T) provede rozdělení (a,b)-stromu T na dva (a,b)-stromy T1 a T2 tak, že T1 je (a,b)-strom reprezentující prvky z S < x a T2 je (a,b)-strom reprezentující prvky z S > x, kde S je množina, reprezentovaná (a,b)-stromem T. Základní myšlenkou pro implementace této operace je použití dvou zásobníků (a,b)-stromů. Procházíme strom T od kořene k listům a na každé úrovni vložíme do prvního zásobníku ty podstromy bratrů aktuálního vrcholu, které obsahují prvky menší než prvek reprezentovaný aktuálním vrcholem. Do druhého zásobníku vložíme podstromy s většími prvky.

Věta: Algoritmy pro operace MEMBER, INSERT, DELETE, MIN, MAX, JOIN a SPLIT v (a, b)-stromech vyžadují v nejhorším případě čas O($\log_a|S|$), kde S je reprezentovaná množina.

Optimalizace 1: Vyvažování stránek

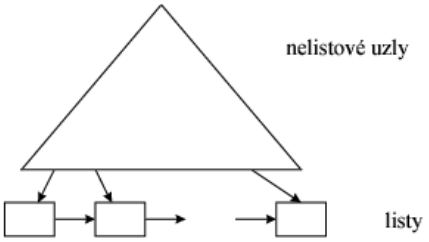
Lepší naplnenosti uzlu za cenu snížení rychlosti se dá dosáhnout pomocí **vyvažování stránek** – pri pretecení stránky nejdrív kontroluji, jestli nejsou volné sousední. Pokud ano, obsah sourozenců přerozdělim tak, aby byly stejně zaplněny. Změny jsou zapsány do rodičovské stránky. Tím se proces zastaví, protože se nevyvolává nové vkládání. Podobně je možné postupovat při mazání (i pokud není třeba slévat).

Optimalizace 2: Odložení štepení

Dalším vylepšením je **odložení štepení** – ke každému listu nebo skupine listu prísluší stránka pretecení, kam se vkládají záznamy, které se už do daného místa nevejdou. Posloupnost klíčů ve stránce je monotónní. Nové vkládání a štepení je provedeno až tehdy, jestliže se stránka přetečení i všechny príslušné uzly naplní. Takto upravený strom s více než 1 úrovní má vždy všechny listy zaplnené (za predpokladu nepoužití mazání). Prísluší-li stránky pretecení skupinám listu, musím je pri mazání a pridávání listu taktéž štepit a slévat.

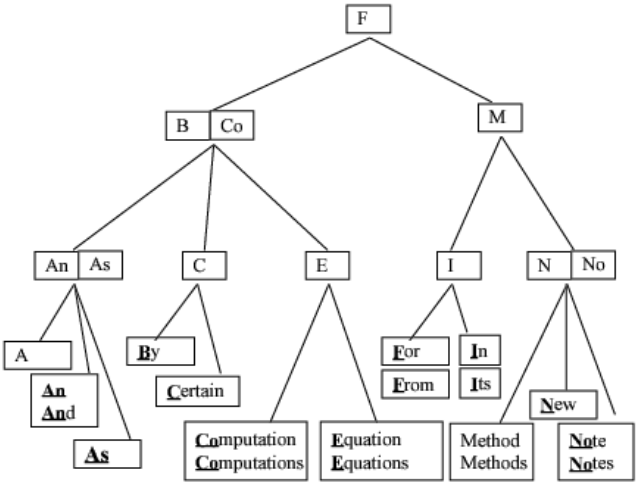
Operace insert a delete lze implementovat i pro paralelni zpracovani "zbytku" stromu jinym procesem. Provadim operace stepeni/spojovani jiz na ceste smerem dolu a pokud se odchylim od cesty nejakeho jineho procesu, muze ten pokracovat svou cestou, protoze je zaruceno, ze se nebude vracet zpet a pritom menit strom, ktery bych mohl menit ja. Cenou za tento pristup je, ze k nekterym stepenim/spojenim by pri beznem postupu nedoslo.

Definice (B+ stromy): jsou mírným vylepšením B-stromu pro zrychlení intervalových dotazu: všechny uzly ve stejné úrovni (a nebo jenom listy) jsou spojeny do spojového seznamu (možná je jednosmerná i obousmerná varianta).



Definice (B* stromy): (rádu m) jsou úpravou B-stromu na základe vyvažování stránek. Druhá podmínka B-stromu se upraví tak, že každý uzel krome korene má minimálne $\lceil (2m-1)/3 \rceil$ a max. m synu. Listy mají opet jen stejné rozmezí pro pocet datových záznamu. Pri vkládání prvku se stepení odkládá opet do té doby, dokud nejsou plní i sourozenci daného listu; potom se štepí bud 2 listy do 3, nebo 3 do 4 (bud s pomocí jednoho nebo dvou sousedních sourozencu). Odebírání podobne zahrnuje slévání 3 uzlu do 2 (nebo 4 do 3). Při obém lze ve složitejší variante zapojit ješte více uzlu.

Definice (Prefixové stromy): Tento druh stromu slouží k uložení dat, klícovaných retezci. Jde o redundantní stromy, data jsou uložena až v listech; vyhledávací klíce jsou vždy co nejkratší možné prefixy retezcu, nutné k odlišení uzlu. Celé hodnoty klícu (a další data) se nacházejí až v listech. Pri vkládání a štepení stránek se nejakou heuristikou hledá nejkratší prefix, který by vzniklé stránky oddelil. Vylepšená varianta neukládá u synu predponu klíce, kterou má rodic – je to pametove efektivnejší, ale zvyšuje výpocetní nároky.

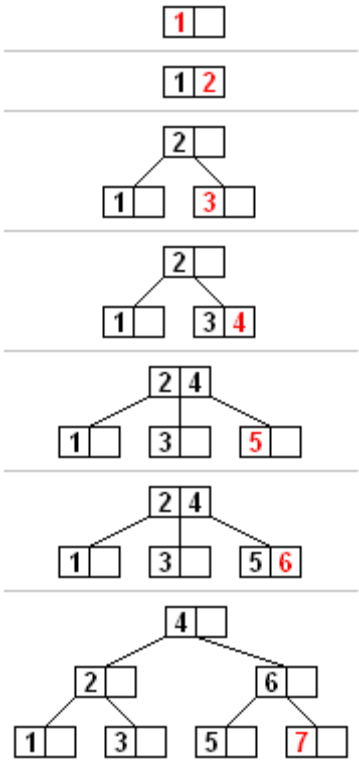


Definice (Stromy s promennou délkou záznamu): Jde o modifikaci B-stromu, která umožnuje do nej uložit záznamy promenné délky. Listy se neštepí podle poctu záznamu, ale zhruba na poloviny podle velikosti dat. Druhá podmínka B-stromu se upraví následovne: celková délka záznamu v jednom uzlu je minimálne B/2 a maximálne B (kde B je nejaká zvolená hodnota, vetš. velikost stránky na disku). Existuje i varianta s podmínkou „2/3“, jako mají B*-stromy.

Problémem této struktury je tendence delších záznamu ke stoupání ke koreni, címž se snižuje arita záznamu. To se reší hledáním delícího záznamu s min. délkou tak, aby vzniklé uzly splnovaly podmínky stromu (a je to docela nárocné). Navíc štepení je složitejší – 1 stránka se muže štepit na 3 (vložím-li hodne dlouhý záznam), muže dojít ke zmenšení stromu pri vložení apod., bežne se používá obecný algoritmus nahrazování, jehož speciální prípady jsou insert a delete.

Definice (Vícerozmerné B-stromy): Používají se, je-li potreba efektivne hledat záznamy podle více atributu. Jde o propojenou množinu stromu. K jednotlivým atributum príslušejí prvky pole odkazu na seznamy stromu, ve kterých se podle daných atributu dá hledat. Pro první atribut je potreba jen 1 strom, v nem je pro každý klíc odkaz na celý strom 2. atributu (pro další je to podobné). Stromy stejného atributu jsou ve spojovém seznamu. Mohu hledat všechny záznamy, pro které znám hodnoty všech atributu, nebo jenom jejich podmnožinu – vyžaduje to projít více stromu, ale není třeba množinových operací.

Priklad (insert posloupnosti 1,2,… do neredundantního B-stromu)



Hašování: řešení kolizí.

Uvod

Definice (slovníkový problém): Dáno univerzum U, máme reprezentovat S ⊆ U a navrhnout algoritmy pro operace:

- **MEMBER(x)** - zjistí zda x ∈ S a pokud ano nalezne kde,
- **INSERT(x)** - pokud x ∉ S, vloží x do struktury reprezentující S,
- **DELETE(x)** - když x ∈ S, smaže x ze struktury reprezentující S.

Mejme univerzum U a množinu S ⊆ U takovou, že |S| << |U|. Dále mejme funkci h : U → {0, 1, . . . ,m − 1}. Množinu S potom reprezentujeme tabulkou (polem) o velikosti m tak, že prvek x ∈ S je uložen na rádku h(x).

Definice (Hašovací funkce, kolize): Funkci h : U → {0, 1, . . . ,m − 1} potom nazýváme hašovací funkcí. Situaci h(s) = h(t), pro s ≠ t; s, t ∈ S nazveme kolize.

Jelikož mohutnost univerza U je vetší než velikost hašovací tabulky, nelze se kolizím úplne vyhnout.

Neformalne

Hasovani - reprezentace velke mnoziny do male - kdyz rozsah moznych hodnot vyrazne prevysuje pocet k ulozeni. Spocitam hasovaci funkci, ktera mi (v konstatnim case) ukaze na misto do tabulky, kde se dany prvek nachazi. Nejjednodussi hasovaci funkci je asi modulo p, kde p je nejake vhodne prvocislo (a zaroven velikost tabulky) vzhledem k poctu ukladanych prvku.

Zakladnim problemem hasovani jsou kolize (pro dva ruzne vstupy vrati hasovaci fce stejny vystup). Jednotlive typy hasovani se lisi hlavne tim, jak se vyporadavaji s kolizemi.

Predpokladame **rovnomerne rozlozeni vstupni mnoziny S**, h(x) spocitatelne v O(1), h rozdeluje rovnomerne, n=|S|, m = velikost tabulky, α = n/m - faktor naplneni tabulky.

Hasovani se separovanymi retezci

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

Member - h(x), projdu seznam; **Insert/Delete** - h(x), projdu a pripadne upravim seznam;

- Ocekavana delka retezce = α
- Ocekavana doba member/insert/delete = O(α)
- Ocekavana delka nejdelsiho retezce O(log n/(log log n))

Očekávaný počet testů

Slozitost prohledání seznamu se může lišit podle toho, jestli tam hledaný prvek je nebo není. **Uspěšným případem** nazveme takovou Operaci(x), kde x ∈ S, **neúspěšný případ** je x ∉ S. V úspěšném případě prohledáváme průměrně jenom polovinu seznamu.

- Očekávaný čas pro neúspěšný případ je $O\left(\left(1-\frac{1}{m}\right)^n+\frac{n}{m}\right) \approx e^{-\alpha}+\alpha$.
- Očekávaný čas pro úspěšný případ je $O\left(\frac{n}{2m}\right)$

H s usporadanymi retezci

Jako vyse, ale prvky v retezcich jsou serazeny ve vzrustajicim poradi dle klice. Trochu rychlejsi neuspesny member (a tim padem insert), ale slozitost je lepsi jenom o konstantu.

Obe metody maji nevyhodu v neefektivnosti vyuziti pameti (zatím co některá mista původního pole (hash tabulky) zustavaji nevyuzite, musime pro kolidující prvky dodatecne alokovat pamet).

H s premistnovanim

Tabulka obsahuje krome klice jeste hodnoty vpred (next) a vzad (prev), pomoci kterych se tvori retezec. **Member** - h(x), postupuje retezcem. **Insert** - h(x), pokud je obsazeno, tak pokud tady retezec zacina (vzad == null), tak vyberu nejake misto a tam vlozim prvek a upravim hodnoty vpred a vzad. Pokud misto je obsazeno a "vzad" toho prvku neni null, tak jej nekam presunu, upravim patricne vsechny hodnoty vpred a vzad a tady zacnu retez pro prvky s has. hodnotou h(x). **Delete** - najdu, smazu a upravim retez.

Member/Insert/Delete – stejny jako hashovani se separovanými retezcami ale neridi se ukazatelmi, ale zaznamami prev a next.

Slozitost: složitost všech operaci stejna jako u hashovani se separovanými retezci.

Priklad

Příklad: U = {1, 2, . . . , 1000}, S = {1, 7, 11, 53, 73, 141, 161} a h(x) = x mod 10. Tabulka vznikla následující posloupností operací: INSERT(1), INSERT(141), INSERT(11), INSERT(73), INSERT(53), INSERT(7), INSERT(161).

řádek	key	next	previous
P(0)			
P(1)	1	9	
P(2)			
P(3)	73	6	
P(4)			
P(5)	161		8
P(6)	53		3
P(7)	7		
P(8)	11	5	9
P(9)	141	8	1

H se dvema ukazateli

H. s presuny ztraci cas s presuny :). H se 2 ukazateli obsahuje v tabulce policka vpred a zacatek. Vpred je pro tvoreni retezce a T[i].zacatek ukazuje na prvek, kde zacina retezec pro prvky h(x) = i. Položka begin v j-tém rádku je vyplnena práve tehdy, když reprezentovaná množina S obsahuje prvek s ∈ S takový, že h(s) = j. Algoritmy jsou potom podobné tem u hašování s premístováním, ale přemísťování prvku je nahrazeno odpovídajícími zmenami v položce begin daných rádku.

Priklad

Tabulka vznikla posloupností operací: INSERT(1), INSERT(141), INSERT(11), INSERT(73), INSERT(53), INSERT(7), INSERT(161). Pak provedeme příkaz INSERT(28) a jako nový řádek zvolíme 4. řádek tabulky.

řádek	key	next	begin
P(0)			
P(1)	1	9	1
P(2)			
P(3)	73	7	3
P(4)			
P(5)	161		
P(6)	7		
P(7)	53		6
P(8)	11	5	
P(9)	141	8	

---------INSERT(28)------→

řádek	key	next	begin
P(0)			
P(1)	1	9	1
P(2)			
P(3)	73	7	3
P(4)	28		
P(5)	161		
P(6)	7		
P(7)	53		6
P(8)	11	5	4
P(9)	141	8	

H s linearnim pridavanim

Patří mezi metody, které nepoužívají žádné položky pro práci s tabulkou. To znamená, že způsob nalezení dalšího řádku řetězce je zabudován přímo do metody a určuje hledání volných řádků. Je-li misto zaplnene, posouvam se cyklicky o nejake k (může být i 1), ktere je nesoudelne s m, dokud nenajdu prazdne policko. Pri vyssim zaplneni tabulky se takhle uz posouvam hodne dlouho. Nevhodne pro >60% zaplneni. **Nelze mazat** (max. oznacovat jako smazany), protoze bych nenasel ty prvky, ktere se nachazeji "za" smazanym v cyklickem pruchodu.

Priklad

Množina je uložena v tabulce vlevo. Tabulka vznikla posloupností operací: INSERT(1), INSERT(11), INSERT(73), INSERT(141), INSERT(161), INSERT(53), INSERT(7). Tabulka vpravo vznikne z tabulky vlevo provedením INSERT(35).

řádek	key
P(0)	
P(1)	1
P(2)	11
P(3)	73
P(4)	141
P(5)	161
P(6)	53
P(7)	7
P(8)	
P(9)	

řádek	key
P(0)	
P(1)	1
P(2)	11
P(3)	73
P(4)	141
P(5)	161
P(6)	53
P(7)	7
P(8)	35
P(9)	

H se dvema funkcemi (otevrene H)

Základní nevýhodou předchozí metody je způsob výběru volného řádku. Je stejný pro všechny řádky a důsledkem toho je velmi brzký vznik shluků řádků, který vede brzy k výraznému zpomalení algoritmů. Metoda se snaží tento nedostatek odstranit tím, že výběr dalších řádků je závislý na vkládaném prvku (ideálem je, aby pro různé prvky při kolizi byly posloupnosti adres, na nichž se hledá volný řádek, různé) a je rovnoměrně rozprostřen po celé hašovací tabulce. Nevýhodou metody je fakt, že nepodporuje operaci DELETE.

Obdoba predesleho, jen misto 'k' volime $h_2(x)$. Neni tak nevyhodne, vzhledem k ruznym hodnotam $h_2(x)$, pro jednotliva x. Pracuje dobře pro zaplneni < cca 90 %. Ocekavana delka neuspesneho memberu - O(1/(1-\alpha)), uspesneho O(1/\alpha * ln(1/(1-\alpha))).

Priklad

Hašovací funkce jsou $h_1(x) = x \bmod 10$ a $h_2(x) = 1 + 2(x \bmod 4)$, když $x \bmod 4 \in \{0, 1\}$, $h_2(x) = 3 + 2(x \bmod 4)$, když $x \bmod 4 \in \{2, 3\}$. Množina je uložena v tabulce vlevo, která vznikla posloupností operací: INSERRT(1), INSERT(73), INSERT(53), INSERT(141), INSERT(161), INSERT(11), INSERT(7). Aplikujme operaci INSERT(35). Pak h2(35) = 9, tedy posloupnost možných adres pro uložení x = 35 je (5, 4, 3, 2, 1, 0, 9, 8, 7, 6). Výsledek je uložen v tabulce vpravo:

řádek	key
P(0)	11
P(1)	1
P(2)	
P(3)	73
P(4)	141
P(5)	7
P(6)	53
P(7)	161
P(8)	
P(9)	

řádek	key
P(0)	11
P(1)	1
P(2)	35
P(3)	73
P(4)	141
P(5)	7
P(6)	53
P(7)	161
P(8)	
P(9)	

Srustajici hasovani (Lisch, Eisch)

V tabulke je okrem kluca aj ukazatel jednosměrného spojového seznamu. Na rozdíl od predchozích metod zde nejsou retezce separované, v jednom retezci mohou být prvky s ruznou hodnotou hašovací funkce. Když máme pridat prvek s, tak ho zaradíme do retezce, který se nachází na h(s)-tém rádku tabulky. Retezce tedy v této metode srustají. Ruzné verze této metody se liší tím, kam pridáváme nový prvek a podle práce s pametí. Delí se na standardní srůstající hašování bez pomocné pameti a na hašování používající pomocnou pamet, kterému se ríká jen srustající hašování.

Nejdríve se budeme venovat metodám standardního srustajícího hašování (bez pomocné pameti):

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

Prirozená efektivní operace DELETE pro standardní srustající hašování není známa. Na druhou stranu i primitivní algoritmy mají rozumnou ocekávanou casovou složitost (wtf?). Problem operacie DELETE je ze musí zachovavat nahodnost usporiadania prvkov v retazcoch.

Další otázka zní, proc používat metodu EISCH, když programy pro metodu LISCH jsou jednodušší. Odpoved je na první pohled dost prekvapující. Pri úspěšném vyhledávání je metoda EISCH rychlejší (cca. o 15%) než metoda LISCH. Je to proto, že je o něco pravdepodobnejší, že se bude pracovat s novým prvkem. V neúspešném prípade jsou samozrejme obe metody stejné, nebot retezce jsou u obou stejne dlouhé.

Srustajici H s pomocnou pameti (Lich, Eich, Vich)

Standardní srůstající hašování má tu nevýhodu, že se při větším zaplnění tabulky mohou vytvořit dlouhé řetězce. Tabulku tedy prodloužíme o pomocnou pamět (tzv. sklep), do které se nedostane hašovací funkce, a kolidující prvky přidáváme odspodu.

Metody srustajícího hašování (s pomocnou pametí) mají použitou pamet rozdelenou na dve cásti. Na tu prímo adresovatelnou hašovací funkcí a na pomocnou cást. Adresovací cást má m rádku, pokud hašovací funkce má hodnoty z oboru {0, 1, . . . ,m−1}, v pomocné části jsou rádky ke kterým nemáme prístup pres hašovací funkci. Když pri pridávání nového prvku vznikne kolize, tak se nejprve vybere volný rádek z pomocné části a teprve když je pomocné část zaplnena použijí se k ukládání kolidujících prvku rádky z adresovatelné části tabulky. Tato strategie oddaluje srustání retezcu. Srustající hašování se tedy, aspon dokud není zaplnena pomocná část tabulky, podobá hašování se separovanými retezci. Existují základní tri varianty:

- **LICH** – **late-insertion coalesced hashing** – vkládá prvek na konec retezce,
- **EICH** – **early-insertion coalesced hashing** – vkládá novy prvek x bud do radku h(x), pokud byl prazdny, nebo do retezce vždy hned za prvek na radku h(x)
- **VICH** – **varied-insertion coalesced hashing** – vklada novy prvek x za poslední prvek retezce, který je ještě v pomocne casti, a když řetězec neobsahuje zadny radek z pomocne casti, je novy prvek vlozen do retezce za prvek na radku h(x). T.j. v pomocne casti sa chova jako LICH, v adresove jako EICH.

Shrnutí

Zde uvedeme poradí metod hašování podle ocekávaného poctu testu.
Neúspešné vyhledávání:

1. Hašování s usporádanými separovanými retezci
2. Hašování se separovanými retezci = Hašování s premístováním
3. Hašování se dvema ukazateli
4. VICH = LICH
5. EICH
6. LISCH = EISCH
7. Hašování s lineárním pridáváním

Úspešné vyhledávání

1. H. s usporádanými retezci = H. se separovanými retezci = H. s premístováním
2. Hašování se dvema ukazateli
3. VICH
4. LICH
5. EICH
6. EISCH
7. LISCH
8. Hašování s lineárním pridáváním

Tabulka se obvykle prehasovava, pokud se zaplni nebo zaplneni klesne na ctvrtinu.

Externí hašování

Externí hašování reší trochu jiný problém, než výše popsané metody. Chceme uložit data na externí médium a protože prístup k externím médiím je o nekolik rádu pomalejší, než práce v interní pameti, bude naším cílem minimalizovat pocet prístupu do ní. Externí pamet bývá rozdelena na stránky a ty vetšinou nacítáme do interní pameti celé. Tato operace je však velice pomalá. Problémem externího hašování je tedy nalézt datovou strukturu pro uložení dat na vnejší pameti a algoritmy pro operace INSERT, DELETE a MEMBER, tak abychom použili co nejmenší počet komunikací mezi vnejší a vnitrní pametí.

Metod externího hašování je opet mnoho. Nekteré používají pomocnou datovou strukturu v interní pameti, kterou casto nazýváme adresár. Pokud metody nemají žádnou takovou pomocnou strukturu neobejdou se obvykle bez oblasti pretecení. Nekteré známejší metody vnejšího hašování jsou napríklad: „Litwinovo lineární hašování“, „Faginovo rozširitelné hašování“, „Cormackovo perfektní hašování“ nebo „Perfektní hašování Larsona a Kajli“.

Univerzální hašování

U normalnich hasovacich metod predpokladame rovnomerne rozdeleni vstupni mnoziny. To však nejde zaručit a nerovnomernost a z ni vznikle zpozdeni (casova složitost member/insert/delete) by nam mohlo vadit. Muzeme však ovlivnit vyber hasovaci funkce. A proto požadavek rovnomernosti vstupnich dat prevedeme na požadavek rovnomernosti vyberu z nejake mnoziny nějakých hashovacich funkci. Pro kazdou vstupni mnozinu existuje "hodne" funkci z H, ktere se chovaji dobře (malo kolizi). Následující analýza je platná pro všechny množiny $S \subseteq U$. Je dělána nikoliv přes všechny množiny $S \subseteq U$, ale přes všechny funkce $h \in H$ (tj. množina S je vždy pevně daná, funkce $h \in H$ se volí).

Univerzalni hasovani pouzivame jako rozsireni hasovani se separovanymi retezci. Pokud po insertu zjistime, ze je tabulka plna, tak prehasujem do tabulky dvojnasobne velikosti. Pokud po insertu zjistime, ze je delka retezce delsi nez nejaka konstanta d_i (kterou zatim neumime urcit optimalne), tak prehasujem jinou dostupnou funkci.

Při návrhu formalizace se však objevily technické potíže. Požadavek, aby funkce $h \in H$ byly rychle spočitatelné, vede k tomu, aby byly zadané analyticky. Jak uvidíme dále, formalizace vyžaduje znát velikost H, její stanovení je ale obecně pro takto zadané funkce problematické. Jednoduché řešení tohoto problému je následující. Předpokládáme, že máme množinu indexovaných funkcí a místo s funkcemi budeme pracovat s jejich indexy. Dvě funkce budeme považovat za různé, pokud mají různé indexy (i když jsou to stejné funkce:).

Definice: Soubor hashovacích funkcí H (h:U -> {0, 1, ...m-1}) se nazývá **c-univerzální**, pokud:

$$\forall x \neq y \in U;\ |\{h \in H : h(x) = h(y)\}| \leq \frac{c|H|}{m}$$

Nejprve ukážeme, že c-univerzální systémy existují

Konstrukce H: Předpokládejme: U = (0, 1, ... N-1), kde N je prvočíslo (pro každé přirozené číslo n existuje prvočíslo p, ze n <= p <= 2n, takže přinejhorším natáhnem množinu na dvojnasobek, aby tohle platilo). Definujeme:

$$H = H_{a,b}(x) = ((ax + b)\ mod\ N)\ mod\ m : a, b = 0..N - 1$$

, všechny funkce v U jsou lineární funkce a proto je lze rychle vyčíslit.

Tvrzení: Takto zkonstruované H je c-univerzální systém pro

Důkaz: $|H| = N^2$, což je počet dvojic (a, b).

Nechť (x, y) jsou libovolné, ale pevné. Zároveň platí x y. Kolize nastane v případech, když:

Neboli:

$$ax + b = q - rm \ (mod\ N)$$
$$ay + b = q - sm \ (mod\ N)$$

kde (a, b) jsou neznámé a parametry (q, r, s) nabývají všech hodnot takových, že

$$q \in \{0, \dots, m-1\} \wedge r, s \in \left\{0, \dots, \left\lceil \frac{N}{m} \right\rceil - 1\right\}$$

N je prvočíslo, tedy $\mathbb{Z}_N$ je těleso a pro každou trojici parametrů (q, r, s) má soustava právě jedno řešení (a, b). Počet kolidujících funkcí je přesně tolik, jako počet trojic (q, r, s), který je $m\left\lceil \frac{N}{m} \right\rceil^2$. Lze tedy shrnout, že pro každá x, y ∈ U taková, že x ≠ y, existuje $m\left\lceil \frac{N}{m} \right\rceil^2$ dvojic (a, b) z U × U takových, že $h_{a,b}(x) = h_{a,b}(y)$.

$$\left|\{h_{a,b} : h_{a,b}(x) = h_{a,b}(y)\}\right| \leq m\left\lceil \frac{N}{m} \right\rceil^2 = \frac{\left\lceil \frac{N}{m} \right\rceil^2}{\left(\frac{N}{m}\right)^2} \frac{N^2}{m} = c\frac{|H|}{m}$$

➔ tento univerzální systém lze zkonstruovat. Velikost H je N^2.

Očekávaná délka řetězce

Defince: Mějme libovolnou pevnou S ⊆ U, libovolné pevné x ∈ U a funkci h : U → {0 .. m − 1}. **$S_{h,x}$** = řetězec prvků y ∈ S, pro které platí h(y) = h(x). Zaveďme:

$$\delta_h(x, y) := x \neq y \wedge h_i(x) = h_i(y)$$
$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$$

Chceme spočítat průměrnou délku S_x, kde průměr počítáme přes všechny h ∈ H, kde H je c-univerzální systém.

Věta: Když H je c-universální systém, pak ∀S ⊆ U a ∀x ∈ U je očekávaná hodnota

$$\delta_h(x, S) = \begin{cases} \frac{c(|S| - 1)}{m} & x \in S \\ \frac{c|S|}{m} & x \notin S \end{cases}$$

kde průměr se bere přes h ∈ H a předpokládáme rovnoměrný výběr funkcí z H.

Důkaz: $\sum_{h \in H} \delta_h(x, S) = \sum_{h \in H} \sum_{y \in S, x \neq y} \delta_h(x, y) = \sum_{y \in S, x \neq y} \sum_{h \in H} \delta_h(x, y) = \sum_{y \in S, x \neq y} |\{h \in H; h(x) = h(y)\}| \leq \sum_{y \in S, y \neq x} \frac{c|H|}{m} = \begin{cases} \frac{nc|H|}{m} & x \notin S \\ \frac{(n-1)c|H|}{m} & x \in S \end{cases}$

Věta: Nechť H je c-univerzální systém, pak pro každou množinu $S \subset U,\ |S| = n$ a pro každé x je očekávaný čas operací **insert/member/delete(x)** roven:

$$O\left(\frac{cn}{m}\right)$$

je to průměr přes všechny funkce z H, ne přes všechny vstupy. Takže když to pro jednu funkci není splněno, můžu náhodně vybrat jinou a mám slušnou šanci, že pro ni to, na tom samem vstupu, splněno bude.

Věta: Nechť H je c-univerzální systém, pak S U a x platí, pokud je μ průměrná délka řetězce pro prvky s hodnotou h(x), pak: t >1 plati:
Pravděpodobnost, že řetězec h(x) bude , je menší než –.

Věta: Když H je c-univerzální systém na universu U = {0,1,...N-1} hašující do tabulky velikosti m, pak ———

Reprezentace S a operace MEMBER, INSERT, DELETE.

Je dána základní velikost tabulky m a dále pro j = 0, 1, ... čísla (parametry) d_j a c-univerzální hašovací systémy $H_j = \{h_i \mid i \in I_j\}$ z univerza do tabulky s $2^j m$ řádky. Množina $S \subseteq U$ je reprezentována následovně: je dáno j takové, že když j > 0, pak $2^{j-2}m \leq |S| \leq 2^j m$, když j = 0, pak $|S| \leq m$, a je zvolen index $i \in I_j$. Dále máme prosté řetězce $r_0, r_1, \ldots, r_{2^j m-1}$, jejichž délky jsou nejvýše d_j, a řetězec r_k obsahuje prvky $\{s \in S \mid h_i(s) = k\}$ (jako v hashování s uspořádanými řetězci).

MEMBER(x): Jako v hashování s uspořádanými řetězci.

INSERT(x): Prohledá řetězec $r_k(x)$ ($h_i(x)=k$) a když tento řetězec neobsahuje prvek x, pak ho přidá. Když $2^{j-2}m \leq |S| \leq 2^j m$ a délka řetězce $r_k(x)$ je nejvýše d_j, pak operace končí. Když $|S| > 2^j m$, tak se nejdříve zvětší j o 1. Pak se náhodně zvolí $i \in I_j$ a zkonstruují se řetězce reprezentující S. Když některý z nich má délku větší než dj , tak se volba a konstrukce řetězců opakuje tak dlouho, dokud se nepovede zvolit $i \in I_j$ takové, že všechny zkonstruované řetězce mají délku nejvýše dj.

DELETE(x): se řeší analogicky jako INSERT.

Perfektní hašování

Hasovani takove, kde nejsou kolize. Operace INSERT je bud zakazana, protože přidaný prvek může způsobit kolizi. INSERT lze povolit jen když je málo pravděpodobný. Prvky navíc se dávají jinam a po jisté době se vše přepočítá do jedné tabulky pro novou perfektní hašovací funkci. Typický příklad použití je tabulka klíčových slov kompilátoru.

Chceme pro danou množinu $S \subseteq U$ nalézt funkci h: U → {0, 1, ..m-1} takovou, že:

1) **h je perfektní vůči S** (tj. $\forall s, t \in S, s \neq t: h(s) \neq h(t)$).
2) m je srovnatelné s n ($|S| = n$) - není praktické hašovat do příliš velkých tabulek – ztácí se jeden ze základních důvodů pro použití hašování.
3) h(x) je rychle spočitatelná - jinak hašování není rychlé.
4) h nevyžaduje příliš paměti - uloženi h nesmí vyžadovat moc paměti, nejvýhodnější je analytické zadání (když zadání h bude vyžadovat příliš mnoho paměti, např. když by byla dána tabulkou, pak se ztrácí důvod k použití hašování stejně jako v bodě 2).

Jedna z výhod této metody je, že nalezení perfektní hašovací funkce nevyžaduje velkou rychlost. Na její nalezení můžeme spotřebovat více času, protože se provádí jen na začátku úlohy.

Konstrukce perfektní hašovací funkce

Předpokládejme, že univerzum je tvaru U = {0, 1, …, N − 1}, kde N je prvočíslo. Mějme množinu S ⊆ U o velikosti n. Pro k = 1, 2, …, N – 1 budeme uvažovat funkce:

$$h_k(x) = (k * x \; mod \; N) \; mod \; m$$

Pro i = 0, 1, . . . ,m − 1 a k = 1, 2, …, N − 1 označme:

$$b_k^i = |\{x |\; x \in S; h_k(x) = i\}|$$

Hodnoty b_k^i lze považovat za veličiny, které ukazují odchylku funkce h_k od perfektnosti (udávají počet prvků množiny S, které se zobrazily na i-tý řádek tabulky).

- Pokud $b_k^i \geq 2$, potom $\left(b_k^i\right)^2 - b_k^i \geq 2$
- Naopak $b_k^i \leq 1$, implikuje $\left(b_k^i\right)^2 - b_k^i = 0$

Věta: Funkce h_k je perfektní, právě když $\sum_{i=0}^{m-1}\left(b_k^i\right)^2 < n + 2$.

Důkaz. Když h_k je perfektní funkce, pak $b_i^k \leq 1$ pro každé $i = 0, 1, \ldots, m-1$, a tedy

$$\sum_{i=0}^{m-1} (b_i^k)^2 = \sum_{i=0}^{m-1} b_i^k = n < n + 2.$$

Když h_k není perfektní, pak existuje $i_0 \in \{0, 1, \ldots, m-1\}$ takové, že $b_{i_0}^k \geq 2$. Protože $(b_i^k)^2 \geq b_i^k$ pro každé $i = 0, 1, \ldots, m-1$ a protože $(b_{i_0}^k)^2 \geq b_{i_0}^k + 2$, dostáváme

$$\sum_{i=0}^{m-1} (b_i^k)^2 \geq \sum_{i=0}^{m-1} b_i^k + 2 = n + 2. \qquad \square$$

Věta (bez důkazu): Existuje *k* takové, že:

$$\sum_{i=0}^{m-1} (b_k^i)^2 \leq n + \frac{2n(n-1)}{m}$$

Důsledek

Pro speciální velikosti tabulky dostáváme dosazením do výrazu výše následující:

1. Pro m = n : nalezitelné v čase O(nN) :
2. Pro m = 1 + n(n − 1) : nalezitelné v čase O(nN), že h_k je perfektní, tj.:

.

Důkaz: Probíráme všechny možnosti pro k, těch je N a musíme zahashovat n prvků. Velikost tabulky ve druhém důsledku ale nevyhovuje požadavku 2, protože m= (n^2).

Konstrukce perfektní hašovací funkce pro m = O(n)

Neformálně: Vezmeme funkci z důsledku 1) a rozdělíme S do *m* množin podle $h_k(s) = i$ (množiny kolidujících prvků pro každé *i*). Pro tyto množiny nalezneme perfektní funkci podle důsledku dva. Tahle kombinace dvou funkcí pak tvoří výslednou perfektní funkci. Do <3n se to vleze, protože pro ty malé funkce mám k dispozici druhou mocninu prostoru (vzhledem k počtu prvků). Kolizní řetězce se rozprostřou tak, že se najde perfektní hašovací funkce pro každý z nich, a díky tomu, že těch řetězců je málo, se to celé vedje do 3*n*.

Formálně: Nalezneme k takové, že pro m = n platí . Pro každé i {0, …, n − 1} vezmeme množinu kolidujících prvků $S_i = \{s \quad S : h_k(s) = i\}$. Označme $n_i = |S_i|$. Pro každé i nalezneme k_i takové, že pro $m_i = 1 + n_i(n_i - 1)$ je perfektní pro S_i. Každou zahašovanou množinu S_i uložíme ve výsledné tabulce od pozice d_i:

Definujeme: **$g(x) = d_i +$ (x), kde $i = h_k(x)$**, která je perfektní a velikost tabulky je:

Ovšem na zakódování této funkce potřebujeme hodně paměti: nevadí nám d_i, ale k a každé k_i je velikosti O(N), tedy potřebujeme n $\log_2$ N bitů, což odporuje našemu požadavku (4). V dalších krocích budeme zmenšovat čísla definující hašovací funkci.

Možnosti dynamizace jednotlivých datových struktur

V uspořádaném poli umíme rychle vyhledávat, ale přidat prvky znamená celé ho přebudovat. Ve srůstajícím hašování zase nešly prvky mazat, ve velmi komprimovaných trie ani přidávat, ani mazat. Ukážeme obecnou metodu, jak tyto problémy řešit, podobným přístupem jako u binomiálních hald.

Máme obecnou statickou strukturu (neumožňující INSERT a DELETE) a zkoumáme jak do ní toto přidat. Chceme, aby se vyhledávalo v nové struktuře přibližně stejně rychle jako v té původní. Dále, když vytvoření původní struktury pro n-prvkovou množinu trvalo t, pak operace INSERT by měla vyžadovat čas t/n.

Zobecněný vyhledávací problém

Je funkce $f : U_1 \times 2^{U_2} \to U_3$, kde U_1, U_2 a U_3 jsou univerza.

Rešení zobecněného vyhledávacího problému

Pro $x \in U_1, A \subseteq U_2$ je nalezení hodnoty f(x, A) ∈ U_3.

- Položme $U_1 = U_2 = U$, $U_3 = \{0, 1\}$, f(x,A) = 1, když x ∈ A, a f(x, A) = 0, když x A. Pak operace MEMBER je vyhledávací problém.
- Nechť $U_1 = U_2$ = Euklidovský prostor, U_3 = množina nezáporných reálných čísel, f(x, A) je vzdálenost x od A. Pak určení vzdálenosti v Euklidovkských prostorech je vyhledávací problém.

Rozložitelnost vyhledavacího problému

Rekneme, ze **vyhledavaci problem je rozlozitelny**, kdyz existuje **binarni operace** $\oplus$ (spočitatelná v konstantním čase O(1)) na univerzu U_3 takova, ze pro disjunktni mnoziny A,B ⊆ U_2 a pro x ∈ U_1 plati: **f(x, A)** $\oplus$ **f(x, B) = f(x, A ∪ B)**

Definice: Řekneme, že **S je statická datová struktura** řešící vyhledávací problém f, když je dán algoritmus, který pro množinu A ⊆ U_2 zkonstruuje datovou strukturu S(A) a algoritmus, který pro x ∈ U_1 a S(A) vyčíslí f(x, A).

Definice: Řekneme, že **S je semidynamická datová struktura**, když navíc existuje algoritmus, který pro x ∈ U_2 a pro S(A) zkonstruuje S-strukturu S(A∪ {x}) a algoritmus vyčíslující f(y, A ∪ {x}) pro každé y ∈ U_1. Tato operace se nazývá **INSERT(x)**.

Definice: Řekneme, že **S je dynamická datová struktura**, když ještě navíc existuje algoritmus, který pro x ∈ U_2 a pro S(A) zkonstruuje S-strukturu S(A \ {x}) a algoritmus vyčíslující f(y, A \ {x}) pro každé y ∈ U_1. Tato operace se nazývá **DELETE(x)**.

Naším cílem je nalézt metody, které změní statickou strukturu řešící vyhledávací problém f na semidynamickou nebo dynamickou strukturu řešící f.

Nechť f je rozložitelný vyhledávací problém a S je statická datová struktura, která ho řeší. S tedy je tvořena pevnou množinou A U_2, |A| = n a obsahuje operaci, která pro vstup x počítá f(x,A). Potom označme:

- **$Q_S(n)$** = čas potřebný pro výpočet f(x, A)
- **$S_S(n)$** = paměť potřebná pro vybudování S
- **$P_S(n)$** = čas potřebný pro vybudování S

Semidynamizace

Věta: Máme rozložitelný vyhledávací problém f a máme pro něj statickou strukturu, která ho řeší v čase Q(n), vyžaduje S(n) paméti a vytvoří se v čase P(n), kde Q(n), P(n)/n , S(n)/n jsou neklesající funkce. Pak existuje semidynamická datová struktura D, řešící f v čase $Q_S(n)$ O(Q(n) · log n) vyžadující $S_S(n)$ O(S(n)) paměti a umožňující INSERT s amortizovanou složitostí $P_S(n)$ O(P(n)/n · log n).

Konstrukce semidynamicke struktury

Máme množinu A a vytvoříme pro ni novou strukturu D. Nechť jsou A_i A takové, že bud' $|A_i| = 2^i$ nebo A_i = , A_i A_j = pro i j, = A. Platí, že A_i právě když (i + 1)-ní bit v dvojkovém rozvoji čísla |A| je 1.

Nová struktura D reprezentující A je potom:

- Nějaká dynamická struktura T reprezentující A (např. (a,b)-strom, červeno-černý strom, AVL strom).
- Pro každé A_i máme S strukturu reprezentující A_i.
- Pro každé A_i seznam prvků v A_i. Prvky téchto seznamů jsou projpojeny s odpovídajícími prvky ve stromě.

MEMBER(x, A) = **f(x, A)** = Prejdi vsetky A_i (f(x, A_i)) a jednotlive vysledky spoj pomocou $\oplus$ do výsledného f(x, A). Platí, že když A_i , pak i .

Časová složitost, která je potřebná pro MEMBER v nové struktuře:

$$log_2|A| + \sum_{i=0}^{log|A|} Q(2^i) \leq log|A| + \sum_{i=0}^{log|A|} Q(|A|) = log|A|(Q(|A|) + 1)$$

INSERT(x, A)
if x T **then**
nalezneme nejmenší j takové, že A_j =
$A_j = \{x\}$ pro i < j. A_i = pro i < j.
vytvoříme strukturu S a spojový seznam pro A_j
x přidáme do T.
end if

Slozitost pro INSERT: Kdy se buduje znovu (tedy podruhé) S struktura pro Aj (měřeno počtem INSERTů)?

- Musí se naplnit všechny A_i pro i < j. To je $2^j - 1$ INSERTů.
- Provede se úspěšný INSERT, který vyprázdní A_i pro i j
- Znovu se musí naplnit A_i tj. $2^j - 1$ úspěšných INSERTů
- Další úspešný INSERT vytvoří teprve S strukturu pro Aj tj. $2^j - 1 + 1 + 2^j - 1 + 1 = 2 \cdot 2j = 2^{j+1}$ úspěšných INSERTů.

Amortizovaný čas operace INSERT je teda:

$$log|A| + \sum_{i=0}^{log|A|} \frac{P(2^j)}{2^{j+1}} \leq log|A| + \sum_{i=0}^{log|A|} \frac{P|A|}{|A|} = O(log|A| \cdot \frac{P|A|}{|A|})$$

Semidynamická struktura s rychlým insertem

Protože amortizovaná složitost není vždy uspokojující (odhad amortizované složitosti je vhodný pro dávkové zpracování požadavků, zatímco pro interaktivní zpracování je důležitý odhad v nejhorším případě), hledaly se další metody, které by vedly k algoritmu pro operaci INSERT, který by měl dobrý odhad složitosti v nejhorším případě.

Základní idea je rozložit budování S-struktury pro množinu velikosti 2^i do 2^i po sobě jdoucích INSERTů. Problém je, že S-struktury musí pokrývat celou množinu A, jinak nejsme schopni vyřešit vyhledávací problém f. Řešení spočívá v tom, že budeme mít více množin velikosti 2^i, pro které je vybudovaná S-struktura. Pak, když máme vybudované S-struktury pro dvě množiny $A_{i,0}$ a $A_{i,1}$ velikosti 2^i, zažneme budovat S-strukturu pro množinu $A_{i,0} \cup A_{i,1}$ velikosti 2^{i+1} a teprve až tuto S-strukturu vytvoříme, zrušíme S-struktury pro množiny $A_{i,0}$ a $A_{i,1}$.

Konstrukce množiny pro rychlý INSERT

Máme množinu A budeme mít rozklad A na disjunktní množiny $A_{i,}j$; i = 0, 1,... j ∈ 0, 1, ..., kj , kde kj ∈ 0, 1, 2. $|A_{i,}j| = 2^i$ a platí: když $A_{i,0}$ existuje pro i > 0, pak existují $A_{i-1,0}$,$A_{i-1,1}$

Struktura D:

- reprezentace A (pomocí (a,b)-stromů, červeno-černých stromů, ...)
- ∀ existující $A_{i,}j$ je S struktura reprezentující $A_{i,}j$
- ∀ existující $A_{i,}j$ je spojový seznam reprezentující $A_{i,}j$
- když $A_{i,0}$ a $A_{i,1}$ existují pro nějaké i, pak je ”rozpracovaná” S struktura pro množinu $A_{i-1,ki+1} = A_{i,\ 0} \cup A_{i,1}$. tj. bylo provedeno několik kroků pro její vytvoření, ale není dokončena.

$A \subseteq U_2$, $i_0 \in N$
∀i = 0, 1, ..., i_0 je dáno $j_i \in 0, 1, 2$ takové, že $j_i > 0$ když $i < i_0$.
∀i = 0, 1, ..., i_0 a ∀j = 0, 1, ..., j_i je $A_{i,}j \in A$ taková, že $|A_{i,}j| < 2^i$

Definice: $A_{i,}j$, i = 0, 1, ..., i_0, j = 1, 2, ..., j_i je rozklad A.

Pro každé $A_{i,}j$ je dána S struktura reprezentující $A_{i,}j$ a spojový seznam prvků z $A_{i,}j$, navíc dána datová struktura reprezentující A. Když $A_{i,}1$ existuje, pak je rozpracovaná S struktura pro $A_{i+1,ji+1+1} = A_{1,0} \cup A_{i,1}$.

Dynamizace

Ještě přidáme operaci **falešný DELETE** (implementovana pomocou oznacenia prvku za smazany - **Oznaceni(x)**). Kdyz aplikujeme Oznaceni(x) na strukturu S(A), nastane jedna z nasledujicich dvou moznosti:

- $x \notin A$ a pak se struktura S(A) nezmeni
- $x \in A$ a pak se vytvori struktura S‘(A), ktera vyzaduje stejnou pamet jako S(A), a algoritmus, ktery pro vstup $y \in U_1$ spocita $f(y, A \setminus \{x\})$ (misto f(y, A)) ve stejnem case jako algoritmus pro strukturu S(A) a vstup y.

Pro jednodussi komunikaci oznacime $B = A \setminus \{x_i \mid i = 1, 2, \ldots, k\}$ a S‘(B) vzniklou strukturu a rekneme, ze S(A) je puvodcem S‘(B). Pak S‘(B) vyzaduje $S_S(|A|)$ pameti, pro $y \in U_1$ vypocteme f(y, B) v case $Q_S(|A|)$ a pro $x \in U_2$ zkonstruujeme strukturu S‘(B \ {x}) v case $D_S(|A|)$. Budeme predpokladat, ze funkce D_S je neklesajici.

Postup pro vytvoreni dynamicke struktury

Mejme danu mnozinu $A \subseteq U_2$. Struktura P(A) se sklada ze **souboru struktur** $\{ S‘(A_i) \mid i = 0, 1, \ldots, |\log |A|| + 3\}$ a **souboru spojovych seznamu** $\{s(A_i) \mid i = 0, 1, \ldots, |\log |A|| + 3\}$, kde $A_i \cap A_j = \emptyset$ pro ruzna i, j a sjednoceni všech $A_i = A$. $A_i = \emptyset$ nebo $2^{i-3} < |A_i| \leq 2^i$ pro kazde i. Kdyz $A_i \neq \emptyset$, pak $S‘(A_i)$ je struktura pro mnozinu A_i, jejim puvodcem je $S(B_i)$, kde $|B_i| \leq 2^i$, a $s(A_i)$ je prosty seznam prvku z A_i, kde u kazdeho prvku je odkaz na jmeno mnoziny, ve ktere lezi, a navic je u kazdeho seznamu $s(A_i)$ uvedena velikost mnoziny A_i. Dale je dana datova struktura T reprezentujici mnozinu A a pro každý prvek $a \in A$ jsou dany ukazatele spojujici prvek a v T s prvkem a v seznamu A_i.

MEMBER(x) = to iste jako semidynamicke.

INSERT(x)
if $x \notin A$ **then**
vlož x do T;
nalezneme nejmenší j takové, že $\left|\bigcup_{i \leq j} A_i\right| < 2^j$;
položíme $A_j = \bigcup_{i \leq j} A_i \cup x$;
$A_i = \emptyset$ pro $i < j$;
Vytvoříme S-strukturu a spojový seznam pro Aj (x přidáme do struktury reprezentující A a přidáme požadované ukazatele);
end if

DELETE(x) = sialene.

Složitost

Amortizovaně INSERT v $O(\log(n) * P_S(n)/n)$, DELETE v $O(D_S(n) + \log n + P_S(n)/n))$, kde D(n) je cas na skonstruovani S(A – {x}). Pokud P(n) je n^ε, tak INSERT vychází O(P(n)/n). DELETE naplní A_j jen do půlky, takže to sice stačí, ale nekazí to složitost INSERTu získanou v semidynamické kapitole.

Mapování datových struktur do stránek vnější paměti počítače, casová složitost algoritmů vyjádřená v počtu I/O operací

Soubor - identifikovaná kolekce (ne mnozina) logicky souvisejících záznamů.

- **Homogenni soubor** – obsahuje pouze záznamy jediného typu
- **Nehomogenní soubor** - **opak homogenního souboru** – obsahuje buď záznamy různých typů, záznamy se složitější strukturou, nebo záznamy různých délek.

Operace nad soubory

- **INSERT** - vložení záznamu do souboru
- **DELETE** - rušení záznamu v souboru
- **UPDATE** - změna některých položek záznamu v souboru
- **FETCH** – ziskani zaznamu ze souboru do vnitrni pameti.
- **Aktualizace souboru** - posloupnost operací INSERT, DELETE a UPDATE.

Logický záznam – množina atributů.

Fyzický záznam (délky **R**) – fyzická reprezentace logického záznamu na médiu. Může obsahovat pomocná data navíc (oddělovače, příznaky…). Fyzické záznamy se ukládají do ***fyzických stránek*** (***bloků*** – jejich velikost značíme **B**). Delka stranek muze byt fixovana (napr. celistvy násobek sektoru) nebo ji lze formátovat softwarem.

Blokovací faktor (**b**) – poměr B/R.
Blokování (umísťování záznamů do bloků); říkáme, že záznamy jsou:

- ***Neblokované*** (b = 1, B = R)
- ***Blokované*** (b > 1)
- ***Přerostlé*** (b < 1)

Schéma organizace souborů (SOS) - **popis logické paměťové struktury**, do níž může být soubor zobrazen spolu s algoritmy provádějícími operace nad touto strukturou. SOS muze obsahovat vice logickych souboru. **Primární soubor** – soubor obsahující uživatelská data; **N** – počet záznamů v něm.

Dotazy nad soubory

Dotaz na úplnou shodu (zadány hodnoty všech atributů), na **částečnou shodu** (zadány hodnoty jen některých atributů), **dotaz na úplnou intervalovou shodu** (zadán interval hodnot pro každý z atributů), **dotaz na částečnou intervalovou shodu** (zadány intervaly pro vybrané atributy). Vyváženost struktury (rozlišují se podle ní *statické* a *dynamické* SOS):

- **Omezení délky cesty** - ve struktuře při vyhledání záznamu (např. O(log M), kde M je počet logických stránek). -> **Staticke metody**.
- **Rovnoměrná naplněnost stránek** - faktor naplnění stránky λ ($0 \leq \alpha < \lambda < \beta \leq 1$). Někdy se λ označuje průměrné naplnění stránek. -> metody splnujici omezeni delky cesty + rovnomernu naplněnost = **dynamicke metody**.

Vnejsi pameti

- **Magentická páska** - Sekvenční čtení, využívá se buffer (velikosti jedné stránky), páska obsahuje meziblokové mezery.
- **Magnetický disk** – ***Stopy*** (soustredne kruznice), ***cylindr*** (nebo valec se sklada ze všech stop stejného prumeru), ***sektory*** (stopa rozdelena na bloky stejne velikosti). Parametry: ***s* – seek** (prumerny cas presunu raminka z jedneho cylindru na jinej), ***r* - rotation** (rovno polovině doby otáčky disku = prumerny cas), ***btt* - block transfer time** (cas potřebný na presun bloku z disku do vnitrni pameti nebo naopak).

Počty I/O operací - základní přehled

- **FETCH** - Cas pro načtení jedné stránky $T_F = s + r + btt$, pro další stránky na stejném cylindru se nemusí hlavičky přesouvat ($T_F = r + btt$).
- **REWRITE** - Přepsání stránky na disku (tj. přenos stránky z disku do vnitrni paměti, změnu ve vnitrni paměti a přenos zpět na disk), lze vyřešit v době, než se plotna disku jednou otočí, tedy $T_RW = 2r - btt + btt = 2r$.
- **Insert (T_I)** - Vložení zánamu (je nejprve zapotřebí načíst stránku, do které se bude zapisovat, do paměti)
- **Delete (T_D)** - Smazání stráky (obvykle prohlášení za neplatnou).
- **Update (T_U)** - Změna záznamu ve stránce.
- **Reorganizace (T_Y)** - Pro jednotlivé typy organizace souborů bude úvaha nad jejich počty IO operací uvedena u jejich popisu.

Statické metody organizace souborů

Hromada

Nehomogenní soubor. Záznamy proměnné délky (záznamy proměnné struktury – uvádí se pak i jména či kódy atributů). Sekvenční uložení ve vymezeném adresovém prostoru. Složitost vyhledání záznamu O(N).

Neuspořádaný sekvenční soubor - oproti hromadě záznamy pevné délky.

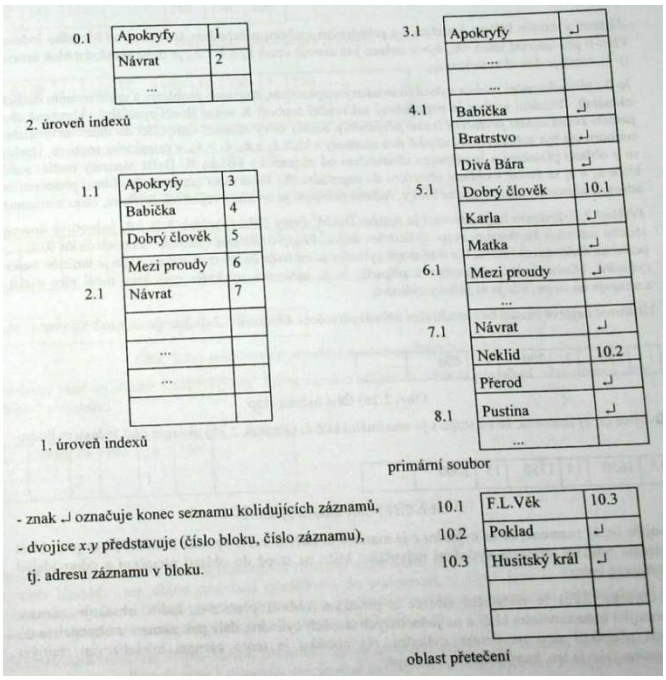
Uspořádaný sekvenční soubor – usporadany podle vyhledávacího klíče. Aktualizovane zaznamy se umistuji do zvlastniho neusporadaneho sekvencniho souboru zvaného **soubor aktualizaci (soubor transakci),** kteri se po provedeni operace REORGANIZATION setridi a zmerguje se s puvodnnim souborem. Vyhodou je, ze v setridenem usporadanem soubore se muze vyhledavat pulenim intervalu.

Odhady pro uspořádaný sekvenční soubor:

- $T_F = log_2(N/\lfloor b \rfloor)(s + r + btt)$ - nalezení záznamu metodou půlení intervalů.
- $T_I = s + r + btt + T_R W + T_Y / o$ - načte se stránka kam se bude vkládat, přepíše se a připočte se poměrný čas potřebný pro reorganizaci souboru (až k ní dojde).
- Update $T_U = T_F + 2r$ nemění-li se klíč, $T_U \approx T_I$ mění-li se klíč.

Indexsekvencni soubor

Je tvoren primárním souborem (**usporadany sekvencni soubor** podle primárního klice K – v podstate 0-ta uroven indexu), viceurovnovou strukturou zvanou **index** (nejvyssi uroven (k-ta) = jeden zaznam je vždy v pameti tzv. **master index**) a **oblasti přetečeni** (na pridavanie novych zaznamov). Záznam v k-té úrovni obsahuje největší (nejmenší) klíč z bloku na k-1 úrovni.



Počet urovni indexu

Chceme, aby počet urovni indexu byla co nejmensi (2,3,4) protože počet urovni predstavuje počet pristupu na disk při kazdem pristupu na data v primárním souboru.

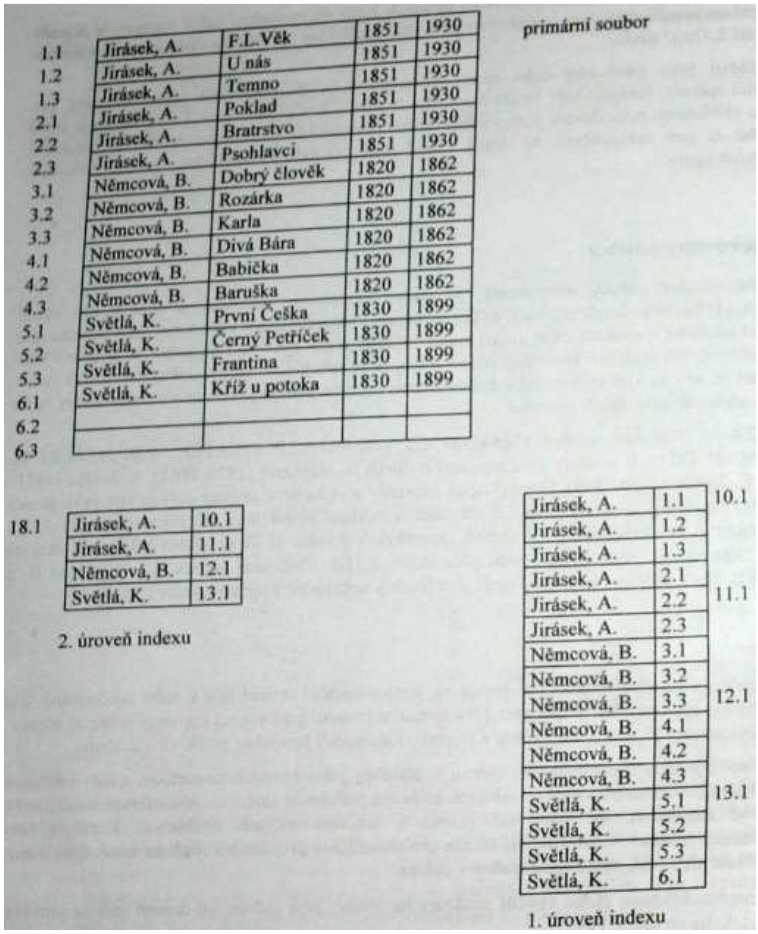
k = , kde p = , V je prumerna velikost klice, P je velikost ukazatele na nizsi uroven.

Oblast přetečeni

Vkladaji se tam zaznamy, které se nevejdou do původního primárního souboru. Je proto vhodne aby každý blok dat primárního souboru mnel nejakou rezervu, aby se nemuselo hned vsechno cpat do oblasti přetečeni.

Indexovany soubor

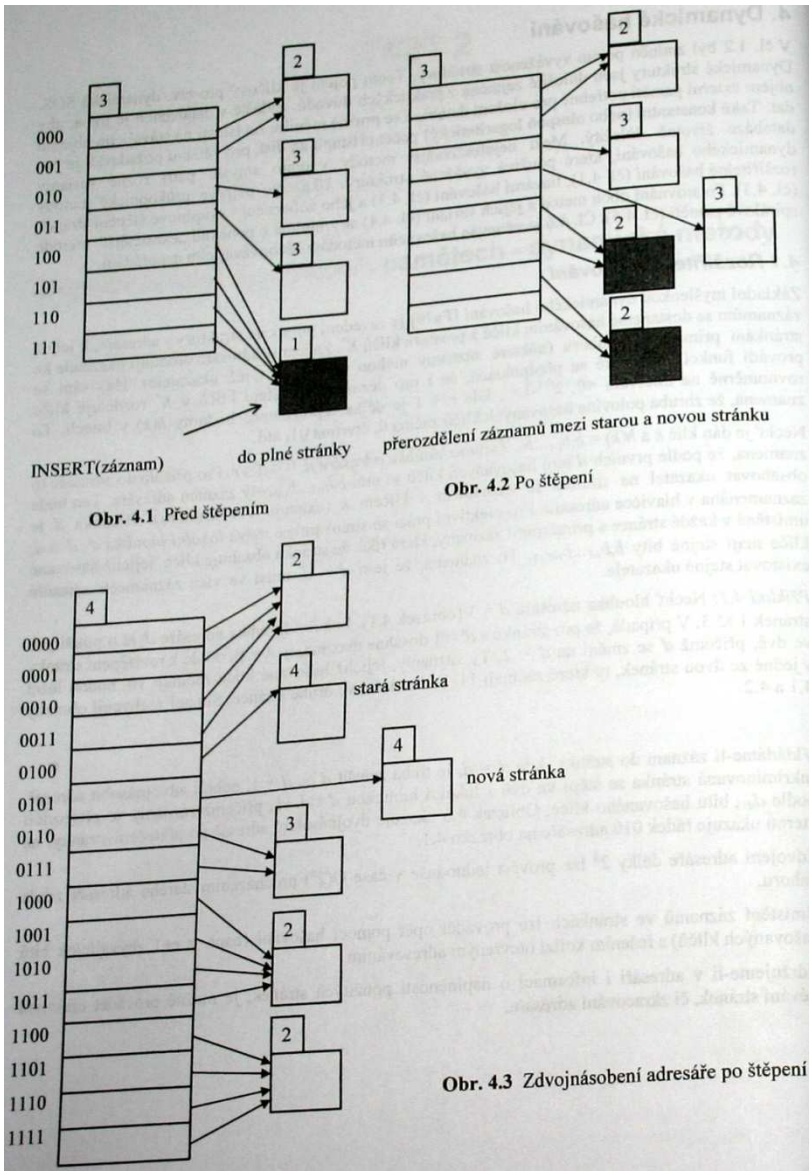
Někdy nazývaný invertovaný soubor. Sklada se z **primárního souboru** a z **indexů pro různé vyhledávací klíče** (ale staci klidne i jeden ak chceme vyhledavat iba podle jedneho klice). **Indexují se záznamy**, ne bloky z ceho vyplyva ze primární soubor **nemusí být setříděn**. V indexu se tentýž klíč může opakovat a taky mohou/nemusí být setříděné. Počet urovni indexu stejne jako u index sekvenčního.



Dynamické metody organizace souborů

Rozsiritelne hashovani

K lokalizaci stránek s daty využívá adresář. Připouští jak přidávání, tak odebírání záznamů. K získání dat stačí 1-2 přístupy na disk. Neumí se vypořádat se situací, kdy pro příliš mnoho různých klíčů je hašovaná hodnota totožná. Adresář bývá možné držet ve vnitřní paměti (pak je jen 1 přístup na disk).



Obr. 4.1 Před štěpením

Obr. 4.2 Po štěpení

Obr. 4.3 Zdvojnásobení adresáře po štěpení

Linearni hashovani

Stránky je možné štěpit po pevném počtu vložení záznamů. Předpokládá se hašovací funkce h(K), která pro libovolný klíč vrátí binární celé číslo v dostatečném rozsahu.

Skupinove stepeni stranek

Stránky štěpeny po určitém počtu přidaných záznamů do souboru. Snaha o rovnoměrnější rozdělení záznamů do stránek než u lineárního hašování. Soubor dělený do *s* skupin po *g* stránkách. Skupiny se při štěpení přerozdělují do *g*+1 stránek. Lineární (Litwinovo) hašování je skupinovým štěpením stránek pro *g*=1.

Třídění ve vnitřní a vnější paměti

Formulace ulohy:

Necht U je totalne usporadane univerzum.

Vstup: Prosta posloupnost {a1, a2, …, an} prvku z U.

Vystup: Rostouci posloupnost {b1, b2, …, bn} takova, ze {ai | i=1, 2, …, n} = {bi | i = 1, 2, …, n}.

Vnejsi pamet

Slevani

Pomoci slevani získáme z několika setridenych poslopnosti zaznamu jedinou setridenou poslopnost. (Pouzitelne ak mam k dispozicii aspon 3 subory: z dvoch citam behy a do tretieho zapisujem vysledky zlievania, potom subor s vašimi zliatymi behmi rozdělím do dvoch vstupnych a proces opakujem. Ak mam k dispozicii aspon 4 subory, tak mozem rovno na striedacku zapisovat zliate behy z dvoch vstupnych do dvoch výstupných suborov, tie sa v dalsom kole prehodia (vstpne na vystupne a naopak).

Trídení haldou

Pro třídení ve vnitřní pameti (ale i modifikace - třídění dvojitou haldou - se používá jako pomocná rutina při vnějším třídění – merge sort) se používá algoritmus trídení haldou (heapsort), který se dá zakomponovat do vylepšení trídení sléváním (viz níže). Jeho základem je datová struktura halda (konkrétne maximální halda, max-heap), reprezentovaná jako pole záznamu, na kterém je binární stromová struktura: záznam k má vždy vyšší klíc než jeho dva synové, nacházející se na pozicích 2k+1 a 2k+2 pri císlování od 0 (pokud tato pozice není vetší než velikost haldy, v opacném prípade záznam nemá syny). Na pozici 0 se tak nachází záznam s nejvyšším klícem. Casová složitost algoritmu je
O(n log n) pro pole záznamu velikosti n. Postup trídení je následovný:

- nejvetší prvek (z pozice 0) se prohodí s tím prvkem, jehož císlo pozice odpovídá aktuální velikosti haldy
- velikost haldy se zmenší o 1
- dokud neplatí podmínka, že klíc prvku získaného z konce haldy je vetší ne oba klíce jeho synu, prohazuje se tento se synem s vetším klícem (a tak posouvá v halde dál)
- toto se opakuje, dokud je velikost haldy vetší než 1, odzadu tak v poli vzniká setrídená posloupnost

Merge sort

- na zacátku každého kroku data rozdelí do 2 souboru
- nacte 2 záznamy, každý z jednoho souboru a porovná je
- ve správném poradí je zapíše do výstupního souboru, ze vstupních souboru si nacte další dva

- v prvním kroku získám usporádané posloupnosti délky 2; v dalších krocích vždy porovnám nactené prvky, zapíšu menší z nich a ze souboru odkud tento pocházel si nactu další, takže získám vždy usporádané posloupnosti dvojnásobné délky než v predchozím kroku
- po $\lceil \log n \rceil$ krocích je soubor s n záznamy setrídený.

Vylepšení se dosáhne napr. prímo strídavým zapisováním výstupu do 2 souboru, kdy se zbavím nutnosti na zacátku každého kroku data delit, nebo použitím více souboru najednou. Je taky možné využít rostoucích posloupností prvku, které se v souboru nacházejí již pred zapocetím trídení.

N-cestny algoritmus (Merge sort)

Pouziva sa pro trideni dat (ulozenych v souboru na sekundarni pameti), které se nevejdou do interni pameti. Idea: zaklad jako v trideni slevanim – slevaji se behy dokud nezískáme jediny beh. Na vytváření pocatecnich behu muzme pouzite napr. trideni haldou. K dispozici mame vyrovnavaci pamet o velikosti n+1 stranek:

while !end(file) **do begin**

nacti do pameti n stranek

setrid obsah n stranek (jednu volnu stranku vyuzi jako buffer pre zapisovanie)

uloz vysledek jako samostatni beh

end

while existuje vice nez jeden beh **do begin**

vezmi n nejkratších behu a slij je do jednoho behu

(opat pouzivaj jednu volnu stranku jako buffer)

end

Algoritmus garantuje, ze data budou 1x ctena a zapisovana pro vtvoreni pocatecnych behu a $\log_n m$ krat (kde m je počet behu) přečtena a zapsana při slevani.

Behy pro trideni (resp. externi trideni) lze delat pomoci **2 hald**. Udelam jednu pres celou pamet. Trham minimum a novy nacitany prvek: pokud je vetsi nez odeslane min tak zatridim do haldy, jinak ulozim do pameti na volne misto (stavim 2. haldu).

Dvojitá halda

Delší behy pri slévání se dají vytváret pomocí dvojité haldy – v pameti mám dve haldy z celkem n prvku, opakovane z první haldy odebírám a zapisuji minimální prvek do výstupního behu a nacítám další prvek, pokud ten je vetší než minimum haldy, vložím ho do prvé haldy, pokud je menší, vložím ho do druhé haldy, která vzniká od konce mého pole. Až se první halda vycerpá, použiji druhou a zacnu nový beh. Toto v nejhorším prípade dává stejnou velikost behu jako obyčejná halda, prumerne je 2x lepší.

Vnitrni pamet - **Porovnavaci**

Buble sort - prochazim seznamem a kdyz najdu dva sousedni prvky, ktere jsou v opacnem poradi nez maji byt, tak je prehodim. Pruchody koncim v dobe, kdyz jsem pri pruchodu nic nezmenil - je setrizeno. O(n^2), nejhorší.

Trideni slevanim – viz. hore slevani sa da pouzit aj v pamati (bez pouzitia suborov).

Selection sort – tridi poslopnost tak, ze jednym pruchodem nalezne jeji nejmensi prvek, který vyradi a vlozi do vysledne posloupnosti (ve verzi která tridi na miste , ho vymeni s levym krajnim prvkem pole). Tento proces pak opakuje se zbytkem puvodni posloupnosti. O(n^2) nejhorší i nejlepší.

Insertion sort – tridi tak ze do jiz setrideneho zacatku posloupnosti vklada další prvek, který pomoci vymen zaradi na spravne místo a tento proces (zacina druhym prvkem zleva) opakuje. O(n^2) nejhorší i nejlepší.

Shell sort - zdokonaleny insertion sort. Pri zatrizovani neporovnavam se sousedem, ale s nekym o vice indexu vzdalenym - treba 5. Pote, co skoncim, tak opakuju se skakanim o 3 a naposledy skacu o 1 (=insert sort). Pri dobre k - vzdalenosti dvou porovnavanych prvku, lze slozitost stlacit (v soucasnosti, zlepseni je otevreny problem) az na O(n log^2(n))

Merge sort - rozdelim posloupnost na pulky rekurzivne az na jednotlive prvky. Pak skladam vzdy dve setrizene casti (pocinaje dvema samostatnymi prvky a konce polovinami posloupnosti) - O(n logn) best, worst i avg.

Quicksort - je jedním z nejrychlejších algoritmu pro trídení na vnitrní pameti, prestože v nejhorším prípade muže jeho casová složitost dosáhnout až _(n2). Pro ideální i prumerná data dosahuje _(n log n). Je také založen na principu „rozdel a panuj“, i když ponekud jiným zpusobem než predchozí zminovaný, od nehož se liší i tím, že není stabilní. Algoritmus nejdrív vybere nejaký prvek, tzv. pivot, a prvky s klícem vetší než pivot presune do jiné casti pole než ty s klícem menším. Pak rekurzivne trídí obe casti pole – když se dostane k polím délky 1, problém je vyrešen. Postup vypadá takto:

1. Vyber pivot (jeden prvek ze seznamu). Tady jde o nejvetší magii, protože k dosažení nejlepší rychlosti by se mel pokaždé vybírat medián. Nejjednodušší je vybrat první, ale tento výber ovlivnuje výslednou rychlost práce, takže se vyplatí napr vzít tri prvky, porovnat je a vzít si z nich ten prostrední.
2. Postupuj od zacátku pole a hledej první prvek vetší nebo rovný než pivot. Až ho najdeš, postupuj od konce a najdi první prvek menší než pivot.
3. Prvky prohod a opakuj krok 2 a 3, dokud se hledání od zacátku a od konce nepotká na nejaké pozici – tu pojmenujeme treba k.
4. Rekurzivním voláním setrid prvky (0, . . . , k) a (k+1, . . . , n−1) (má-li trídené pole délku n) – to znamená pro obe casti pole pokracuj od kroku 1. Pokud je k = 0 nebo k = n−2, není treba už rekurzivního volání, protože posloupnosti délky 1 jsou setrídené.

Pro algoritmus existuje i nerekurzivní verze (stací rekurzi nahradit zásobníkem úseku cekajících na zpracování). Je videt, že na volbe pivotu závisí všechno – pokud pokaždé jako pivot volím 1. nebo n − 1. hodnotu v poli v poradí podle velikosti, delím pak vždy na casti o délce 1 a n−1, takže tento rekurzivní krok provedu až nkrát a dostanu se k casu $\theta(n^2)$. Samozrejme, díky existenci algoritmu pro nalezení mediánu v case $\theta(n)$ je možné i tady dosáhnout zarucené složitosti $\theta(n \log n)$, ale v

praxi je to kvuli vysoké multiplikativní konstante nepoužitelné – k výberu pivotu se vetšinou s úspechem užívá nejaká jednoduchá heuristika, jak je nastíneno v popisu algoritmu samotného.

Heapsort bývá pomalejší než quicksort, ale zarucuje nízkou casovou složitost i pro nejhorší prípad a navíc potrebuje méne pameti – nároky quicksortu navíc (krome trídené posloupnosti) jsou O(log n) minimálne, kvuli nutnosti použití rekurzivního volání nebo zásobníku. Oproti mergesortu ho nelze použít na data se sekvencním prístupem, tyto nevýhody ale vyvažuje relativní jednoduchostí implementace a rychlostí v prumerném prípade. Variantou quicksortu je introsort, který ho kombinuje s heapsortem, pokud hloubka rekurze dosáhne nejakých neprijatelných hodnot – tak je zarucena casová složitost _(n log n) i v nejhorším prípade (samozrejme je to ale v nejhorším prípade porád pomalejší než použití jen heapsortu). Jedna z variant tohoto algoritmu se dá použít k hledání k-tého nejmenšího prvku (tedy i mediánu), kdy dosahuje složitosti O(n) prumerne až $O(n^2)$ nejhure.

Heap sort - postavim haldu (jde O(n)), n krat extractmin (prip. extractmax) - O(n log(n)) celkem.

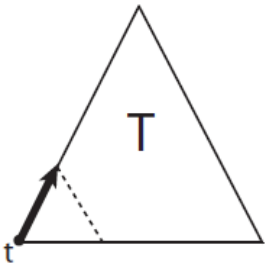
A-Sort - Na první pohled se zdá, že použití (a, b) stromů ke třídění není výhodné. Paměťové nároky budou oproti běžnému třídění v poli asi pětkrát větší. Aby se tedy třídění (a, b) stromem vyplatilo, muselo by přinést zvýšení rychlosti. Je to možné, jestliže vstupní data jsou již částečně setříděná. Pro účely A-sortu rozšíříme reprezentaci takto:

- Listy stromu jsou propojeny do seznamu
- Je známa cesta z nejmenšího (nejlevějšího) listu do kořene (uložená např. v zásobníku)

Použijeme (2, 3)-strom. Proč, to si zdůvodníme až po odvození složitosti A-sortu. Nechť vstupní posloupnost je $a_1, \ldots, a_n$. Postupně odzadu vkládáme její prvky do stromu modifikovaným INSERTem. Na konci přečteme setříděnou posloupnost pomocí spojového seznamu listů.

INSERT(a = $a_1, \ldots, a_n$**)**:
k := n;
while k > 1 **do**
    A-INSERT(a_k);
    k := k – 1;
end while

A-INSERT pracuje téměř stejně jako původní INSERT - najde správný list a potom případně přidá nový prvek. K nalezení správného listu ovšem využívá cestu z nejmenšího listu.



A-INSERT(x):
t := nejmenší list stromu T;
repeat
    t := otec t;
until t je kořen ∨ $x \leq H_t[1]$;
AB_TREE_INSERT(x, t); //jako v původním INSERTu, pouze jsme jinak inicializovali t

Složitost A-Sort-u

Označme $\mathbf{f_i} = |\{j > i : a_j < a_i\}|$. Je to tedy počet prvků posloupnosti, které v nesetříděné posloupnosti následují a_i, ale v setříděné patří před a_i. Hodnota **F** = $\sum f_i$, zvaná **počet inverzí**, vyjadřuje uspořádanost vstupní posloupnosti. Pro správně uspořádanou posloupnost je F = 0, pro obrácené uspořádanou posloupnost je F = n(n −1)/2. To jsou také mezní hodnoty, jichž může F nabývat.

A-sort vyžaduje čas O(n * max(1, log((F + 1)/n))). V nejhorším případě to je tedy O(n log n).

Vnitrni pamet – Neporovnavaci

Stabilni trizeni - pokud zachova poradi prvku se stejnym klicem takove, jake bylo na vstupu. (radix aj counting sort zachovavaju su su stabilne).

Radix sort - rozdelim do kastliku 0-9 (a-z, ..., podle abecedy v klicich) podle nejmene duleziteho znaku. Znovu postavim celou posloupnost serazenim kastliku za sebe dle jejich (znameho) poradi. Udelam to same pro 2. nejmene dulezity znak a dokola az po nejdulezitejsi.

Radix sort je zvláštní třídící algoritmus – jeho složitost je totiž lineární. Dosahuje to tím, že neporovnává všechny třídené prvky (složitost problému třídení pomocí porovnávání je O(n log n), takže by to jinak nebylo možné), je ho ale možné použít jen pro třídení dat podle klíce z nejaké ne príliš velké množiny – max. rozsah třídených hodnot závisí na tom, jak velké pole si mužeme dovolit vymezit v pameti pro tento úcel.

Nejjednodušší varianta (tzv. pigeonhole sort, nebo-li counting sort) opravdu pocítá s klíci ze zadaného rozmezí [l, h]. Pro nej si pripraví cílové pole velikosti h − l + 1, tj. „prihrádky“. Do nich pak prímo podle klíce prehazuje ctené prvky (jestliže prihrádky realizujeme jako seznamy, bude třídení dokonce stabilní). Nakonec projde prihrádky od zacátku do konce a co v nich najde, to vypíše (a výstup bude setrídený). Variantou counting sortu je bucket sort, kdy se do jedné prihrádky nedávají jen prvky se stejným klícem, ale prvky s klícem v nejakém malém rozmezí – ty pak lze setrídit rychle, protože jich zrejme nebude mnoho, a navíc se ušetrí pamet.

Protože ale klíce velikosti max. tisícu hodnot jsou vetšinou trochu málo, v praxi se bežne používají složitejší varianty – ty zahrnují nekolik pruchodu nahore popsaného algoritmu, pri nichž se trídí jenom podle casti klíce. Ty se delí na ty, které zacínají od nejméne významné casti klíce (least significant digit radix sort) a ty, které jdou od nejvýznamnejší casti (most significant digit). První z nich mají tu výhodu, že lze zachovat stabilitu trídení, druhá zase muže trídit i podle klícu ruzné délky a zastavovat se po nalezení unikátních prefixu, takže se hodí napr. pro lexikografické trídení podle retezcových klícu.

Least significant digit vypadá následovne:

1. Vezmi nejméne významnou cást klíce (urcitý pocet bitu).
2. Rozdel podle této casti klíce data do prihrádek, ale v nich zachovej jejich poradí (to je nutné kvuli následnému pruchodu, zároven to delá z tohoto algoritmu stabilní trídení).
3. Opakuj toto pro další (významnejší) cást klíce.

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

1. Vezmi nejvýznamnejší cást klíce (první písmeno, napríklad).
2. Rozdel prvky podle této casti do prihrádek (takže v jedné se jich octne docela hodne)
3. Rekurzivne setrid každou z prihrádek (zacni podle další casti klíce), pokud je v ní více než jeden prvek (tohle zarucí zastavení za rozlišujícím prefixem).
4. Slep prihrádky do jedné (setrídené) posloupnosti.

Popisované algoritmy vetšinou potrebují O(n + (h − l)) casu k trídení, je-li h − l (zhruba) pocet prihrádek – to znamená, že sice jde o složitost lineární, ale lineární i v poctu prihrádek, což se nemusí vždy oproti konvencnímu trídení vyplatit. Navíc jsou problémem vysoké nároky na pamet (nelze trídení provést „na místeÿ v jediném poli). Pro malou množinu hodnot klícu (nebo u most signifiant digit varianty krátké odlišující prefixy) jsou ale casove efektivnejší.

Bogo sort - srovnej vstupni posloupnost nahodne a zkus, jestli je setrizena :))

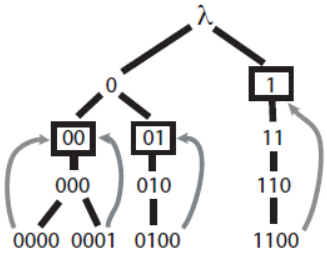
Externí hashování

Máme dáno universum U a vnější paměť'ové medium, které je rozděleno do stránek. Každá stránka může obsahovat nejvýše b záznamů. Chceme navrhnout ukládání prvků do stránek tak, aby operace MEMBER, INSERT, DELETE vyžadovaly jen konstatní počet přístupů do externí paměti, aby stránky byly dostatečně zaplněné. Tedy jinými slovy: chceme minimalizovat počet přístupů do vnější paměti. Vyhledáme vždy stránku ve vnější paměti - natáhneme ji do vnitřní paměti a tam provedeme vyhledávání - toto vyhledávání nás ale nezajíá.

Necht' U je universum, S ⊆ U. Externí medium má stránky o velikosti b. Je dána hašovací funkce h : U → $\{0, 1\}^k$ prostá (**kódovací funkce**), kde k je libovolné. Funkce h tedy generuje binární kódy délky k.

Vytvoříme strom množiny S. Vrcholy stromu jsou všechny prefixy slov h(s), s ∈ S. Pro vrchol α jsou jeho synové α0 a α1 (pokud existují). Listy jsou prvky h(s), s ∈ S.

Přílad: Necht' U = $\{0, 1\}^4$, b = 2 a S = {0100, 0001, 0000, 1001}



Definice: Pro s ∈ S definujeme **d(s)** = délka nejkratšího prefixu α slova h(s), že pod α je nejvýše b listů. Alternativní definice d(s) je, že d(s) = délka nejkratšího prefixu h(s) taková, že počet prvků t ∈ S, jejichž prefix délky d(s) je stejný jako prefix h(s), je nejvýše b.

Na obrázku vrchol reprezentující prefix 0, by ve svém podstromě měl 3 prvky, což je více, než požadujeme pomocí hodnoty b.

Definice: **d(S) = max { d(s) | s** ∈ **S }**. Z obrázku plyne, že hodnota d(S) = 2.

Reprezentace

- **Adresář (funkcia)** - ke každému slovu α délky d(S) je přiřazena adresa stránky, která obsahuje prvky s ∈ S, že h(s) má stejný prefix délky d(s) jako α. Pokud S takové prvky nemá, tak je mu přiřazena hodnota NIL.
  [v adresári sú uložené aj čísla d(S) a počet záznamov každej stránky]
- **Stránky (datová část)** - Prvek s ∈ S je uložen na stránce, která obsahuje všechny prvky t ∈ S takové, že prefix h(t) délky d(s) je stejný jako h(s). Prvky množiy S sú v stránke usporiadané lexikograficky.

Náš příklad potom vypadá nasledovně:

00	→	0000, 0001
01	→	0100
10	↘	
11	→	1001

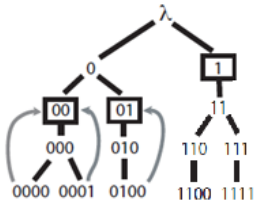
MEMBER(x):

1. Vypočteme h(x) a **načteme** adresář do interní paměti. Vezmeme prefix α slova h(x) o délce d(S) a nalezneme adresu stránky (vememe ukazatel na stránku) příslušející k α. Když je to stránka NIL, pak x ∉ S a konec.
2. **Načteme** stránku příslušející k α do interní paměti. Prohledáme ji, a pokud neobsahuje x, pak x ∉ S a konec. Když obsahuje x, pak provedeme požadované změny.
3. Stránku **uložíme** do externí paměti na její původní místo.

INSERT(x):

1. Vypočteme h(x) a **načteme** adresář do interní paměti. Vezmeme prefix α slova h(x) o délce d(S) (číslo, ktoré si držím poznačené v adresári) a nalezneme adresu stránky příslušející k α. Když stránka přiřazená k α je NIL, pokračujeme krokem 3, v opačném případě pokračujeme krokem 2.
2. **Načteme** stránku přiřazenou slovu α. Když x je uloženo na této stránce, pak skončíme. Když x není na této stránce, tak ho tam přidáme a následně:
   a. Pokud je na stránce nejvýše b prvků, **uložíme** stránku na její původní místo a skončíme.
   b. Když je na stránce více než b prvků s ∈ S, pak nalezneme nová h(s), která nám stránku rozdělí, a vytvoříme dvě stránky – jednu **uložíme** na místo původní stránky a druhou uložíme do nové stránky (do nového rámce na disku). Pokračujeme krokem 4.
3. Vytvoříme v interní paměti novou stránku, která obsahuje x, nalezneme novou stránku v externí paměti (rámec) a tam **uložíme** vytvořenou stránku. Všem slovům α v adresáři, která mají se slovem h(x) společný prefix délky d(x), bude přiřazena tato stránka a pokračujeme krokem 4.
4. **Načteme** opět adresář do interní paměti, aktualizujeme adresy přiřazených stránek a případně adresář zvětšíme (jeho d(S)). (To nastane, když nové slovo h(x) má d(x) větší než d(S). Pak nové d(S) je právě h(x). Aktualizovaný adresář **uložíme** do externí paměti.

INSERT(1111):



00	→	0000, 0001
01	→	0100
10	↘	
		1001, 1111
11	↗	

INSERT(0010):

000	→	0000, 0001
001	→	0010
010	→	0100
011	↗	
100	↘	
101	→	1001, 1111
110	↗	
111	↗	

DELETE(x):

1. Spočítáme h(x), **natáhneme** adresář, nalezneme d(S), nalezneme adresu stránky odpovídající prefixu h(x) délky d(S), zjistíme zda po odebrání prvku x může nastat spojování stránek a pokud ano, nalezneme adresu stránky, která se spojí s naší stránkou (kandidát).
2. **Natáhneme** odpovídající stránku do paměti, zjistíme zda obsahuje x, pokud ne, tak konec. Jinak:
   a. Obsahuje x a nemůže dojít ke sloučení stránek (stránka kandidáta dohromady obsahují více než b prvků), tak odstraníme x, stránku **uložíme** na stejné místo a aktualizujeme adresář (počty prvků na stránce).
   b. Obsahuje x a dojde ke slučování, pak odstraníme x, **natáhneme** druhou stránku a stránky sloučíme, **uložíme** novou stránku.
3. **Natáhneme** a aktualizujeme adresář: Nejprve provedeme opravení odkazů na stránky: pokud u sudého záznamu v adresáři dojde k vyprázdnění stránky, bude tam odkaz na NIL. Pokud dojde k vyprázdnění u liché stránky, přehodí se odkaz na předchozí (sudou) stránku. Potom se testuje, zda jde adresář zmenšit. Zmenšování provádíme tak dlouho, dokud to jde.

Test na zmenšení adresáře - adresy stránek přiřazené (2i + 1)-ímu slovu a (2i + 2)-ému slovu o délce d(S) jsou stejné pro všechna i. Pak se tato slova spojí a d(S) se zmenší o 1.

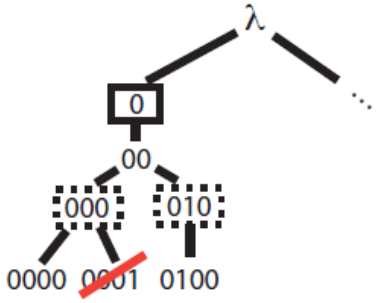
Příklad

DELETE(0100):

V adresáři budou pak položky 010 a 011 ukazovat na prázdnou stránku (NIL). Adresář zůstal po této operaci stejný.

DELETE(0001):

Je vidět, že po smazání prvku 0001 může být adresář opět zmenšen.



000	→	0000, 0010
001	↗	
010	→	NIL
011	↗	
100	↘	
101	→	1001, 1111
110	↗	
111	↗	

Zmenšení

00	→	0000, 0010
01	↗	
10	→	1100, 1111
11	↗	

a ještě zmenšení na:

0	→	0000, 0010
1	→	1100, 1111

Věta: Operace MEMBER vyžaduje **nejvýše tři** operace s externí pamětí. Operace INSERT a DELETE vyžadují **nejvýše šest** operací s externí pamětí.

Architektura pocitacu a siti

Uvod

Architektura pocítaca

Architektura pocítaca popisuje „všetko, co by mal vediet ten, ktorý programuje v assembleri / tvorí operacný systém“. Teda:

- z akých castí – štruktúra pocítaca, usporadanie
- význam castí – funkcia casti, ich vnútorná štruktúra
- ako spolu casti komunikujú – riadenie komukácie
- ako sa jednotlivé casti ovládajú, aká je ich funkcnost navonok

Víceúrovnová organizace pocítace

- Mikroprogramová úroven (priamo technické vybavenie pocítaca)
- Strojový jazyk pocítace (virtuálny stroj nad obvodovým riešením; vybavenie – popis architektúry a organizácie)
- Úroven operacního systému (doplnenie predchádzajúcej úrovne o súbor makroinštrukcií a novú organizáciu pamäti)
- Úroven assembleru (najnižšia úroven ludsky orientovaného jazyka)
- Úroven vyšších programovacích jazyku (obecné alebo problémovo orientované; prvá nestrojovo orientovaná úroven)
- Úroven aplikacních programu

Je teda potrebné definovat

- Inštrukcný súbor (definícia prechodovej funkcie medzi stavmi pocítaca, formát inštrukcie, spôsob zápisu, možnosti adresovania operandov)
- Registrový model (rozlišovanie registrov procesoru: podla volby, pomocou urcenia registru – explicitný/implicitný register; podla funkcie registru – riadiaci register/register operandu)
- Definice specializovaných jednotek (jednotka na výpocet vo floatoch; fetch/decode/execute jednotky)
- Paralelismy (rozklad na úlohy, ktoré sa dajú spracovat súcasne – granularita (programy, podprogramy, inštrukcie. . .))
- Stupen predikce (schopnost pripravit sa na ocakávanú udalost (nacítanie inštrukcie, nastavenie prenosu dát) – explicitná predikcia, štatistika, heuristiky, adaptívna predikcia)
- Datové struktury a reprezentáciu dát (spôsob uloženia dát v pocítaci, mapovacie funkcie medzi reálnym svetom a vnútorným uložením, minimálna a maximálna velkost adresovatelné jednotky)
- Adresové konvencie (ako sa pristupuje k dátovým štruktúram – segment+offset alebo lineárna adresácia; velkost pamäti a jej šírika, „povolené“ miesta)
- Rízení (spolupráca procesoru a ostatných jednotiek, interakcia s okolím, prerušenia – vnútorne/vonkajšie)

- Vstupy a výstupy (metódy prenosu dát medzi procesorom a ostatnými jednotkami/ pocítacom a okolím; zahrnuje definície dátových štruktúr, identifikácia zdroja/ciela, dátových ciest, protokoly, reakcie na chyby).
- Šíre datových cest
- Stupen sdílení (na úrovni obvodov – zdielanie obvodov procesoru a IO; na úrovni jednotiek – zdielanie ALU viacerými procesormi)

Von Neumannova architektura a její alternativy

Vychází z koncepce vzniklé v USA v letech 1940/1945, její hlavní tvůrce je John von Neumann.

Jednoducho

Vnitřní strukturu počítače tvoří pamět, vstup, výstup a procesor. Strukura počítače se nemění s typem úlohy. Je jednotná pamět pro text programu i data. Rozdíl mezi programem a daty je dán jen jejich interpretací. Pamět je posloupnost stejně velkých pamětových míst. Počítač je řízen tokem instrukcí. Ty se provádějí, kdyz na ně dojde řada. V kazdém okamziku probíhá jen jedna činnost.

Obecne

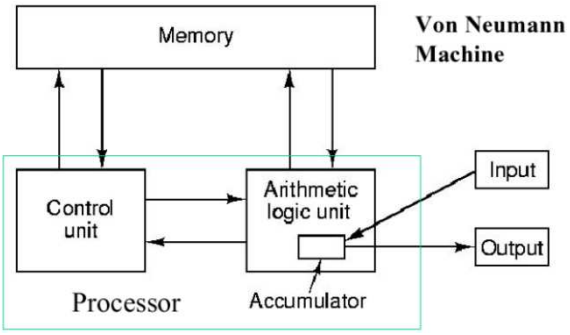
- Pocítac se skládá z rídící jednotky, ALU, pameti a I/O jednotek
- Štruktúra pocítaca sa nemení typom úlohy (tj. pocítac je programovaný obsahem pameti).
- Program se nejprve zavede do pameti, z ní se postupne poporade vybírají instrukce (a následující krok závisí na predchozím), poradí lze zmenit instrukcemi skoku.
- Do jedné pameti, delené na bunky stejné velikosti, se ukládají i zpracovávaná data. Data jsou reprezentovaná binárne.
- V každém okamžiku je vykonávána jen jedna cinnost. Je to architektura SISD (viz Flynnova taxonomie).

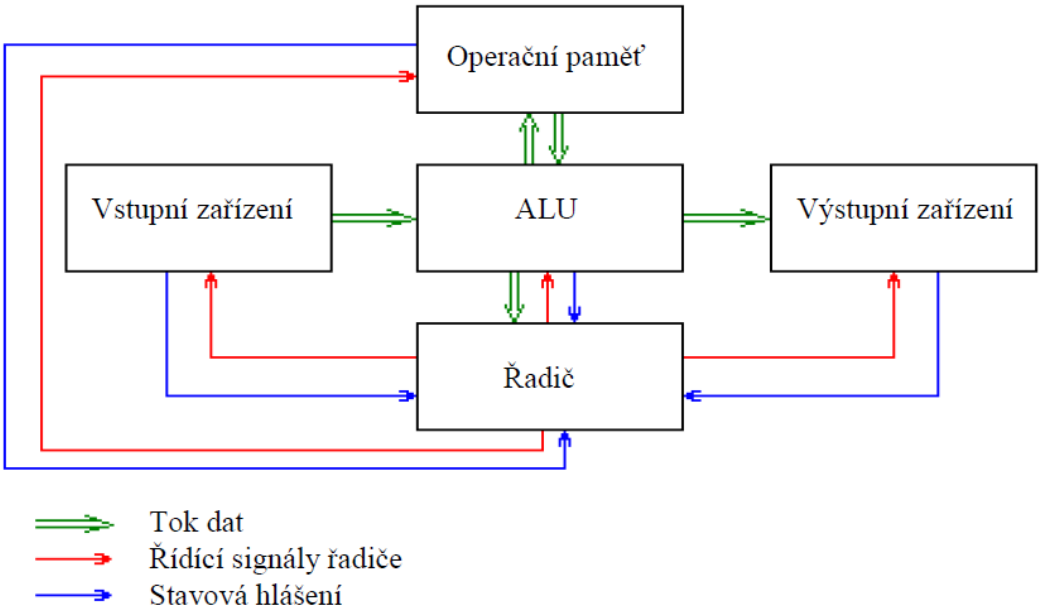
Je pevne daná instrukcní sada. Strojová instrukce obsahuje operacní znak, který urcuje druh operace, pocet parametru atd., a operandovú cást – umístnení jednotlivých operandu. Vykonat jednu instrukci znamená:

- (fetch) nacítat inštrukciu z pamäti do procesoru
- (decode) zistit o akú inštrukciu ide
- (load) pripravit zdrojové operandy
- (execute) vykonat operáciu
- (store) ulozit cielové operandy

Pri vykonávání programu jsou potrebné ruzné registry – nejduležitejší jsou: PC (**Program Counter**, obsahuje adresu následující instrukce), IR (**Instruction Register**, adresa práve vykonávané instrukce), SP (**Stack Pointer**, ukazatel na vrchol zásobníku), MAR (**memory access register** – adresa do operacní pameti), MBR (**memory buffer register**, dáta cítána/zapisována do pameti).

Struktura jednoprocesorového pocítace podle Von Neumanna:





Nevyhody: zastaralá, navrhována jako sekvenční, von Neumann bottleneck (propustnost dat mezi RAM a CPU je prilis nizka (relativne vzhledem k velikosti pameti a rychlosti cpu)).

Na Von Neumanove architekture jsou zalozeny vsechny dnesni mainstreamove pocitace. I kdyz od te doby doslo ke rade uprav (asynchronni I/O, cache, virtualni pamet).

Alternativy - počítač řízený tokem dat - operace se provede, jakmile jsou známy vsechny její operandy.

Harvardska architektura

Vytvorena až po Von Neumannove, liší se hlavne tím, že program se ukládá do jiné pameti než data (tzn. jsou 2 „druhy pameti“ – instrukcí a dat). Príklady jsou DSP procesory a mikrokontrolery (napr. AVR od Atmelu, a PIC – mají pamet na program a data a RISC instrukcní sadu; výhoda oddelených pametí je, že mužou mít ruznou bitovou hloubku – 8 bitové data, ale 12-, 14- ci 16- bitové instrukce (napr. ARM musí obcas použít více než jednou instrukci na zpracování obsahu plné velikosti)).

Oproti Von Neumannove nehrozí nebezpecí prepsání programu sebou samým, ale kvuli vetšímu poctu pametových sbernic je nárocnejší na výrobu. Pamet navíc nelze delit podle potreby (rozdelení je už dané).

Multiprocesory

Procesor

(CPU – central processing unit) je ústrední výkonnou jednotkou pocítace, která cte z pameti instrukce a na jejich základe vykonává program.

Základnými súcastami procesora sú:

- radic nebo rídicí jednotka, která rídí tok programu, tj. nacítání instrukcí, jejich dekódování, nacítání operandu instrukcí z operacní pameti a ukládání výsledku zpracování instrukcí
- sada registru k uchování operandu a mezivýsledku.
- jedna nebo více aritmeticko-logických jednotek (ALU), které provádí s daty aritmetické a logické operace.
- nekteré procesory obsahují jednu nebo nekolik jednotek plovoucí cárky (FPU), které provádí operace v plovoucí rádové cárce.

Súcasné procesory navyše casto obsahujú dalšie rozsiahle funkcné bloky (cache, rôzne periférie) – ktoré z „ortodoxného hladiska“ nie sú priamo súcastou jadra procesoru. Niektoré procesory môžu obsahovat viac jadier (+logiku slúžiacu k ich vzájomnému prepojeniu). Dalším trendom je SoC (System on Chip), kde sa na cipe procesora nachádzajú aj dalšie subsystémy napr. na spracovanie zvuku, grafiky alebo pripojenie externých periférií (takéto riešenia sa využívajú väcšinou v PDA, domácej elektronike, mobiloch atd.).

Delení podle instrucní sady

Podla inštrukcnej sady je možné procesory rozdelit na:

- **CISC (Complex Instruction Set Computer):** poskytuje rozsiahlu inštrukcnú sadu spolu s rôznymi variantami inštrukcií. Jedna inštrukcia napr. môže vykonat vela low-level operácií (nacítanie z pamäti, vykonat aritmetickú operáciu a výsledok uložit). Takéto inštrukcie zjednodušovali zápis programov (inštrukcie boli bližšie vyšším programovacím jazykom) a zmenšovali velkost programu a pocet prístupov do pamäti – co bolo v 60tych rokoch dôležité. Avšak nie vždy je vykonanie jednej zložitej operácie rýchlejšie ako vykonanie viac menej zložitých miesto toho (napr. kvôli zložitému dekódovaniu a použitiu mikrokódu na volanie jednoduchých „podinštrukciíÿ). Príkladmi CISC architektúr procesorov sú System/360, Motorola 68000 a Intel x86. V súcasnosti napr. x86 rozkladá zložité inštrukcie na „micro-operationsÿ ktoré môžu byt pipeline-ou spracované paralelne a vyšší výkon je tak dosahovaný na väcšom rozsahu inštrukcií. Vdaka tomu sú súcasné x86 procesory minimálne rovnako výkonné ako ozajstné RISC architektúry.
- **RISC (Reduced Instruction Set Computer):** design CPU ktorý uprednosnuje jednoduchšiu inštrukcnú sadu a menšiu zložitost adresovacích modelov – vdaka comu je možné dosiahnut lacnejšiu implementáciu, väcšiu úroven paralelizmu a úcinnejšie kompilátory. Dôvodom vzniku bolo aj nevyužívanie celej CISC inštrukcnej sady a uprednostnovania len obmedzenej podmnožiny (designéri procesorov potom optimalizovali len tieto podmnožiny a tak sa zvyšné inštrukcie používali ešte menej.. .). Kvôli väcšiemu poctu inštrukcií

však musia RISC procesory castejšie pristupovat k pamäti. . . Príkladmi RISC procesorov sú napr. SPARC a ARM. V architekturách typu Post-RISC jde o spojení RISCových vlastností s technikami zvýšení výkonu, jako je out-of-order vykonávání a paralelismus.

- **VLIW (Very Long Instruction Word or VLIW):** refers to a CPU architecture designed to take advantage of instruction level parallelism (ILP). A processor that executes every instruction one after the other (i.e. a non-pipelined scalar architecture) may use processor resources inefficiently, potentially leading to poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (this is pipelining), or even executing multiple instructions entirely simultaneously as in superscalar architectures. The VLIW approach, on the other hand, executes operation in parallel based on a fixed schedule determined when programs are compiled. Since determining the order of execution of operations (including which operations can execute simultaneously) is handled by the compiler, the procesor does not need the scheduling hardware that the three techniques described above require. As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs.
- **EPIC (Nekdy oznacován za poddruh VLIW):** Explicitly Parallel Instruction Computing (EPIC) is a computing paradigm that began to be researched in the 1990s. This paradigm is also called Independence architectures. It was used by Intel and HP in the development of Intel's IA-64 architecture, and has been implemented in Intel's Itanium and Itanium 2 line of server processors. The goal of EPIC was to increase the ability of microprocessors to exekute software instructions in parallel, by using the compiler, rather than komplex on-die circuitry, to identify and leverage opportunities for parallel execution. This would allow performance to be scaled more rapidly in future procesor designs, without resorting to ever-higher clock frequencies, which have since become problematic due to associated power and cooling issues.

Rekneme, že procesor má **ortogonální instrukcní sadu**, pokud žádná instrukce nepredpokládá implicitne použití nekterých registru. To umožnuje jednodušší práci algoritmum pridelování registru v prekladacích. Príkladem neortogonální instrukcní sady je i x86.

Další delení

Dalej je možné procesory rozdelit podla dlžky operandov v bitoch (8, 16, 32, 64. . .), ktorý je procesor schopný spracovat v jednom kroku. V embedded zariadeniach sa najcastejšie používajú 4- a 8-bitové procesory. V PDA, mobiloch a videohrách 8 resp. 16 bitové. 32 a viac bitov využíajú napr. osobné pocítace a laserové tlaciarne. Dôležitou vlastnostou je aj taktovacia frekvencia jadra, MIPS (millions of instructions per second) a jeho rýchlost. V súcasnosti je tažké dávat do svislosti výkon procesorov s ich frekvenciou (resp. MIPS) – kým Pentium zvládne na výpocet vo floatoch, jednoduchý 8-bitový PIC na to potrebuje ovela viac taktov. Dalším „problémom" je superskalarita procesorov, ktorá im umožnuje vykonat viacero nezávislých inštrukcií pocas jedného taktu.

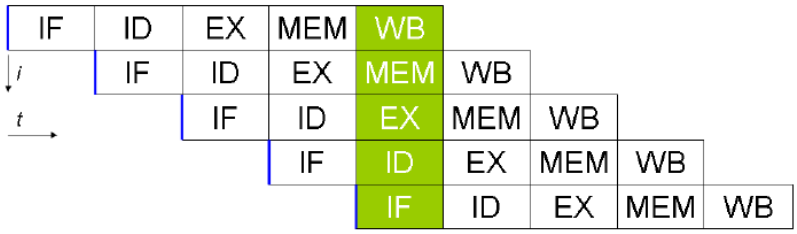
Techniky pro zvýšení výkonu

Zvyšovat výkon (procesorov) je možné viacerými spôsobmi. Najjednoduchším (a najpomalším) typom je Subskalárny CPU (nacíta a spracúva len jednu inštrukciu naraz – preto musí celý procesor cakat kým vykonávanie inštrukcie skončí; je tak zdržovaný dlhšie trvajúcimi inštrukciami).

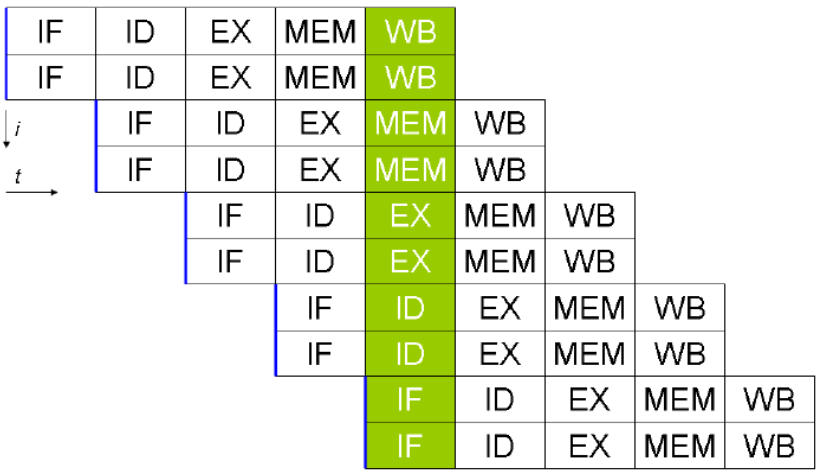


Pokusy o dosiahnutie skalárneho a lepšieho výkonu vyústili do designov ktoré sa správajú menej lineárne a viac paralelne. Co sa týka paralelizmu v procesoroch, používajú sa dva druhy pojmov na ich klasifikáciu – Instruction level parallelism (zvyšovanie rýchlosti vykonávania inštrukcií v procesore a teda zväčšovanie využitia prostriedkov na cipe) a Thread level parallelism (zväčšovanie poctu vlákien, ktoré dokáže CPU vykonávat naraz).

Pipeline: Zlepšenie je možné dosiahnút pomocou „instruction pipeliningÿ-u, ktoré je použíté vo väčšine moderných procesorov. Umožnuje vykonanie via ako jednej inštrukcie v jednom kroku vdaka rozloženiu spracovávania inštrukcie na viac menších krokov:



Superskalarita: Dialša možnost je použitie superscalar designu, ktorý obsahuje dlhú inštrukcnú pipeline a viacero identických execution jednotiek.



Out of order execution:

1. Nactení instrukce, prípadne její rozdrobení na mikroinstrukce
2. Zarazení do vyckávací stanice (instruction pool)
3. Instrukce ceká na všechny svoje operandy
4. Instrukce se vykoná ve své výkonné jednotce (je vybírána z instruction poolu nezávisle na ostatních)
5. Výsledky se uchovají ve fronte (reorder buffer)
6. Až se všechny starší instrukce zapíší do registru, zapíše se výsledek této instrukce (opetovné razení)

Predikce skoku: Hluboké pipeliny mají problém, pokud podmínený skok není proveden; dynamická predicke skoku (historie CPU – vzory nejaké hloubky) vs. statická (bez nápovedy – skok vpred se neprovede, skok vzad se provede; s nápovedou – prekladac odhaduje pravdepodobnost skoku)

Spekulativní vykonávaní: Vykonávání kódu, který nemusí být zapotrebí; významná disproporce mezi rychlostí CPU a pameti; typické využití je znacné predsunutí ctecích operací; CPU provádí i odsouvání zápisových operací

Data parallelism: SIMD inštrukcie (napr. multimediálne inštrukcie), vektorové procesory. . .

Multiprocesor

O multiprocesoru mluvíme, pokud je použito dvou nebo více procesoru (CPU) v rámci jednoho pocítacového systému. Termín je také používán mluvíme-li o schopnosti systému využívat více procesoru a/nebo schopnosti rozdelovat úlohy mezi jednotlivými procesory.

Proc multiprocessing?

Snaha o zvýšení výkonu počítačového systému je spojení více počítačů či procesorů do jednoho celku, do tzv. vícepočítačových a víceprocesorových systémů. Výhodou je vyšší výkon, vyšší flexibilita v požadavcích na zpracování dat a vyšší spolehlivost systému. Tento rys spočívající v zálohování procesorů se významně projeví u tzv. nepřetržitého zpracování, kdy je odstavení systému v důsledku výskytu určité chyby prakticky nemyslitelné.

Vztah k datum a instrukcím

Klasicke jednoprocesorove systemy jsou typu **SISD** (single instruction, single data). V multiprocessingu mozu byt procesory pouzivane na vykonanie jednej sekvencie instrukcii vo viacerych kontextech – na viacerych datach (single-instruction, multiple-data or **SIMD**, například scitanie dvoch vektorov); viac sekvencii instrukcii v jednom kontexte – na jedny data (multiple-instruction, single-data or **MISD**, used for redundancy in fail-safe systems, v praxi se nevyskytuje), alebo viac sekvencii instrukcii vo viacerych kontextech (multiple-instruction, multiple-data or **MIMD**).

Symetrie

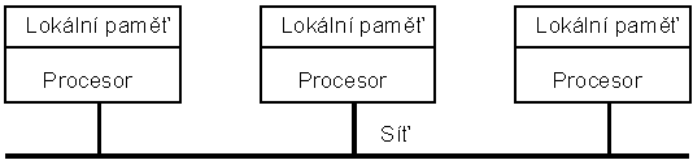
V multiprocesorovych systemoch si mozu byt vsetky procesory ekvivalentne alebo niektore mozu byt urcene pre speciálně ucely. Kombinacia hardwaru a design operacneho systemu urcuju, symetriu v danom systeme. Napriklad hardware alebo aj software mozu vyzadovat, aby len jeden presne urceny procesor reagoval na vsetky hardwarove prerusenia, pritom vsetky ostatne ulohy (praca) v systeme boli ekvivalentne rozdelene medzi vsetky procesory. V niektorych pripadoch je například nutne aby sa exekucia kernel-mode kodu vykonavala vyhradne len jedenym konkretny procesorom alebo aby bola vykonavana vždy len jednym procesorom sucasne. Exekucia user-mode kodu moze byt vykonavana libovolnou kombinaciou procesorov. Multiprocesorove systémy v ktorych su pouzite taketo obmedzenia su častokrát jednoduchsie nedesignovat ale stracaju na efektivnosti v porovnani so systemamy, kde su vsetky procesory ekvivalentne.

Systemy, kde su si vsetky procesory vždy ekvivalentne sa nazyvaju symetricke multiprocesorove systémy (symmetric multiprocessing (**SMP**) systems) = UMA.

Systemy, kde si nie su vsetky procesory vždy ekvivalentne, možno rozdelit na viac kategorii: asymmetric multiprocessing (**ASMP**), nonuniform memory access (**NUMA**)…

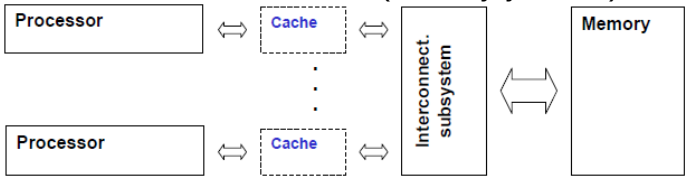
Paralelni Architektury

Volně spojené (Loosely coupled, grids) = **multicomputery** - propojené skupiny počítačů – mají vlastní pamet. Spojení je nejčastěji realizováno Gb ethernetem.

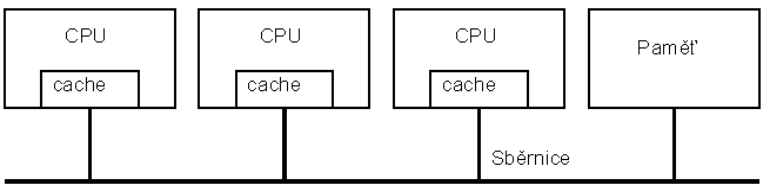


Těsně spojené (Tightly coupled) = **multiprocesory** - mají sdilenou pamět.

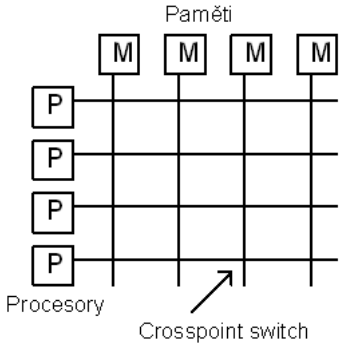
- **UMA** (Uniform memory access) - symetrický přístup k paměti. Procesory mohou mít vlastní lokální pamět v podobě **cache**. Udržovat cache každého procesoru konzistentní vyhledem k ostatním je netriviální. Bottleneck tohoto přístupu je ve sběrnici na kterou jsou napojeny procesory. **CC-UMA** (cache coherent UMA) – ak jeden procesor zapíše niečo do zdielanej paměti, všetky ostatné sa o tom dozvedia (vo svojej cache). Max. počet procesorů 32 - 64.



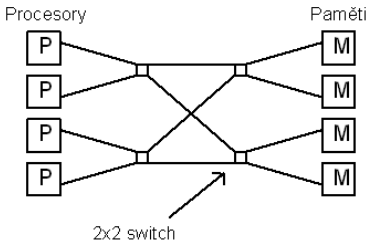
  - **Propojené sběrnicí** – sbernicova architektura, levná, běžně dostupná.



- **Propojené n^2 přepínači** (crossbar nebo crosspoint switched) . Sběrnicová mřížka = procesory × paměťové bloky. Nákladné – kvadratický počet přepínačů.

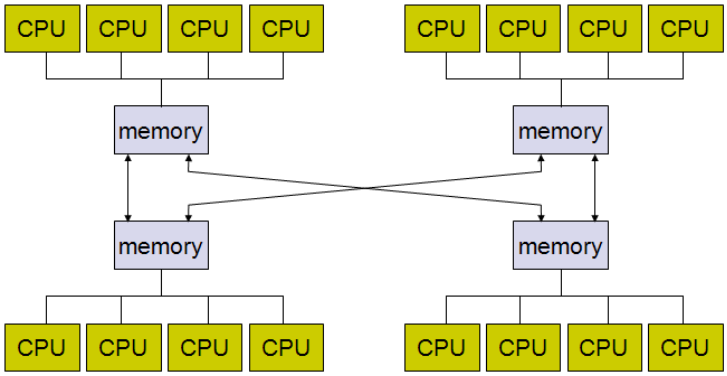


- **Propojené $n*\log_2 n$ přepínači** (multistage switched, omega networks) Postupně přepínaná cesta (přes sběrnici) mezi procesorem a pamětí.



- **Propojené jinak**, třeba hyperkrychle, klika...

- **NUMA** (Non uniform memory access) - každý procesor ma “svoji“ paměť. Pamět je fyzicky aj logicky sdílena, ale cas pristupu k datam zavisi na tom, kde sa potrebne data nachadzaju vzhladom na procesor, ktory data potrebuje. K vlastnej pamati pristupuje teda rychlejsie jako k vzdialenej pamati, ktora je bud lokalna pamat nějakého ineho procesora (resp. procesorov). K pamati se přistupuje pomocou globálnej adresácie. NUMA systémy vznikajú většinou propojením více UMA systémů.
  - **S-COMA**, skupina procesorů má svoji "bližší" paměť, požadavek na obsah paměti náležející jiným procesorům se propaguje vyšší vrstvě.
  - **ccNUMA (cache-coherent NUMA)**, globální adresace, hodně řeší cache. Directory memory - speciální paměť instancí a jejich kopií a platností.



Mikroprogramové a klasické řadiče

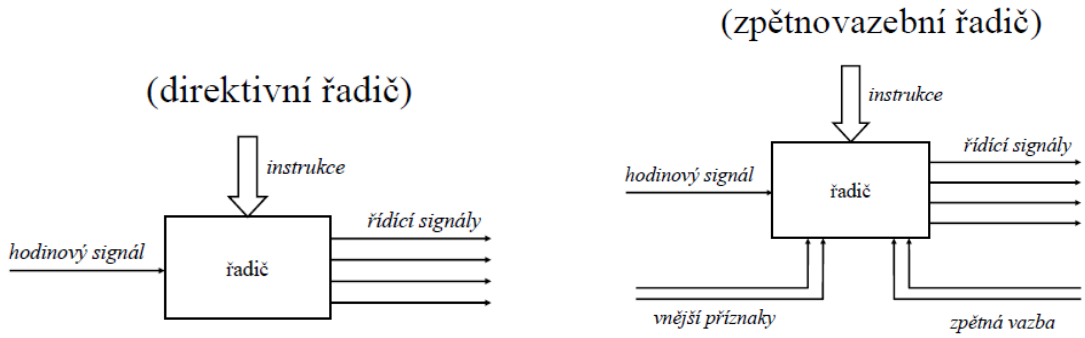
Řadič je castou historickej referencnenej Von Neumanovej architektury pocitaca. V sucasnosti je typicky aj castou CPU. Kontroluje data flow procesoru a koordinuje aktivity aj iných zariadeni. Vo vseobecnosti zajistuje provadeni mikroprogramu (Při provedeni strojove akce se postupne provadeji dilci akce – **mikrooperace**. Predpis o tom, jake mikrooperace a v jakem poradi se meji provádět se nazyva **mikroprogram**).

CPU instrukce sestává z řady fází (cyklů), např. **fetch**, **decode**, **execute**, **store**. Každý cyklus sestává z posloupnosti elementárních akcí – **mikrooperací** (např. přesun mezi registry, přesun mezi registrem a sběrnicí (k paměti), jednoduchá ALU operace).

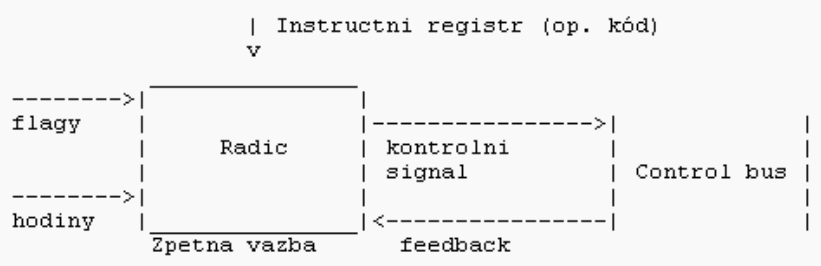
Funkce řadiče – (řadič zajišťuje v podstatě dva úkoly):

1. Zpracování mikro-operace ve správném pořadí jak vyžaduje prováděný program.
2. Generování řídících signálů, které pro každou mikro-operaci způsobí její vykonání.

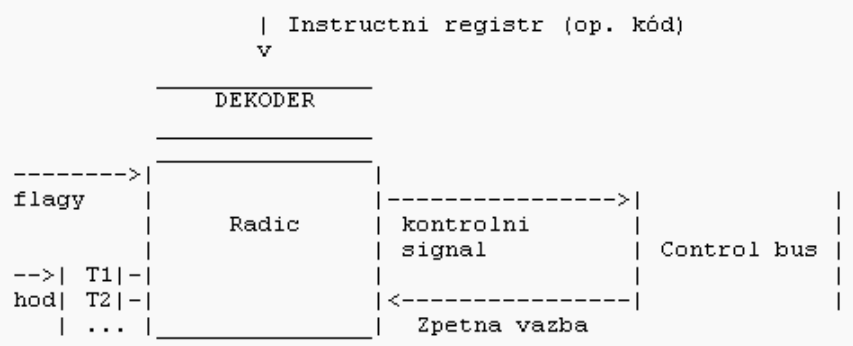
Řídící signály spouští otevírání a zavírání logických hradel vykonávajících mikrooperace. I/O periferie mají většinou vlastní řídící jednotku, která spracovává I/O operace, aby procesor (resp. řadič) nemusel na pomalou periferii čekat. Procesor teda jen pověří řídící jednotku příslušné periferie aby I/O operaci spracovala a sám pak může dělat neco jiného (užitečného). Samotná I/O operace se stává **asynchronní** (nečeká na ni procesor). Aby se ale procesor (resp. řadič) dozvědel, že řídící jednotka příslušné I/O periferie je už hotová, potřebuje mít cesty pro spětnou vazbu. Vzniká tzv. **zpětnovazební řadič**.



Klasický řadič (hard-wired) má všechny postupy vykonávání instrukcí zadrátované v HW. Pro každou instrukci a stav je zadrátováno, jaké řídící signály mají být dány. Jde vlastně o konecny stavový automat (přechody stavovým diagramem, standardní metody sekvenční logiky). Je používaný hlavně v RISC architekturách, protože mají menší počet instrukcí. Je rychlý a má méně prvků (logických a tím pádem i elektrických).



Mikroprogramový řadič – logika řadiče je určena mikroprogramem. **Mikroprogram** je sekvence příkazů ve speciálním mikroprogramovacím jazyce. Je flexibilnější a typický pro architektúry CISC. Trochu komplikace s pipeliningem, řídící signály ještě závisí na instrukčním cyklu.



Provádění mikrokódu

1. Dekódování operačního kódu instrukce
2. Vyzdvihování mikroinstrukcí z řídící paměti
3. Control address register najde mikroinstrukci, která se má vykonat
4. Microinstruction registr přijme tuto instrukci a adresu následující mikroinstrukce
5. Mikroinstrukce je vykonána
6. Adresa další mikroinstrukce je použita pro vyhledání další
7. Jsou-li vykonány všechny mikroinstrukce, načte se další konvenční instrukce. Ne-li, vykonávají se další mikroinstrukce
8. Podmíněné skoky jsou provedeny tak, že je modifikována adresa další mikroinstrukce

Mikroprogramování

Mikroprogram vypadá nějak takto: Propoj registr 1 se vstupem ALU "A", propoj registr 2 se vstupem ALU "B", nastav ALU na sčítání, vynuluj carry bit v ALU, propoj výstup ALU s registrem 8, aktualizuj registr příznaků podle příznaků v ALU, posuň čítač instrukcí, načti další instrukci...

Mikroinstrukce jsou uloženy v oblasti zvané **control store**, z něhož vybírá instrukce **mikrosequncer** - ten typicky nějak spojuje počítadlo, aktuální hodnotu instrukce kterou ALU zpracovává a pole v mikroinstrukci které říká kam dál. Mikroinstrukce je zlozena z:

- vstupni operandy
- operace ALU
- adresu pro vysledek operace
- adresu pristi mikroinstrukce

Horizontalni format mikroinstrukce

- mikroinstrukce obsahuje přímo hodnoty řídících signálů
- není třeba dekódovat -> rychlost
- libovolná kombinace -> pružnost
- velké nároky na prostor
- mikroinstrukce obsahuje za sebou pokyny pro všechna propojování, nastavování, skok dál, etc; instrukce jsou široké 56 bitů nebo i víc.

Vertikální formát mikroinstrukce

- instrukce jsou zakódované (musí existovat dekodér který generuje signaly)
- některé kombinace se vylučují navzájem -> možno zakódovat -> menší objem
- nutno dekódovat -> zpomalení, zesložitění
- pevný návrh -> méně pružné

Format může vypadat třeba takhle :

```
_____________________________________________________________________________
|ISTRUKCE RIZENI CPU|INSTRUKCE RIZENI NA BUS|PODMINKA SKOKU|ADRESA MIKROINSTRUKCE PO SKOKU|
-----------------------------------------------------------------------------
```

Provedení mikroinstrukce

- Nastav řídící signály
- Podle podminky se rozhodni
  - Buď proveď další instrukci
  - Nebo skoč

Vertikální mikrokód -- mikroinstrukce jsou zakódované, takže jsou kratší, ale vyžadují další zpracování, aby se z nich dostaly ty pokyny. Někdy to je assembler

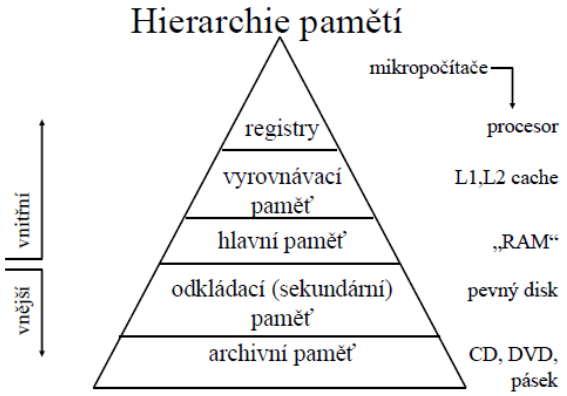
jednoduššího počítače, který emuluje složitější počítač. Další možná forma je dvojice (jednotka, pokyn_jednotce) -- je potřeba míň paměti, ale ALU je pomalejší.

Nanoprogramování, instrukce mikrokódu se komprimuje tak, že je vlastně jen indexem v adresáři nanoinstrukcí, které něco dělají. U mikroprogramování se každá instrukce používá k řízení okolního hardware a to ať už přímo nebo až po dekódvání. Vzhledem k tomu, že se vyskytuje pouze konečné množství různých kombinací, kterých řídící bity mohou nabývat a toto mnoství je relativně malé - je výhodné použít nanopamět, ve které budou uloženy řídící signály a tuto pamět adresovat pomocí zvláštního pole v mikroinstrukci. Tento dvojúrovňový mikropaměťový systém se nazývá nanoprogramování.

Se klesající cenou tranzistoru začal dominovat horizontální mikrokód.

Wilkesova řídíci jednotka je založena na řídící matici. Mikroinstukce vybere řádek, podle toho jestli je na dané pozici v řádku jednička se provedou řídící signály odpovídající sloupcům v první části matice a vybere se nasledující instrukce podle jedniček a nul ve sloupcích v příslušném řádku druhé části matice.

Paměťová hierarchie



- **Registry CPU** - 10ky-100vky bajtu (IA-32: obecné registy pár 10tek), IA-64 až kB (extrém), stejne rychlé jako CPU.
- **Vyrovnavaci pamet (cache)** - z pohledu aplikací není prímo adresovatelná; dnes rádove MB, rozdelení podle úcelu, nekolik vrstev. L1 cache (cca 10ky kB) – delené instrukce/data; L2 (cca MB) sdílené instr&data, beží na rychlosti CPU (drív bývala pomalejší), servery – L3 (cca 10MB). Vyrovnává rozdíl rychlosti CPU a RAM. Využívá lokality programu – cyklení na míste; sekvencního prístupu k datum. Pokud nenajdu co chci v cache – „cache-miss“, nacítá se potrebné z RAM (po blocích), jinak (v 95-7% prípadu) nastane „cache-hit“, tj. požadovaná data v cache opravdu jsou a do RAM nemusím.
- **Operacni (hlavni) pamet RAM** - prímo adresovatelná procesorem, 100MB – GB; pomalejší než CPU; CAS – doba prístupu na urc. místo – nejvíc zdržuje (v 1 sloupci už cte rychle, dat. tok dostatecný), další – latence – doba než data dotecou do CPU – hraje roli vzdálenost (AMD- integrovaný radic v CPU)
- **Sekundarni pamet** - není prímo adresovatelná, typicky HDD; náh. prístup, ale pomalejší. 100GB, ruzné druhy – IDE, SATA, SCSI; nejvíc zdržuje prístupová doba (cas seeku) cca 2-10ms; obvykle sektor – 512 B; roli hraje i rychlost otácení (4200 – 15000 RPM) – taky rádove ms.
- **Archivni pamet** - nejpomalejší, z teorie nejvetší, dnes ale neplatí; typicky – pásky; pro vetší kapacitu – autoloadery; sekvencní prístup; dnes – kvuli rychlosti casto zálohování RAIDem.

Vyrovnávací paměti

obvykle pouziti - tam, kde je vyrazny rozdil v rychlostech - procesor vs ram, pristupy na disk. Vyuziva se myslenky **casove lokality** pristupu (cacheovanie) - data jednou pouzita budou pravdepodobne pouzita podruhe a pametove lokality. Kdyz pouziju nejaka data, tak asi budu pouzivat i data ulozena v nejblizsim okoli (hlavni operacni pamet pracuje po strankach, takze typicky nactu i okoli) = **prostorova lokalita**.

Cache (vyrovnavaci pamet) ma typicky mnohem mensi kapacitu nez ta pamet, ktera je cacheovana (CPU ma radove 0,5MB cache vs RAM 0,5GB; HDD ma cca. 8MB cache a kapacitu 200GB). RAM je v podstate taky cache sekundarni pameti (disku).

Data v cache jsou ulozena spolu se svou adresou. K vyhledavani je mozno pouzit nekolik druhu mapovani:

- **Plne asociativni** - při hledání hledána přímo adresa v asociativním adresáři.
- **Prime** - každý blok v cache ma své místo.
- **Skupinové asociativní mapování** - kombinace předchozích dvou – asociativni vyhledavani v ramci dane mnoziny/skupiny.

Pro uvolnovani bloku z cache mozno pouzit ruzne algoritmy (LRU, LFU, FIFO, ...).

Je třeba zajistit, aby data v cache odpovídala datům hlavní paměti:

- **Write-thru** - používat průpis (co se zapíše do vyrovnávací paměti, zapíše se okamžitě i do hlavní). Nevyhodu su caste zapisy do hlavnej pamati.
- **Write-back** - uklízet modifikované bloky do hlavní paměti (vyšší výkon). Při zápisu do vyrovnávací paměti se pouze nastaví (dirty) příznak a do hlavní paměti se data zapíší, až když jsou vyhozena z cache (a pouze pokud mají nastaveno dirty).

U multiprocesoru je potřeba zaručit konzistenci dat mezi procesorovými cache.

Efektivni pristupova doba - "amortizovany" cas potrebny k pristupu do pameti - pomoci cetnosti vypadku stranek a prumerneho casu pristupu do cache a cacheovne pameti se spocita, jaky byl prumerny pristupovy cas ke strance.

Stránkování a segmentace

Správce pameti

Cást OS, která spravuje pametovou hierarchii se nazývá správce pameti (memory manager):

- udržuje informace o volné/plné cásti pameti
- stará se o pridelování pameti
- a „výmenu pameti s diskem“

Prirazení adresy

- pri prekladu (je již známo umístení procesu, generuje se absolutní kód, PS:
- statické linkování)
- pri zavádení (OS rozhodne o umístení – generuje se kód s relokacemi, PS:
- dynamické linkování)
- za behu (proces se muže stehovat i za behu, relokacní registr)

Overlay – Proces potrebuje více pameti než je skutecne k dispozici. Programátor tedy rozdelí program na nezávislé cásti (které s v pameti podle potreby vymennují) a cást nezbytnou pro všechny cásti. . .

Výmena (swapping) – delá se, protože proces musí být v hlavní pameti, aby jeho instrukce mohly být vykonávány procesorem.. . Jde o výmenu obsahu pameti mezi hlavní a záložní.

Preklad adresy – nutný, protože proces pracuje v logickém (virtuálním) adresovém prostoru, ale HW pracuje s fyzickým adresovým prostorem.. .

Spojité pridelování – pridelení jednoho bloku / více pametových oddílu (pevne – pamet pevne rozdelena na cásti pro ruzné velikosti bloku/volne – v libovolné cásti volné pameti muže být alokován libovolne veliký blok)

Informace o obsazení pameti – bitová mapa / spojový seznam volných bloku (spojování uvolneného bloku se sousedy)

Alokacní algoritmy

- **First-fit** – první volný dostatecné velikosti – rychlý, obcas ale rozdelí velkou díru
- **Next-fit** – další volný dostatecné velikosti – jako First-fit, ale rychlejší
- **Best-fit** – nejmenší volný dostatecné velikosti – pomalý (prohledává celý seznam), zanechává malinké díry (ale nechává velké díry vcelku)
- **Worst-fit** – nejvetší volný – pomalý (prohledává celý seznam), rozdelí velké díry
- **Buddy systém** – pamet rozdelena na bloky o velikosti 2^n, bloky stejné velikosti v seznamu, pri pridelení zaokrouhlit na nejbližší 2^n, pokud není volný,

rozštípnou se vetší bloky na príslušné menší velikosti, pri uvolnení pameti se slucují sousední bloky (buddy)

Fragmentace pameti

- **externí** – volný prostor rozdelen na malé kousky, pravidlo 50% – po nejaké dobe behu programu bude cca 50% pameti fragmentováno a u toho to zustává
- **interní** – nevyužití celého prideleného prostoru
- **sesypání** – pouze pri prirazení adresy za behu, nebo segmentaci – nelze při statickém pridelení adresy

Virtualni pamet - zajišťuje větší adresový prostor, než je k dispozici fyzicke pameti. Zaistuje aj ochranu pamati procesu (do pamate procesu nemá sancu siahnut iny proce -> preklad virtualnych na fyzicke). Existují dvě základní metody:

Stránkování

Procesy mají k dispozici velký virtuální adresový prostor. Operační paměť tvoří fyzický adresový prostor. Virtuální i fyzická pamět je rozdělena na bloky stejné velikosti - virtuální na **stránky**, fyzická na **rámce**. Proces se odvolává pouze na adresy ve svém virtuálním prostoru.

32 bitova virtualni "pametova adresa" pouzivana procesem je rozdelena na adresu stranky a adresu ve strance. U 4kB stranek se prvnich **20 bitu** pouzije pro mechanismus hledani **prislusneho ramce** a az je nalezen, tak poslednich **12 bitu** urcuje poradi v ramci, kde je to, co hledame - **offset**.

Operační systém plní **stránkovací tabulky** informacemi o tom, které stránky jsou v kterých rámcích - **mapování**. Při pokusu procesu o přístup k paměti jednotka řízení paměti (**mmu - memory management unit**) dle stránkovacích tabulek zjistí, zda se stránka obsahující požadované místo nalézá v operační paměti. Pokud ano, provede **překlad virtuální adresy na fyzickou**.

Pokud ne, vygeneruje přerušení - nastal **výpadek stránky** (page fault). OS potom nalezne **volný rámec** (případně nějaku stranku a k nej prislusny ramec uvolní tím, že jej obsah zapíše do odkládací paměti (ak bola modifikovana) - viz strategie vymeny stranek) a z odkládací paměti **natáhne požadovanou stránku do rámce** a zavede mapovani do stránkovací tabulky. Procesor nyní zopakuje instrukci, která přerušení vyvolala. Celý postup je pro proces naprosto transparentní.

Pro 32 bitovy prostor a 4kB stranky maji strankovaci tabulky uz velikost 4MB, takze se pouziva **vice urovnove strankovani** (strategie jako indexovani souboru) aby se ušetřil prostor potřebný pro stránkovací tabulky.

Zvysovani vykonu: mmu primo soucasti procesoru, **TLB - transaction lookaside buffer** - naposledy pouzita mapovani jsou v malem asociativnim bufferu, kde se vyhledaji rychleji.

Algoritmus vypadku stranky

- výjímka procesoru
- uložit stav CPU (kontext)
- zjistit VA
- kontrola platnosti adresy a práv
- nalezení volného rámce
- zrušit mapování na nalezený rámec
- pokud je vyhazovaný rámec vyhazován, spustit ukládání na disk
- načíst z disku požadovanou stránku do rámce
- zavést mapování
- obnovit kontext

Pri implementaci stránkování je nutno brat v úvahu

- **znovuspuštení instrukce** - je potreba aby procesor po výpadku zkusil prístup do pameti znova. dnes umí všechny CPU, napr. 68xxx - problémy (prerušení v pulce instrukce)
- **sdílení stránek** - jednomu rámci odp. víc stránek -> pokud s ním neco delám, týká se to všech stránek! musím vše ost. odmapovat. musím si pamatovat mapování pro každý rámec - obrácené tabulky.
- **odstranení položky z TLB pri rušení mapování** - nestací zmenit tabulky, musí se vyhodit i z TLB (kde to muže, ale nemusí být). problém - u multiprocesoru má každá CPU vlastní TLB, tabulky jsou sdílené -> CPU pri rušení mapování musí poslat interrupt s rozkazem ke smazání všem (i sobe), pockat na potvrzení akce od všech.

Strategie vymeny stranek

Optimalni stranka - vyhodit stranku, ktera bude potreba za nejdelsi casovy usek - neimplementovatelna teoreticka strategie.

FIFO - klasicka fronta – vyhazuje sa stranka ktora bola v pamati najdlhsie, trpi Beladiovou anomalii.

LFU - Least Frequently Used - obetí je nejdéle nepouzitý rámec, stranky maji citac pristupu. Hlavni nedostatek je ten ze stranka hodne pouzita minule a uz nikdy nepouzita v budoucnu ma malo sanci na vyhozeni – reseni aging.

LRU - Least Recently Used - Least Recently Used - nejlepsi aproximace optimální strategie.

- casovy "citac" - obsahuje timestamp posledniho pristupu - moc nefunguje při procházení velkých struktur.
- zasobnik implementovany jako dvojity spojovy seznam. Pokud pouziju stranku tak se presune na vrchol. Nevic pouzivana stranka je na vrcholu, nejmin pouzivana je na dne. Vse je v O(1).

NRU - Not Recently Used - stránky mají čítač přístupu a špinavosti, čítač přístupu se občas maže, pak se první vyhazují stránky nečtené a čisté, pak nečtené a špinavé atd.

One Hand Clock - kruhový seznam, při nutnosti vyhodit stránku se pustí ručička, která nuluje access bit pokud je nastavený a vyhazuje stránku pokud je nulový.

Two Hand Clock - první ručička maže accessed bity, druhá (v pevném odstupu za ní) vyhazuje stránky s nulovým bitem, odstup ručiček určuje agresivitu.

Inverzni strankovani

Prevazne u 64bit. architektur - virtualni prostor by byl uz moc velky, tak se to organizuje pres ramce fyzicke pameti, ne pres stranky.

Problemy:

- Pristupuje se na virtualni adresu:
  - naivni reseni - linearni prohledavani polozek (protoze neumime rychle urcit ktery fyzicky ramec odpovida virtualni adrese)
  - efektivni reseni - pouzije se hashovani - nejdrive sahneme na hash table a tak zjistime kde hledat polozku v page table.
- Sdileni pameti:
  - Uz to nejde tak jako u primych page table - fyzicky ramec odkazuje na virtualni stranku jako 1-1. Pouzije se stejna virtualni adresa stranky ve vsech ASID (???). Anglicky receno: " ... allow the page table to contain only one mapping of a virtual address to the shared physical address.".

Segmentace

Program je rozdělen na segmenty (tj. části programu se stejnými vlastnostmi (kód, data, zásobníky)). Paměť je taky rozdělena na segmenty. Program přistupuje na adresy rozdělené na (segment/offset). Segmenty mohou být různě velké, a kromě umístení ve fyzické paměti mají i příznaky co na ně může sahat. Segmenty jdou přesouvat i zvětšovat aniž by o tom aplikace musela nezbytně vědět (stačí je jinak přeházet v paměti, číslo segmentu i offset v kódu se nemění). Při vypadku je nutne vymenit cely segment (ten muze byt velky). Prevod adresy se deje segmentovou tabulkou.

Stránkování + segmentace: Kazdy proces ma iluzi pro sebe vyhrazeneho prostoru - segmentu. Pro hledani fyzicke stranky se pak pouziva i cislo segmentu prirazene procesu. MMU nejdrive v tabulce segmentu najde, kde zacina tabulka stranek pro dany segment a pak se pokracuje jako vyse.

Vstupne-výstupní subsystémy

K I/O zarízením (subsystémům) je možné pristupovat dvema zpusoby: pomocí **portu** (speciální adresový prostor CPU) nebo **pametovým mapováním** (namapování do fyzické pameti). Abstrakce - většinou se programátorovi zarízení jeví jako stream nebo jako soubor. I/O zarízení mají ruzné charakteristiky:

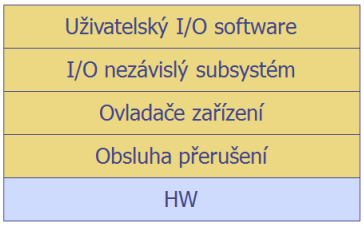
- **Druh** – **blokové** (disk, sítová karta), **znakové** (klávesnice, myš)
- **Prístup** – **sekvencní** (datová páska), **náhodný** (hdd, cd)
- **Komunikace** – **synchronní** (pracuje s daty na žádost – disk), **asynchronní** („nevyžádaná“ data – sítová karta)
- **Sdílení** – **sdílené** (preemptivní, lze odebrat – sítová karta (po multiplexu OS)), **vyhrazené** (nepreemptivní – tiskárna, sdílení se realizuje pres spooling). Reálne se rozdíly stírají.
- **Rychlost** (od nekolika Bps po GBps)
- **Smer dat** – R/W, R/O (CD-ROM), W/O (tiskárna)

Cíle I/O software

- **Nezávislost zarízení** – programy nemusí dopredu vedet, s jakým presne zarízením budou pracovat – je jedno jestli pracuji se souborem na pevném disku, diskete nebo na CD-ROM.
- **Jednotné pojmenování** - (na UNIXu /dev)
- **Pripojení (mount)** – casté u vymenitelných zarízení (disketa); možné i u pevných zarízení (disk); nutné pro správnou funkci cache OS.
- **Obsluha chyb** – v mnoha prípadech oprava bez vedomí uživatele (velmi často zpusobeno práve uživatelem).

I/O software

Je typicky rozdělený na 4 části (ve 4 vrstvách).



- **Ovladač zařízení** - pracuje s porty nebo s pamětí. „Rozumí“ konkrétnímu zarízení. Vytváří aspoň trochu jednotné rozhraní pro vyšší vrstvy. Typicky pouze výrobce ví, co zarízení (skutečně) dělá, a proto ovladače zarízení poskytují jednotliví výrobci.
- **Jednotné rozhraní na zarízení** - poskytuje uživatleským programům jednotné rozhraní. Skrývá rozdíly mezi jednotlivými zarízeními. Provádí pomocné úkoly pro ovladače: alokace paměti, hlášení chyb, vyrovnávání (buffering).

Prerušení

Přerušení je událost, která mění sekvenci, ve které CPU vykonává instrukce. Je generováno HW nebo SW. Je pouzivano pro praci s I/O, virtualni pameti, DMA,... Přerušení lze generovat:

- **Synchronně (trap)** - vnitrni událost, vyvolala ho instrukce v běžícím programu, např. výpadek stránky.
- **Asynchronně (interrupt)** - vnější událost, např. zasah uzivatele, porucha HW.
- **Výjimkou (exception)** - vnitrni událost, nesprávné chování programu, např. dělení nulou.

Průběh při přerušení:

1. Vznik příčiny přerušení, nebo vyslání žádosti o přerušení.
2. Rozhodnutí o přijetí či nepřijetí - přerušení lze maskovat - přerušení se případně uplatní až později. Poslat verdikt řadiči přerušení,
3. Identifikace zdroje přerušení (může být čistě programová, nebo s pomocí technických prostředků - žadatel poskytne vektor přerušení, podle něj se rozhodne o obslužném programu) a určení adresy obslužného programu - rutina, ktera vyresi duvod přerušeni.
4. Úschova aktuálního stavu procesoru na zásobník - je možno povolit víceúrovňové přerušení, ale je nutno zabránit zacyklení. Procesor obvykle při přijetí přerušení maskuje všechna přerušení.
5. Provedení obslužného programu. Obvykle je i přeplánován aktuální proces.
6. Obnova stavu procesoru.
7. Návrat do přerušeného programu. Obvykle pomocí zvláštní strojové instrukce, která obnoví to, co CPU uklidil.

Programové přerušení se využívá ke zpřístupnění služeb OS, instrukce call se nepoužívá kvůli ochraně. Vstupní body jsou při volání přes přerušení pevně dány.

DMA (Direct Memory Access)

Je způsob rychlého přenosu dat bez ucasti procesoru (ten k němu akorát dá pokyn a na konci dostane přerušením info že už je hotovo) při I/O operacích přesunech dat v paměti, zobrazování, refreshi... Přenos není řízen strojovými instrukcemi, ale řadičem DMA (nebo si to řídí zařízení samo). Přenos probíhá přímo mezi pamětí a I/O zařízením, procesor je odpojen od sběrnic. Procesor se tak nejen nemusí otravovat s kopírováním, ale ani nemusí na zařízení čekat.

Jak se řadič DMA a procesor dohodnou, kdy provést přenos?:

- **Dávkový režim -** řadič o sběrnice žádá, po dobu přenosu je procesor odstaven.
- **Kradení cyklů -** řadič je schopen uspat procesor, když nepracuje se sběrnicemi a provést přenos; náročné na hardware.
- **Transparentní režim** - řadič pozná, kdy procesor nepracuje se sběrnicemi, a v této době provede přenos - procesor o tom ani neví.

Dnes už jsou sběrnice pro procesor/hlavní pamět (northbridge) a periferie (southbridge) oddělené.

Zpusoby obsluhy periferií

Metody obsluhy periferií:

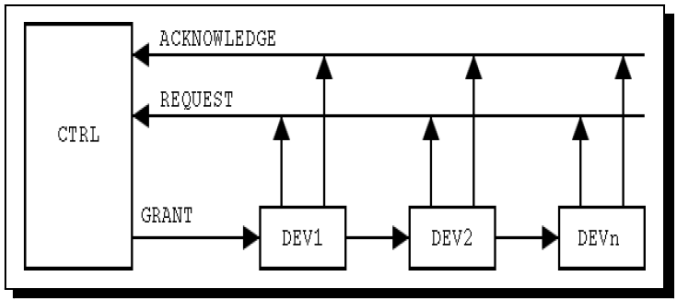
- **Polling** - periodické dotazování na stav zařízení. Tj. ovladač aktivně čeká (dotazuje se) na změnu stavu zařízení. Přenos dat je pak úkolem CPU.
- **Přerušení** - zarízení signalizuje procesoru zmenu stavu (většinou skrze řadič přerušení), obsluzná rutina prerusení reaguje na stav zarízení. Přenos dat je opět úkolem CPU.
- **DMA** - to je rozebráno výše samostatně.

Typy přístupu k periferiím:

- **Adresované (Memmory mapped)** - jeví se jako nějaká část paměti
  - Přímé - adresa je součást instrukce.
  - Nepřímé - součástí instrukce je adresa registru, kde je adresa dat.
- **Port moped** - k přístupu se používá specielní instrukce CPU.
  - Intel 80x86 IN and OUT instrukce.

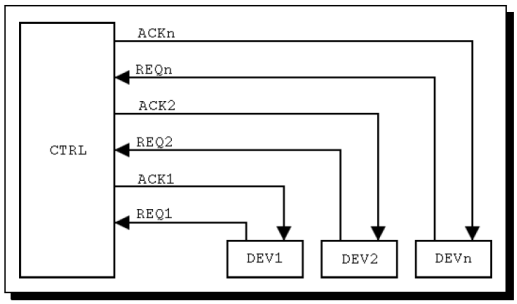
Vstupně-výstupní topologie.

- **Daisy chain** (řetězec počítač -- zařízení -- další zařízení -- poslední zařízení).
  - SCSI se tak fyzicky zapojuje, i když elektricky je to sběrnice
  - MIDI má takovou formu

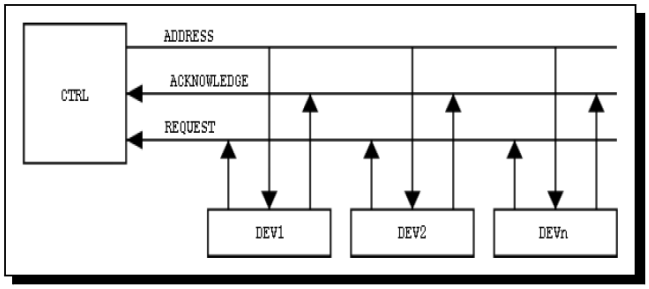


- **Sběrnice** (všechno visí na jednom drátu)

Nezávislé žádosti



Cyklické výzvy (polling)



- **Stromová struktura** (USB/HUB)

Sběrnice a jejich řízení (např. SCSI, USB, AGP, ...)

Sběrnice je zarizeni pro propojeni vice jinych zarizeni. Jedna se o sdilene medium, musi nejak resit konflikty. Jde hlavně o to, aby byla nizka cena pridani zarizeni a zároveň účinost komunikace vysoká. Sběrnice je také potenciální úzké hrdlo v systému.

Sériová komunikace nebo **sériový přenos** je v telekomunikacích a informatice proces přenosu dat postupně po jednotlivých bitech (tj. sekvenčně) pomocí komunikačního kanálu nebo sběrnice. Je v přímém protikladu s **paralelní komunikací**, kde je několik bitů posíláno najednou (linkou obsahující několik paralelních přenosových kanálů). Na první pohled by se mohlo zdát, že paralelní kanálly musí být rychlejší než sériové, protože dokáže posílat víc dat na jedno tiknutí hodin. Opak je však pravdou, protože hodiny sériových sběrnic je možné taktovat na omnoho vyšší frekvence kvůli několika faktorům:

- Nemusí se starat o souběh hodin na několika datových kanálech zároveň.
- Nedochází k rušení jedného dátového kanálu druhým kanálem.

Každá sběrnice (sériová i paralelní) typicky obsahuje:

- **Datove linky** – (data) prenasi data .
- **Adresove linky** – (adress) kam ty data maji jit.
- **Ridici linky** – (control) rezie prenosu.

Sbernicovy protokol

Definuje povolený prubeh transakcí na sbernici a logický význam signálu na vodicích. Standard sběrnice popisuje také elektrické a mechanické parametry sběrnice. Dva základné druhy přenosu dat:

- **Synchronni prenos (protokol)** - podle predem danych hodin.
- **Asynchronni prenos (protokol)** - je poslana zprava "zacatek zpravy, poslouchej", ktera se pouzije pro synchronizaci.

Řízení požadavků

Sběrnice je sdílené médium a přístup k ňemu (požadavky) musí být řízen, resp. řídit se nějakými pravidly, aby nedocházelo ke konfliktům. Existuje několik druhů přístupu:

- Centrální – centrální autorita přiděluje slovo. Ptá se jednotlivých zařízení, jestli nechcou vysílat. Právo ke slovu pak dostávají zařízení dle:
  - Náhody – autorita z času na čas přidělí zařízení slovo.
  - Dle pořadí vzniku – vysílá ten kdo první chtěl.
  - Prioritní – za˘˘řízení mají priority. Podle toho vysílají.
- Distribuované – existuje postup (algoritmus), který dodržují všechna zařízení.
  - Kolizní (CSMA/CD)
  - Token bus
  - Prioritní linka (daisy chain)

Prenos dat po sběrnici

- S ucasti procesoru (zdroj -> cpu -> cíl)
- Bez ucasti procesoru - rizeno radicem (napr. DMA)

Seriove sbernice

Prenasi jen jeden bit "najednou" (pripadne pokud se pouziva nejake slozitejsi kodovani, tak vice bitu informace ve vice prenesenych). Pro přenos dat a řízení používá sběrnice jeden vodič (resp. dvojici signál-nulový vodič) nebo více vodičů. Po fyzikální stránce se datová informace přenáší buď pomocí změny elektrického napětí nebo změny elektrického proudu. Realizace pomocí změny napětí je jednodušší, pomocí změny proudu je složitější, ale má větší odolnost proti elektromagnetickému rušení. Data jsou většinou přenášena v sériové posloupnosti pomocí jednoho signálu. Řízení sběrnice, je buď realizováno pomocí samostatných signálových vodičů, nebo je společně s daty přenášeno pomocí jednoho signálu.

Vyhody - jednodussi, nemusi synchronizovat mezi jednotlivymi kanaly, mensi naroky na místo (úspornejšia kabeláž). Diky tomu lze dosahovat znacne vyssich prenosovych rychlosti na jeden kanal. V soucasnosti zacinaji dominovat. Nektere (PCIe) pouzivaji full-duplex pointtopoint spojeni, cimz eliminuji kolize.

Vyhody novych seriovych sbernic: nenastavuji se piny, zarizeni napajeno ze sbernice, jeden typ kabelu, mnoho pripojitelnych zarizeni.

Zastupci: USB, SATA, PCI-express, FireWire

USB (Universal Serial Bus)

- 12 Mb/s (USB 1.0)
- 480 Mb/s (USB 2.0)
- Control, Bulk, Interrupt, Isochronous prenosy
- Stromová struktura - koncová zarízení / hub
- plug-and-play

Nové univerzální pro připojování periferií k počítačům. Jednotný nástupce sériových RS-232-C i paralelních (Centronics) portů pro myši a klávesnice. Vyvinuto fimamy DEC, IBM, Intel, Microsoft, NEC a nortel v roce 1996. Nyní udržuje sdružení USB Implementers forum.

USB 1.0

Určeno pro jednoduché připojování počítačových periferií (myší, klávesnic, tokenů, externých pamětí) a spotřební elektroniky. Umožňuje napájet zařízení přímo z USB sběrnice (jen zařízení s nízkým příkonem). Umožňuje hot-swap, tj. připojování a odpojovaní za chodu. Umožňuje připojit až 127 zařízení k jedné sběrnici, při čemž běžná PC mají v současnosti minimálně 2 USB sběrnice. Celková přenosová kapacita sběrnice je až **12 Mbps** (jedno zařízení si může alokovat až 6 Mbps). Zařízení se buď přípojí přímo k USB zběrnici (USB portu počítače), nebo skrze rozbočovač (hub) stylem do stromu.

USB kabeláž a rozbočovače (hub)

Kabely mají 4 vodiče: **dva pro napájení** (zařízení s nízkím odběrem mohou být napájena přímo z USB sběrnice (myši, klávesnice...). Zařízení s větším odběrem se musí napájet samostatně (tiskárny, vypalovačky)) a **dva signálové**.

Kebely **nesmí být delší jak 5 metrů**. Kabely lze řetezit pomocí rozbočovačů (hub-ů), avšak mezi počítačem (host-em) a koncovým zařízením nesmí být více jak 6 kabelových segmentů, tj. 30 metrů.

USB hub-y mohou být **samostatné**, **embedded** (zabudována např. v monitoru, klávesnici) a **na kartě** (PCI, PCMCIA…). Mohou být taky **pasivní** (bez napájení), nebo **aktivní** (s vlastním napájením).

Režimy fungování USB sběrnice

Při startu systému nebo při zapojení zařízení počítač (host) přiřadí každému zařízení identifikátor (bus ID) a vyptá se každého zařízení co je zač a v jakém režimu chce komunikovat. Možné režimy komunikace jsou:

- **Řídící (control mode)** – používá hlavně počítač k inicializaci.
- **Režim přerušení (interrupt mode)** – používají zařízení, která mají malé objemy dat k přenesení (myši, klávesnice).
- **Dávkový režim (bulk mode)** – pro zařízení, která potřebují přenášet velké objemy dat se 100% spolehlivostí (externí disky, kamery, PDA…).
- **Izochronní režim** – pro zařízení, která potřebují komunikovat v reálném čase, ale dokáží tolerovat určité stráty dat (audio, video zařízení).

Komunikace je iniciována počítačem (host-em) skrze tzv. **pooling** (počítač se ptá každého zařízení zda nemá nějaká data k přenesení). Existuje i výjimka : zařízení může vyslat žádost (**remote wakeup)** počítači. Isochronní režim a režim přerušení mají vyhrazeny určité časové sloty na sběrnici již při své inicializaci. Takto může být přiděleno až 90% kapacity sběrnice – když by se chtěla připojovat zařízení s dalšími požadavky, budou odmítnuta. V isochronním režimu jsou data přenášena pravidelně, např. každých 40 ms. V režimu přerušení jsou zařízení pravidelně dotazována (pooling). Zařízení pracující v dávkomvém režimu dostávají veškerou kapacitu, která momentálně zbývá. Faktická rychlost přenosu je teda závislá na počtu momentálně aktivních zařízení.

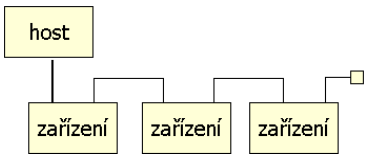
Paralerni sbernice

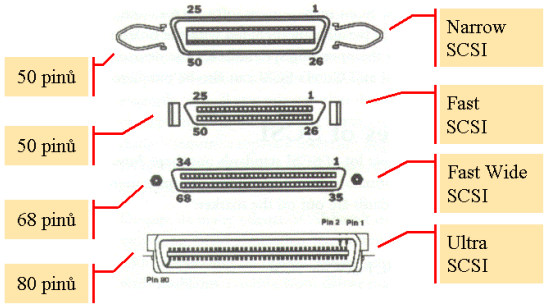
Prenasi se najednou vice bitu.
Zastupci: AGP, ISA, PCI, ATA (aka IDE), SCSI.

SCSI (Small Computer System Interface)

- Nejcastej pripojeni disku
- Paralelní rozhraní, ANSI standard od r. 1986
- 8/16 bit data, a. 640MB/s (160MHz, DDR)
- 15 zarízení na 1 sbernici, podpora hot-plug
- Inteligentní zarízení
- Podmozinu príkazu pouzívá i ATAPI

Vyvinula firma Shugart Associates v r. 1981. Je ANSI standardem od roku 1986. Používá se pro připojování rychlých periferií (disky, CD ROM, scannery, rychlé tikárny…). Jde o paralelní rozhraní šířky 8 nebo 16 bitů (resp. 8 nebo 16 datových drátů). Uzly jsou připojeny do řetězce (sběrnice) **daisy chain** a poslední musí být zakončen (tzv. **zaterminován**). Každé zařízení na sběrnici má 3-místní číselné ID.





AGP (Advanced Graphics Port)

- dvoubodový spoj
- 4x frekvence PCI, tj. 133MHz
- pro prenos dat vyuzita nábezná i sestupná hrana
- pipelining

Je v informačních technologiích speciální patice pro připojení grafické karty. Nejedná se v pravém slova smyslu o sběrnici, protože AGP lze připojit **pouze jedno zařízení**. Jedná se o spojení typu **point-to-point**, které je určeno pro porty k připojení externích zařízení, zejména pro připojení grafických adaptérů a grafických akcelerátorů. Z důvodu zobrazení videa ve velkém rozlišení byl při návrhu AGP kladen důraz na co nejvyšší rychlost přenosu dat **s malou latencí**. Technologie AGP **vznikla úpravou sběrnice PCI**, takže některé řídící signály jsou podobné (adresová a datová část AGP používá stejné vodiče pro přenos jako PCI). Došlo k několika modifikacím, z nichž nejvýraznější je **odstranění arbitrážního obvodu**. Z toho plyne, že je možné připojit pouze jedno zařízeni.

Architektura AGP

Poskytuje rychlý vyhrazený kanál mezi slotem a procesorem, proti sdílené PCI sběrnici. Umožňuje **sideband addressing**, který využívá dalších 8 adresových vodičů. Pomocí sidebandu je možné realizovat jinou techniku adresování, než umožňovala sběrnice PCI. K dispozici je však rovněž 32 hlavních adresových vodičů. Grafické karty na PCI musí načítat textury ze systémové RAM do grafické paměti (framebuffer). AGP karty jsou schopné číst textury přímo z operační paměti pomocí **GARTB** (Graphics Adress Remapping Table), ve kterém jsou vedeny informace o uložení a funkci stránek paměti. V případě, že jsou data z nějaké stránky vyžadována pro vykreslování a stránka je uložena v operační paměti, je proveden přenos příslušné 4 kiB stránky do paměti grafického akcelerátoru, a následně se provede úprava záznamu v GART. Vzhledem k poměrně malé velikosti stránek tak nedochází k jednorázovému zatížení systému, protože se zátěž v čase rozloží.

Verze AGP

- **AGP 1×** – používá 32bitový kanál na 66 MHz umožňující maximální datový tok 266 MB/s, zdvojnásobený z 133 MB/s rychlosti sběrnice PCI 33 MHz / 32-bitů; 3.3 V.
- **AGP 2×** – používá 32bitový kanál běžící na 66 MHz (double pumped) na efektivních 133 MHz umožňující maximální přenosovou rychlost 533 MB/s; 3,3 V.
- **AGP 4×** – používající 32bitový kanál běžící na 66 MHz (quad pumped) na efektivních 266 MHz umožňující maximální přenosovou rychlost 1066 MB/s (1 GB/s); 1,5 V.
- **AGP 8×** – používá 32bitový kanál běžící na 66 MHz (8 bitů za takt) na efektivních 533 MHz umožňující maximální přenosovou rychlost 2133 MB/s (2 GB/s); 0,8 V.

Mezipočítačová komunikace

Prenos v základním pásmu (baseband, nemodulovaný prenos)

Jde o takový druh přenosu, při kterém je vstupní signál okamžitě převáděn na přenosové médium bez činnosti modulačního prvku. Tj. **přenáší se rovnou "data"** a přenášené bity se reprezentují napěťovými úrovněmi (H/L), nebo velikostí proudu (tzv. proudová smyčka), kdy prochází/neprochází el. proud. Přenos v základním pásmu **může být také kódovaný** (kódování Manchester, diferenciální Manchester) - jeden datový bit je "zakódován" do více změn přenášeného signálu. Přenosy v základním pásmu se používají "na krátkou vzdálenost" (např. v sítích LAN (Ethernet)).

Přenosu (v základním pásmu) navětší vzdálenosti brání to, že přenosové cesty nejsou nikdy ideální a vždy nějak negativně ovlivňují přenášený signál. Vykazují: **útlum** (zeslabuje přenášený signál), **zkreslení** (deformuje přenášený signál), **přeslech** („prolínání“ signálů z jiných vedení), **rušení**, **vyzařování**… Důsledek je, ze každá přenosová cesta přenáší některé signály lépe, jiné hůře. Záleží zejména na frekvenci přenášeného signálu a na povaze jeho změn.

Přenos v přeloženém pásmu (broadband, modulovaný přenos)

Idea: Přenášet takový signál, jaký projde přenosovou cestou nejlépe. V praxi: signál harmonického (sinusového, kosinusového) průběhu. Představuje tzv. **nosnou** (nosný signál). Nosný signál ještě sám nenese žádnou informaci. Data, určená k přenosu, se "naloží" na nosný signál a podle přenášených dat se mění některý (některé) z parametrů přenášeného nosného signálu. Jde o tzv. **modulaci**. "Modulování" je tedy (měnění) nosného signálu podle toho, jaká data se mají přenést. **Demodulace** - příjemce musí být schopen rozpoznat (rozlišit) změny nosného signálu, a z nich "získat" přenášená data.

Přenášený signál má vždy charakter analogové veličiny (proudu, napětí, světla …). Záleží na tom, jak vyhodnocuji (interpretuji) to, co je přeneseno.

Analogovy přenos - zajímá mne konkrétní hodnota přenášené veličiny (např. okamžitá hodnota napětí, proudu apod.). Analogový přenos není tedy nikdy ideální (nedokáže přenést hodnotu s ideální přesností).

Digitální přenos - zajímá mne, zda hodnota přenášené veličiny spadá do jednoho intervalu či do druhého intervalu. Digitální přenos je (může být) ideální.

Postup při digitalizaci analogového signalu: analogový signál se "vyvzorkuje" - sejmou se vzorky momentální hodnoty analogového signálu. Velikost každého (analogového) vzorku se vyjádří jako (digitální) číslo.

Prenosova media

Kroucená dvoulinka, twist

Každé dva vodiče, vedené souběžně vedle sebe, fungují jako anténa. Proto je nutne oba vodiče pravidelně (rovnoměrně) zkroutit (typicky 1x za každých 7,5 až 10 cm). Kategorie kroucené dvoulinky: **kategorie 3** (do 10 Mbit/s), **kategorie 5** (dnes nejpouzivanejsi (ethernet) do 100 –150 Mbit/s), **kategorie 6, 7**. Kabely typicky obsahují více kroucených párů. Kroucená dvoulinka umožňuje vytvářet pouze dvoubodové spoje - topologie musí byt stromovitá.

Koaxiální kabely

Koaxiální kabel tvoří dva soustředné (co-axialní) vodiče: **vnitřní (středový) vodič** a **vodivé opletení** (současně slouží jako stínění). Díky stínění méně vyzařuje, lze jej využít na větší vzdálenosti, lze využít na vyšších frekvencích (než kroucená dvoulinka), konstrukčně robustnější, odolnější. Je však málo ohebný a dražší než kroucená dvoulinka. Dříve se používal i v sítích LAN (Ethernet).

Optická vlákna

Optická vlákna mají stále obrovskou rezervu přenosové kapacity. Jejich možnosti jsou dnes využívány jen z malé části. Pracují s vysokými frekvencemi (viditelné světlo cca 10^8MHz). Nabízí obrovskou šířku přenosového pásma, a tedy dle Shannonova teorému mohou dosahovat velmi vysokých přenosových rychlostí. Velmi malý odpor / nízký útlum - dosah až desítky/stovky kilometrů. Nemaji žádné elektromagnetické vyzařování a nejsou citlive na vnější elektromagnetické rušení - lze je použít kdekoli. Díky tzv. vlnovému multiplexu (technologii WDM, resp. DWDM) lze jedno vlákno rozdělit na několik částí, využitelných pro samostatné přenosy. Jsou však pomerne drahe, křehke, mají malou mechanickou odolnost a konektorování je náročné.

Světlo se optickým vláknem šíří „ve svazcích“ tzv. **videch** (mode):

Mnohovidová vákna - přenáší více vidů současně. Přenosové schopnosti jsou obecně horší než u jednovidového vlákna (kratší dosah, nižší dosažitelná přenosová rychlost). Kabely jsou ale lacinější, konektorování je jednodušší a světlo stačí budit diodami LED.

Jednovidová vlákna- přenáší "užitečný signál" pomocí jediného vidu. Nemají zkreslení vznikající vidovou disperzí (jako mnohovidova vlakna), mají obecně větší dosah a umožňují dosahovat vyšší rychlosti.

Jádro optického vlákna je z čistého SiO_2 a je velmi křehké plášť (obalující jádro) je sám obalen izolační vrstvou (neprůsvitnou). Mechanické vlastnosti se zlepšují různým „vyztužováním“ (např. přidáním kovového drátku). Optické kabely obsahují desítky (až stovky) vláken. Kabely se instalují do trubek (tzv. chrániček). Optické sítě mají nejčastěji kruhovou topologii.

Optické vlákno zajišťuje pouze vedení světelného paprsku, nesoucího data je nutné ještě zajistit **zdroj (generování) světla** a **příjem (detekci) světla**. Pak jde o celý **optický přenosový systém**. Pro mnohovidová stačí LED dioda na straně zdroje

světla, a fotodioda či foto tranzistor na straně příjmu. Pro jednovidová vlákna musí být světlo generováno laserem.

Cistě optické systémy - veškeré zpracování probíhá optickou cestou, bez nutnosti převodu z/do elektronické podoby. Opticky opakovac: zesílení optického signálu pomoci zesilovače **EDFA (Erbium Doped Fiber Amplifier)** po dopadu světla se elektron vrací do své původní polohy a přitom je uvolněno světelné záření, které je silnější než byl počáteční podnět na uvolnění elektronu z metastabilní pozice. Jiné zpracování (změna směru, rozbočení atd.) se provadi odrazem paprsků od vhodně natočených zrcadlových ploch.

Bezdrátové (radiové) přenosy

Signál se šíří "volný prostorem" (éterem) prostřednictvím elektromagnetických vln. Rychlost jeho šíření je cca. 300 000 km/s. Omezená je však dostupnost frekvencí (jsou omezeným přírodním zdrojem → omezená přenosová kapacita), větší vliv prostředí (→rušení, interference, podmínky příjmu), větší zranitelnost (vůči odposlechu, útokům).

Parametry: **frekvence (kmitočet) f [Hz], perioda T [s] (f = 1/T), vlnová délka λ [m]**.

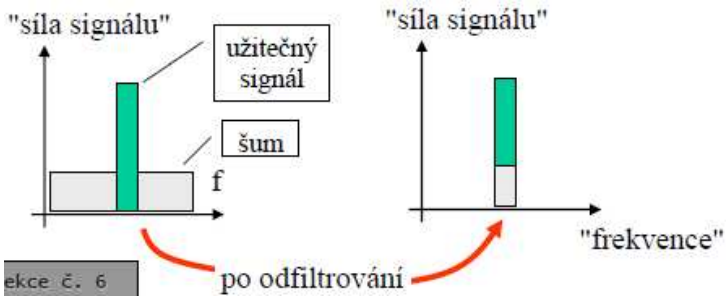
λ = c * T = c / f

Správcem kmitočtového spektra v ČR je ČTÚ (Český telekomunikační úřad), vydává "národní kmitočtovou tabulku" a určuje, jak (za jakých podminek) a kým smí být využity různé části frekvenčního spektra. Dva druhy frekvencii vo frekvencnom spektre (spektrum je rozdelene na dve pasma):

- **Licencni pasmo -** část frekvencí, jejichž využití vyžaduje licenci od ČTÚ. Přiděluje se na žádost, pokud je více zájemců pak v soutěži (výběrovém řízení). GSM (900 MHz, 1800 MHz), FWA (3,5 GHz, 26 GHz)
- **Bezlicencni pasmo -** Není nutná individuální licence (oprávnění). Podmínky využití jsou určeny tzv. generální licencí (dnes: všeobecné oprávnění). Určuje například přípustné vysílací výkony a další parametry.
  2,4 GHz (pro Wi-Fi 802.11b,g), 5 GHz (pro Wi-Fi802.11a,h)

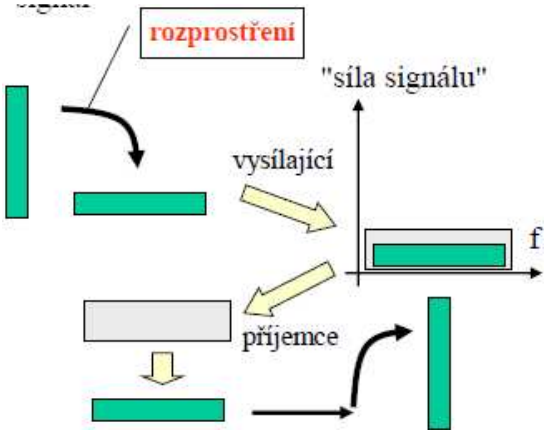
Vysílání v úzkém pásmu (Narrowband)

Vysílá se v úzkém rozsahu frekvencí. Energie vysílače je soustředěna do úzkého rozsahu frekvencí. Rušení (šum) je širokopásmové - rozprostřené do širšího spektra. Reší se dostatečným odstupem signálu od šumu.



Vysílání v rozprostřeném pásmu (Spread Spectrum)

Vysílá se v širokém rozsahu frekvencí. Energie vysílače může být stejná, ale je rozprostřena do širšího rozsahu frekvencí. "Síla signálu" nemusí být vyšší než "síla šumu". Důležité je, aby příjemce dokázal z přijatého signálu extrahovat "užitečný signál".



Techniky vysílání v rozprostřeném spektru

Frequency Hopping (s kmitočtovým skákáním nosné) - vysílá se na (úzkopásmové) nosné frekvenci, která se ale pravidelně přelaďuje, podle (vhodně volené) pseudonáhodné posloupnosti kterou musí znát vysílač i přijímač. Může dojít k "souběhu" více vysílání na stejné frekvenci (a ke vzájemnému rušení) ale je to krátké a lze se z toho zotavit. IEEE 802.11 přeskakuje 2,5x za sekundu, Bluetooth 1600x za sekundu.

Direct Sequence Spread Spectrum (s přímou modulací kódovou posloupností) - Vysílá se sekvence bitů (tzv. chippingcode, chip, „úlomek“) o vyšší modulační rychlosti (zabírá větší šířku pásma). Na ni se modulují (pomocí XOR) přenášená data. Místo 1 "užitečného bitu" se vyšle n pseudonáhodných bitů (tzv. 1 chip, "úlomek"), buďto v základním tvaru nebo invertovaný (XOR).

Vysílač místo 1 bitu vyšle n bitů, kde n je šířka tzv. chipu (úlomku). Je-li je chiping kód roven [c1, c2, c3, c4, c5, c6], tak v pripade bipolárního chipping kodu pro 1 vyšle [c1, c2, c3, c4, c5, c6] a pro 0 vyšle [-c1, -c2, -c3, -c4, -c5, -c6].

Příjemce přijme celý chip [d1, d2, d3, d4, d5, d6] a aplikuje na přijatý "úlomek" (chip) chipping kód odesilatele, udělá s ním XOR:

$\sum d_i \; XOR \; c_i = 6$, pokud byla přenášena 1

$\sum d_i \; XOR \; c_i = -6$, pokud byla přenášena 0

Při zarušeném signálu mohou být součty jiné. Přijímač vyhodnocuje 0 nebo 1 podle toho, zda $\Sigma > 0$ (přijme 1), $\Sigma < 0$ (přijme 0).

Multiplex – Jde o to, jak jeden (širší) přenosový kanál rozdělit na několik (užších, menších) přenosových kanálů, které bude možné využívat samostatně a nezávisle na sobě. Rozdělení větší přenosové kapacity na několik menších kousků.

Inverzní multiplex - jde o to, jak několik (menších, užších) přenosových kanálů sdružit do jednoho celku, aby se choval jako jeden (širší, větší) přenosový kanál. Nejčastější technika implementace je channel bundling (souběžné použití více kanálů).

Analogové techniky multiplexu

Frekvenční multiplex (FDM, Frequency Division Multiplexing) - signály jednotlivých kanálů jsou posunuty do vhodných frekvenčních poloh a „poskládány“ do jednoho širšího přenosového pásma. Jednotlivé složky jsou „vyextrahovány“ a vráceny do původní frekvenční polohy.

Vlnový multiplex (WDM, Wavelength Division Multiplexing) - týká se vedení světla skrze optická vlákna. Dříve bylo možné použít optické vlákno pouze k 1 přenosu (všemi "barvami" současně). Dnes již může být každá barva přenášena samostatně (může přenášet samostatná data a tím představovat samostatný přenosový kanál). Celková přenosová kapacita optického vlákna se tím násobí počtem barev - v praxi desítky až stovky barev současně.

Digitální techniky multiplexu

Casový multiplex (TDM, Time Division Multiplexing) - přenosová cesta se rozdělí v čase na "časová okna" (time sloty) a ty se napevno přiřadí jednotlivým vstupům. Během každého časového okna se celá přenosová cesta věnuje výhradně přenosu dat z daného vstupu. Celková přenosová kapacity se tak dělí v poměru, v jakém jsou rozdělena jednotlivá časová okna. Rozdělení časových oken mezi jednotlivé vstupy nemusí být rovnoměrné. **Rozdělení slotů mezi jednotlivé vstupy je pevné a je dáno předem!** Proto nemusí být přenášená data opatřena žádným identifikátorem (hlavičkou). Technika je vhodná tam, kde jednotlivé kanály (vstupy) produkují rovnoměrnou zátěž.

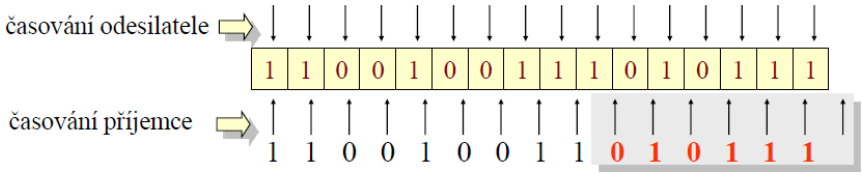
Statistický multiplex (STDM, Statistical TDM) - je vhodný pro nestejnoměrnou zátěž. Nepřiřazuje časové sloty jednotlivým kanálům pevně, ale až na základě skutečné potřeby. Každý "kus dat", který je přenášen v časovém okně, musí sám sebe identifikovat (**musí mít hlavičku**). Jde v zásadě už o variantu paketového přenosu.

Kódový multiplex (CDM, Code Division Multiplexing, CDMA) - každý zdroj (odesilatel) vysílá v celé dostupné šířce pásma. Je to zařízeno tak, že jednotlivá vysílání se vzájemně neruší, ale lze je opět oddělit. Každý vysílač vysílá v rozprostřeném spektru technikou Direct Sequence. Pseudonáhodné posloupnosti (chipping kódy) jednotlivých vysílačů musí být různé a **vzájemně ortogonální**. Každý příjemce má možnost přijímat vysílání všech vysílačů a je schopen si z toho vybrat právě a pouze ta vysílání, která potřebuje. Je to maximálně efektivní technika co do využití frekvenčního spektra.

- **Duplexní přenos -** je možný v obou směrech, a to současně.
- **Poloduplexní přenos -** je možný v obou směrech, ale nikoli současně.
- **Simplexní přenos -** je možný jen v jednom směru.
- **Semiduplex (dusimplex) -** přenos je každým z obou směrů realizován jinak.
- **Asymetrický přenos -** (maximální, nominální) rychlosti jsou v obou směrech různé.

Problematika synchronizace - každý bit je přenášen v rámci určitého bitového intervalu. Tj. přenos bitu není "okamžitý", ale trvá určitou dobu (bitový interval) a přenášená data reprezentuje stav signálu během bitového intervalu. Příjemce vyhodnocuje stav přenášeného signálu "někde" v rámci bitového intervalu. Rozhodující je okamžik vyhodnocení signálu. Na základě vyhodnocení okamžitého stavu signálu pak usuzuje to, jaká data jsou přenášena. **Příjemce se musí "strefit" do správného bitového intervalu jinak přijme nesmyslná data**.

Odesilatel i příjemce odměřují odesílaná data podle vlastních hodinek tyto hodinky musí "tikat" dostatečně souběžně (**synchronně - musí být tzv. v synchronizaci**).



Asynchronní přenos - Je jeden z možných způsobů zajištění synchronizace. Zcela postrádá jakoukoli synchronizaci. Bitový interval nemá konstantní délku. Začátek i konec každého bitového intervalu musí být explicitně vyznačen.

Arytmický přenos - snaží se přenášet celé skupiny bitů, tvořící tzv. znaky. Na začátku každého znaku je tzv. start-bit. Slouží k tomu, aby si příjemce "seřídil své hodinky". Predpoklad: po seřízení na začátku každého znaku budou hodinky příjemce "tikat" po celou dobu trvání daného znaku. Casové prodlevy mezi jednotlivými znaky (skupinami bitu) mohou být různě velké a během prodlevy mezi znaky se hodinky příjemce mohou libovolně "rozejít" (na začátku dalšího znaku budou znovu "zasynchronizovány" pomocí start bitu).

Synchronní přenos - synchronizace je udržována trvale. Přenáší se celé, souvislé a libovolně velké bloky dat. Synchronizace se udržuje po celou dobu přenosu souvislého bloku skrze **samostatný "synchronizační" (časovací) signál** (přenáší "tikání" hodinek odesilatele), **skrze redundantní kódování** (zahrnutí časového signálu do kódování jednotlivých bitů - kódování Manchester) a **synchronizací z dat**. Synchronní přenos je obecně rychlejší než asynchronní a arytmický.

Redundantni kodovani - kódování Manchester u Ethernetu: uprostřed každého bitového intervalu je vždy hrana, která "nese" data (svou polaritou). Současně tato

hrana může sloužit i pro potřeby synchronizace, vyskytuje se vždy, bez ohledu na hodnotu dat.

Synchronizace z dat - přenášený signál neobsahuje žádné časování. Příjemce si průběžně seřizuje hodinky podle datových bitů. Mohou se vyskytnout dlouhé posloupnosti bitů, které negenerují žádnou změnu přenášeného signálu a hodinky příjemce by mohly ztratit synchronizaci. Technika **Bit-stuffing (vkládání bitů) -** pokud by se vyskytla příliš dlouhá sekvence bitů, která by mohla způsobit ztrátu synchronizace, odesilatel vloží do odesílaných dat vhodný bit, který vyvolá hranu a příjemce ji zase odstraní.

Sériové a paralelní kanály

Sériové komunikace zahrnují problematiku připojování periferií k počítačům přes sériová rozhrání (rozhrání RS-232-C, USB, FireWire), přenosy dat po cestách fungujících na principu přepojování okruhů a využití veřejné telefonní sítě pro potřeby „počítačových“ přenosů (modemy a modemové komunikace, komunikační programy).

POTS (Plain Old Telephone System)

První telefonní hovor 10. března, 1876, Boston, Massachussets. 1891 první automatická telefonní ústředna typu "Strowger" pracovala na principu "step-by-step" - krokový přepínač se postupně přepínal na jednotlivé kontakty a tím vznikalo spojení.

Přepojováním na ústředně docházelo ke galvanickému spojení místních smyček, k tzv. **komutaci** (přepojování) a tím ke vzniku souvislé (vodivé) přenosové cesty mezi komunikujícími stranami. Dnes se realizuje elektronicky, ne mechanicky.

Struktura POTS sítě

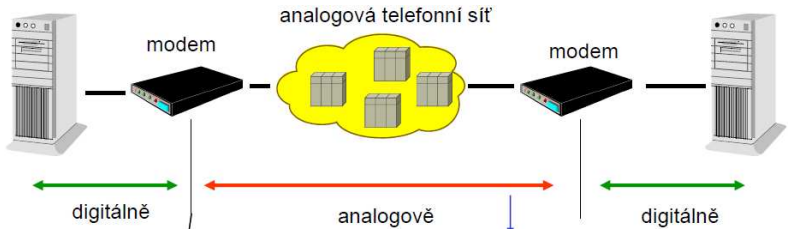
Struktura dnešní (veřejné) telefonní sítě je typicky hierarchická. Nejvýš (v hierarchii) jsou tranzitní ústředny, místní ústředny a nakonec pobočkové ústředny (často privátní). Ústředny jsou vzájemně propojeny přenosovými cestami, které přenáší více hovorů současně.

V **analogové telefonní** síti jsou jednotlivé hovory "skládány" do jedné přenosové cesty pomocí techniky frekvenčního multiplexu. Čím užší je frekvenční pásmo jednotlivého hovoru, tím více hovorů se vejde na existující vzájemné propojení ústředen a proto je snaha omezit frekvenční pásmo hovorů. Toto omezení je realizováno na vstupu do ústředny. Důsledky: **je omezena i dosažitelná přenosová rychlost při přenosu dat po (komutované) telefonní síti**.

Telefonní ústředny fungují plně digitálně, tj. **uvnitř** (přepojování je digitální) a **mezi sebou** (hovory v digitálním tvaru se "slučují" pomocí technik časového multiplexu, případně statistického multiplexu. Místní smyčky fungují stále analogově, protože telefonní přístroje jsou (stále stejně) analogové a k převodu z/do digitálního tvaru dochází na vstupu do ústředny.

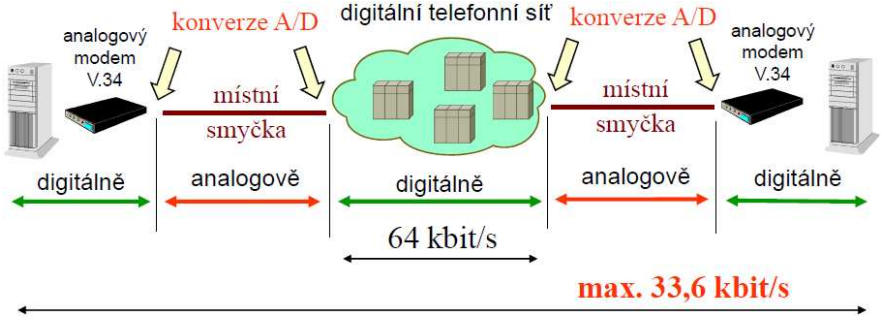
Přenos dat v analogové tel. síti

Telefonní síť se chová jako čistě analogový přenosový kanál. Konverze z/do digitální podoby je zajišťována mimo telefonní síť – pomocí modemů, které patří komunikujícím stranám. Na nich také závisí max. dosahovaná rychlost **do horní hranice 33,6 kbit/s**. Telefonní síť nepozoruje rozdíl mezi hlasovým hovorem a přenosem dat.



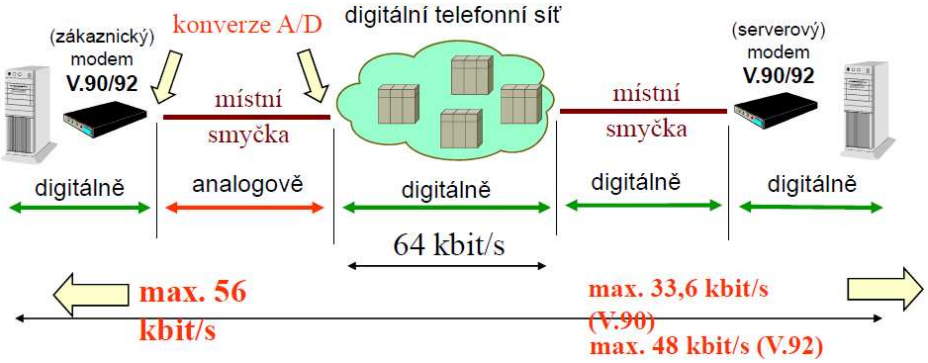
Analogový přenos dat v digitální tel. síti

Telefonní síť se interně chová jako čistě digitální přenosový kanál, ale "externě" stále jako analogový kanál. Příslušnou konverzi si zajišťuje sama, na vstupech do sítě (do ústředny). Pro datové přenosy je to velmi omezující (interně se využívá 64 kbit/s a externě max. 33,6 kbit/s).



Digitální přenos dat v digitální tel. síti

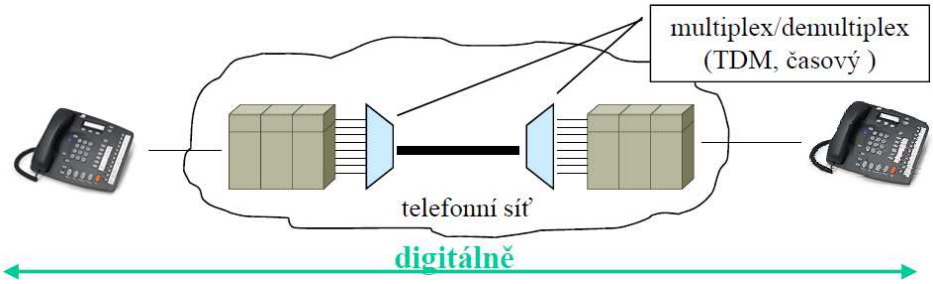
Odstraněním jedné "sady" konverzí lze dosáhnout vyšší přenosové rychlosti. Stále ještě ne 64 kbit/s, ale asymetricky: 56 kbit/s směrem uživateli. Důsledek: modemy nejsou "symetrické", liší se "serverové" a "klientské".



Jestli má přenosový kanál veřejné telefonní sítě má uměle omezenou šířku přenosového pásma - na 3,1 kHz, potom dle Shannona: lze na tom dosáhnout max. cca 30 kbit/s. Existují modemy (standard V.34) pro 33,6 kbit/s které využívají „o něco širší“ přenosové pásmo. Existují modemy (standard V.90) pro 56 kbit/s, které fungují sice analogově, ale jen „proti“ digitálním ústřednám. Modemy jsou programovatelné - lze jim nastavit mnoho parametrů pomocí příkazů AT jazyka.

ISDN síť (Integrated Services Digital Network)

Digitální způsob fungování telefonní sítě se prodlouží až ke koncovým účastníkům. Telefonní ústředny fungují plně digitálně (uvnitř i mezi sebou) ale i místní smyčky fungují digitálně, a proto na konci musí být zařízení komunikující digitálně.



Když bude všechno fungovat digitálně, je možné přidat (integrovat) další služby, k základnímu telefonování (digitální síť s integrovanými službami). Představa „dalších“ služeb: různá přesměrování hovorů, průběžné zobrazování údajů o tarifech, identifikace volajícího, bohatší předvolby, datové přenosy na principu přepojování okruhů i přepojování paketů…

Digitalizace lidského hlasu (PCM, Pulse Coded Modulation)

Je třeba vzorkovat průběh signálu snímaného mikrofonem 8000x za sekundu, tj. 1x za 125 mikosekund (pokud se vezme šířka pásma 4 kHz). Technika PCM: velikost

každého vzorku se vyjádří pomocí 8 bitů (je to v zásadě PAM, pulsně amplitudová modulace). 8000 vzorků za sekundu, každý o 8 bitech = **64 000 bitů za sekundu**. Digitální telefonní sítě předpokládají právě tento způsob digitalizace hlasu (**64 kbit/s**).

Kanály a přípojky ISDN

ISDN počítá (hlavně) s existencí kanálů á 64 kbit/s, označovaných jako **B** (Bearer). Účastnické přípojky jsou řešeny jako kombinace kanálů B a dalších (služebních) kanálů: **A**-analogový kanál 4 kHz, **B**-digitální datový kanál 64 kbit/s, **C**-digitální kanál 8 nebo 16 kbit/s, **D**-digitální kanál 16 kbit/s nebo 64 kbit/s pro signalizaci.

Přípojka ISDN Basic Rate (euroISDN2) - je přípojka určená pro domácnosti. Obsahuje dva B kanály a jeden D kanál. „Předpokládá se, že je možné využít stávající účastnické přípojky (místní smyčky). Poskytuje možnost připojit k jedné přípojce více zařízení (max. 8). Komunikovat současně mohou 2 zařízení. Z hlediska telefonování je k dispozici více funkcí než u analogové linky (identifikace volajícího, přesměrování, konference) a z hlediska datových komunikací vyšší rychlost garantovaného charakteru: připojení k Internetu rychlostí 128 kbit/s (2x 64 kbit/s), skutečných 64 kbit/s k protistraně), velmi rychlé navazování spojení (cca 2 s). Princip zpoplatnění je však stejný jako u analogové linky, tj. samostatně za každý B kanál!

Přípojka Primary Rate (euroISDN30) - přípojka určená pro firmy, kanceláře, providery, operátory. Obsahuje 23x64 kbit/s datové kanály, 1x kanál D 64 kbit/s pro signalizaci. „Spotřeba“ přenosové kapacity je 1,5 Mbps – označené jako T1. v Evropě se používají jinak dimenzované přenosové okruhy místo T1 (1,544 Mbps) se používají okruhy E1 (2,048 Mbps). Tomu se přizpůsobuje i skladba B kanálů v přípojce Primary Rate (30 x B + 1 x D).

Technologie xDLS (Digital Subscriber Line)

Je snaha maximálně využít existující místní smyčky (účastnického vedení). Vychází z pozorování, že přenosový potenciál místních smyček je i v rámci ISDN využit jen minimálně protože nevyužívá celý potenciál přenosového média – kroucené dvoulinky (jen frekvenčí pásmo, kde se pohybuje lidský hlas). Na oba konce účastnického vedení se umístí speciální modemy, které se snaží maximálně využít přenosové schopnosti existující přípojky. Lze takto dosáhnout rychlostí až několika Mbps. Možnosti implementace xDSL technologií:

- **ADSL (Asymmetric DSL)** - využíva jen tzv. "nadhovorové pásmo" - pouze vyšší frekvence, nevyužívané pro přenos hlasu. Možnost přenosu hlasu (nebo ISDN) zůstane zachována. Vyšší rychlost směrem k uživateli. Hovorové a nadhovorové pásmo mohou dokonce využívat různí poskytovatelé služeb. Musí být zajištěno vhodné oddělení hovorového a nadhovorového pásma na principu frekvenčního multiplexu pomocí zařízení zvaného **splitter**.
- **SHDSL (Single pair High bit-rate DSL)** - využívá celé frekvenční pásmo - nebude možné souběžně přenášet i hlas. Symetrické řešení, tj. stejný UP/DOWN bitrate.

Technologie ADSL

Využívá pouze nadhovorové pásmo. Nižší frekvence nechává ADSL volné pro analogový přenos hlasu (300-3400 Hz) nebo ISDN. Max. rychlost směrem k uživateli (downstream) je 6 až 8 Mbps. Max. rychlost směrem od uživatele (upstream) je 600 až 800 kbit/s. Dosah záleží na kvalitě linky. Skutečně dosahovaná rychlost závisí také na délce místní smyčky - na vzdálenosti od ústředny.

Modulace v ADSL

- **DMT (Discrete Multi-Tone)** - využitelné frekvenční pásmo se rozdělí na určitý počet samostatných dílčích pásem - typicky na 256 tzv. subkanálů každý o šířce 4,3125 kHz. V každém pásmu (subkanálu) je přenášena jedna (samostatná) nosná (o modulační rychlosti 4 kBaudy). Na každou nosnou frekvenci jsou (samostatně) namodulována data pomocí QAM (kvadraturní amplitudové modulace QAM).
- **CAP (Carrierless amplitude/phase modulation)** - starší a dnes již nedoporučovaná technika modulace. Nerozděluje dostupné pásmo na části (kanály), ale využívá ho jako celek.
- **QAM (Quadrature Amplitude Modulation)** - používá (používala) se v USA.

Modemy

Modem je zarizeni pouzivane k prevodu digitalniho signalu na analogovy, jeho kodovani pro prenos pres metalicke medium a konverze zpet do digitalni podoby na druhe strane linky. **MO**dulator a **DEM**odulator, odtud modem. Modulovani je "nanaseni" signalu nesouciho informaci na nejakou nosnou vlnu, ktera se dobre siri mediem (prenos v preloženém pásmu). Samotna nosna (typicky sinusoida) nenese informaci. Informace se uklada az do zmen teto vlny tzv. modulovanim.

Pouziva se **frekvencni (w)**, **amplitudova (A)** a **fazova modulace (f)**. Presneji kombinace techto tri: **y = A * sin (w.t + f)**

Modulační rychlost (baud rate) - je počet změn signálu za sekundu. měří se v jednotkách zvaných **BAUD [Bd]**. Modulační rychlost nevypovídá nic o tom, kolik dat se přenáší (to záleží na tom, kolik bitů "nese" (reprezentuje) jedna změna signálu).

Vícestavová modulace - modulovaný (nosný) signál má určitý počet stavů (n). Pak každý z těchto stavů může reprezentovat $\log_2(n)$ bitů. Základní způsoby modulace obvykle nedokážídosáhnout nejvyšších přenosových rychlostí (přesněji: počtu rozlišitelných stavů). Proto se základní způsoby modulace kombinují (např. fázová a amplitudová). Počet stavů nelze libovolně zvyšovat, protože příjemce by je už nedokázal rozlišit. Nejlépe se rozpoznávají stavy u fázové modulace. Po jedné přenosové cestě lze přenášet více nosných (modulovaných) signálů současně musí jen být vhodně frekvenčně posunuty (jde o tzv. frekvenční multiplex).

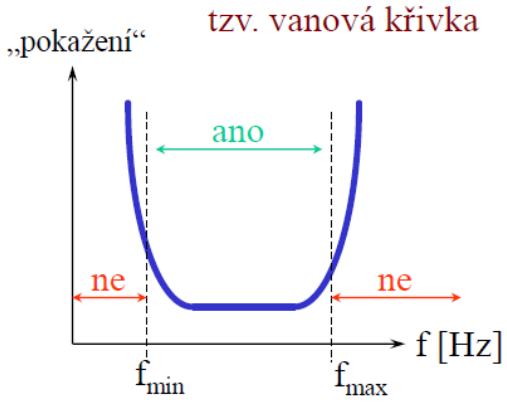
Napr. **QAM** - kvadraturni amplitudova modulace je kombinaci 3 ruznych amplitud a 12 fazovych posunu, coz tvori celkem 36 stavu. Pro snazsi rozpoznatelnost se ale pouziva jen 16 z nich s nejvyssi vzajemnou "vzdalenosti".

Prenosova rychlost (bit rate) = kolik bitu se prenese za sekundu. měří se v bitech za sekundu (resp. v násobcích –kbit/s, Mbit/s atd.). Efektivní (skutečně dosahovaná) přenosová rychlost může být i výrazně nižší (kolize, protokoly, …).

$v_{přenosová} = v_{modulační} * \log_2(n)$

Zvyšování přenosové rychlosti se tedy dosahne zvyšovanim n (počet stavů) nebo zvyšovanim modulační rychlosti (využívání více zdrojů, konkrétně tzv. šířky pásma – je to drahe). Otazka tedy zni jak dlouho lze zvyšovat počet (rozlišovaných) stavů?
→Shannonuv theorem.

Sířka přenosového pásma (bandwidth) - souvisí s reálnými (obvodovými) vlastnostmi přenosových cest. Závislost „míry pokažení přenášeného signálu“ má většinou intervalový charakter (závislý primárně na frekvenci signálu). Lze najít rozsah frekvencí (f_{min} až f_{max}), které daná přenosová cesta přenáší s ještě únosným „pokažením“. Hodnota $f_{max} - f_{min}$ představuje tzv. šířku přenosového pásma. Pro signály harmonického (sinusového) průběhuje závislost na šířce přenosového pásma zřejmá - pokud frekvence harmonického signálu spadá do šířky pásma, signál "projde" (beze změn), jinak neprojde vůbec.



Cím větší je šířka pásma, tím více je přijatý signál „podobný“ tomu, který byl odeslán a tím lépe lze poznat, co má reprezentovat. Při určité rychlosti změn by deformace přijatého signálu byly již tak velké, že by se nedalo poznat, co má signál reprezentovat.

Shannonuv theorem

Říká ze závislost maximalni přenosove rychlosti na šířce pásma je lineární. Nezáleží na použité modulaci a nevyskytuje se tam ani počet rozlišovaných stavů přenášeného signálu. Je to teda ona hranice maximalni prenosove rychlosti.

$\max(v_{prenosová}) = šírkaPásma * \log2(1 + signál/šum)$

Topologie sítí

Rozdeleni podle tvaru (topologie):

- **Sbernice** - Ethernet - co jeden vysila, slysi vsichni.
- **Kruh** - token ring, FDDI - kazdy vysle dal, co mu prislo.
- **Strom** - ethernet 10xbaseT, 100 VG AnyLAN.
- **Hvezda** - typ stromu s korenem a listy bez vnitrnich vrcholu.
- **Klika** - (kazdy s kazdym).
- **Retez** - (jednotlive clanky za sebou) - rozpojeny kruh.
- **Mesh** - (sit, propletenina, proste co nepatri do vyse zminenych, resp. asi vsechno) – internet.

Rozdeleni podle dosahu:

- **WAN (Wide Area Network, rozlehlé sítě) -** dosah na velkou vzdálenost.
- **MAN (Metropolitan Area Network, metropolitní sítě) -** jsou pokusem „zabydlet“ předěl mezi sítěmi LAN a WAN. Není úplný konsensus o tom, co jsou sítě MAN vlastně zač. Jsou to sítě v rozsahu města, sloužící potřebám města.
- **LAN (Local Area Network, lokální síť) -** dosah na krátkou vzdálenost. Technologie: Ethernet
- **PAN (Personal Area Network, "osobní" síť) -** sítě, které vznikají propojením "malých" (osobních) zařízení na krátkou vzdálenost (max. metry). Slouží potřebám jednoho uživatele. Například pro propojeni "stacionárních" (klávesnice, myš, tiskáren) a "mobilních" (mobilních telefonů, bezdrátových telefonů, PDA, hands-free) zařízení. Technologie: USB, Wi-Fi, Bluetooth, IrDA.
- **NAN (Neighbourhood Area Network) -** síť´ propojující uživatele na "regionálním" principu. Bydlí vedle sebe, jsou sousedé, řeší problém "posledního metru". Slouží ke sdílení přístupu k Internetu.
- **CAN (Community Area Network) -** propojuje uživatele příslušející k určité komunitě, která může být vymezena geograficky, profesně, zájmově. oproti sousedské síti bývá komunitní síť obvykle větší, ale hranice není ostrá.

Privátní (počítačová) síť - vlastníkem, provozovatelem i uživatelem je tentýž subjekt i když některé části (např. přenosové trasy) mohou být pronajaty od jiných subjektů. I když tím, kdo síť vybudoval a uvedl do provozu, mohl být jiný subjekt. „Nevyužitá“ část kapacity může být poskytována jinému subjektu, např. i na komerční bázi. Faktickým provozovatelem (správcem) může být i jiný subjekt, například na základě tzv. outsourcingu.

VDS (veřejná (datová) síť) - vlastníkem i provozovatelem sítě je určitý (stejný) subjekt, který sám není uživatelem své sítě. Služby sítě jsou poskytovány na komerčním principu jinim subjektum. Služby mohou být nabízeny zájemcům bez omezení (skutečně „veřejně“). Nabízené služby mají nejčastěji charakter přenosu dat.

VAN (Value Added Networks) – VDS s pridanou hodnotou. "Další hodnotu" lze přidávat i na vyšších úrovních - i nad službami pro "holý" přenos dat. Lze přidávat např.: služby přenosu zpráv (MHS, mail), informační služby, faxové služby, video on demand, uchovávání souborů.

VPN (virtuální privátní síť) - jde o podsíť jiné sítě (například: veřejné datové sítě). Logicky (z pohledu uživatele) jde o samostatnou síť. Uživatel si může myslet, že síť je jen jeho a je mu plně k dispozici. Mají samostatný adresový prostor, přístup k uzlům mimo VPN jen přes bránu, event. i vyhrazená přenosová kapacita.

VLAN (Virtual LAN) – Dosud bylo rozdělení počítačů do jednotlivých sítí je dáno jejich fyzickým umístěním. V sitivh VLAN fyzické umístění nehraje roli, je to záležitost logická. O zařazení do určité sítě rozhoduje správce, pomocí konfiguračních nástrojů.

Přístupové metody

Resi problem pristupu ke sdilenemu prenosovemu mediu. Pokud zacne vysílat více stanic soucasne (alespon 2), dojde k tzv. **kolizi.** Netýká se dvoubodové topologie (dva pocitace spojene jednym mediem). Týká se hlavne sítí LAN které používají "vícebodové" topologie. Resi se na urovni linkove vrstvy (podvrstva MAC (Media Access Control) v RM ISOOSI). Varianty reseni, nekolik ruznych pristupu:

Deterministicke (rizene metody) - mají jednoznacne definovaná pravidla, výsledek není ovlivnen náhodou a je plne predikovatelný.

Nedeterministicke (nerizene metody) - Jejich pravidla obsahují „náhodný“ prvek (pockej náhodne zvolenou dobu). Jejich výsledek tedy není predikovatelný. Nemusí vždy vést k výsledku.

Centralizovane - pocítají s existencí centrálního „arbitra“, který rozhoduje. Vetšinou jde o rízené (deterministické) metody.

Distribuovane - neexistuje v nich centrální rídící prvek. Metodu realizují jednotlivé uzly ve vzájemné soucinnosti.

Vylucujici kolize (CA, Collision Avoidance) - vetšinou rízené metody, snaží se aby ke kolizím nedocházelo.

Metody detekující kolize (CD, Collision Detection) - pripouští výskyt kolizí, ale jsou schopné je rozpoznat a reagovat na ne.

Metody eliminující kolize (CE, Collision Elimination) - kolize úplně eliminují, tyto nenastávají. Obvykle jde o plně řízené metody.

Metody bez detekce kolizí - nedokáží zabránit kolizím, ani je nedokáží detekovat (Aloha).

Rizene centralizovane

Arbitr vyzyva (**polling** - centrální arbitr se pravidelne (cyklicky) dotazuje všech potenciálních zájemcu o vysílání) nebo je dotazovan (**requests** - musí existovat možnost vyslat žádost smerem k arbitru (bud po stejnem mediu -> musí existovat casove sloty na zadosti, nebo po samostatne prenosove ceste)).

Vyhody: arbitr muze menit strategie.

Nevyhody: režie na komunikaci mezi arbitrem a uzly může být významná, single point of failure.

Priklad: 100VG AnyLAN (pouze stromova topologie s hlavnim korenem).

Rizene distribuovane

Nemají centrálního arbitra. Mají plně deterministická “pravidla hry“, vsechny uzly rovnocenne.

Varianty: **Metody logickeho kruhu** (token ring) - siti putuje opravneni vysilat (token), ktery si uzly pravidelne predavaji. **Rezervacni metody** (siti putuje specialni

rezervacni ramec, muže mít podobu bitové mapy zadosti jsou zaznamenavany do tohoto ramce).

Vyhody: distribuovanost - nestoji a nepada s jednim uzlem.

Nevyhody: jednotlive uzly toho musi umet vice.

Priklad: FDDI, TokenRing.

Nerizene distribuovane

Metoda vysilej co chces (radiem), kdyz nedojde potvrzeni (vcas) vysilej znovu. Vzniklo na Hawaiskych ostrovech (mediu = eter - radio), kde nebyla zadna pouzitelna infrastruktura.

Vyhody: radio vzduchem - zadne naklady na vystavbu a udrzbu prenosovych cest.

Nevyhody: velka rezie na kolize a znovu posilani.

Priklad: Aloha

CSMA/CD metody

"Poslechne" si (**C**arrier **S**ense), zda nekdo vysila a pokud ne, tak zacne - malo kolizi ale muzou byt - **M**ultiple **A**ccess. Pokud nekdo vysila, tak s pravdepodobnosti p (p-persistentnost) pocka az skonci a pak zkusi (v zavislosti na metode, Ethernet = 1-persistentni) vysilat znovu. Pokud dojde ke kolizi (**C**olision **D**etection), tak se chvili vysila JAM signal pro utvrzeni kolize v kolizni domene. Ruzne metody pak muzou zkusit hned vysilat znovu, mlcet, ... v Ethernetu se vsechny kolizni uzly odmlci na nahodne zvolenou dobu a pak zacnou vysilat znovu (pokud dojde opet ke kolizi, tak se zdvojnasobi interval, ze ktereho si voli nahodnou dobu k odmlceni).

Vyhody: mohou byt velmi efektivni - mala rezie u nizke zateze media.

Nevyhody: nezarucuji vysledek (mohou byt kolize do te doby, nez to uzel vzda), pri vyssi zatezi prenosoveho media se velmi zvysuje rezie.

Priklad: Ethernet

Kolizni domena - kam az se siri kolize - repeatery siri, bridge, routery ji ohranicuji.

Srovnani

Nerizene metody funguji lepe v mensich sitich s mensi vytizenosti media (LAN). Rizene funguji lepe ve vetsich sitich s vetsi vytizenosti (paterni site). V Ethernetu se celkove se ocekava nizke vyuziti prenosoveho media a tim i malo kolizi.

Pristupove metody bezdrátových siti

Bezdrátové sítě jsou specifické větším rušením/porychy/chybovost v přenosovém kanále ("éteru"). Příposlech nosné (CS) nemusí být spolehlivý na rozdíl od "drátových" sítí. Trpi problémem skryté a předsunuté stanice. Nelze detekovat kolize

během vysílání. Rádiová (RF) rozhraní jsou typicky pouze poloduplexní, a neumožňují proto současně přijímat i vysílat a metody CD tedy nepřipadají v úvahu.

Metody sítí Wi-Fi (802.11)

DCF (Distributed Coordination Function) - je povinná varianta pristupove metody. Nemá žádný centrální prvek/autoritu. Varianta CSMA/CA povinna. Varianta CSMA/CA s výměnou RTS/CTS volitelná a implementovaná v "lepších" produktech.

PCF (Point Coordination Function) - je volitelná varianta pristupove metody. AP řídí veškerou komunikaci, ke kolizím vůbec nedochází. V praxi není (moc) implementováno.

Používá se potvrzování kvůli poloduplexnímu způsobu fungování rádiového rozhraní. Protoze vysílající uzel nepozná, že došlo ke kolizi a nedozví se, že by měl přenos opakovat. Rámec se odvysílá vždy celý a příjemce musí přijetí rámce explicitně potvrdit.

Bluetooth - nemá klasickou přístupovou metodu. Místo ní používá velmi rychlé přeskakování mezi frekvencemi (1600x za sekundu). Spoléhá se na to, že případné kolize budou velmi krátké (625 mikro sekund) a bude jich velmi málo. Návrh technologie Bluetooth byl přizpůsoben spíše potřebám přenosu hlasu kde občasné velmi krátké výpadky (kvůli kolizím) nejsou znatelné. Pro přenos dat je to větší problém, ale dá se řešit.

Síťové technologie - ATM

ATM (Asynchronous Transfer Mode) - prenosova technologie pouzivana v nekterych paternich sitich. Je orientovana na QoS, je draha, jen spojovana, nema broadcast, ale zase nema omezenou prenosovou rychlost. Puvodem ze sveta spoju, ale respektuje cast. i svet pocitacu.

ATM pracuje s bunkami dat. Maji vzdy 53B, z toho 5 je hlavicka , 48 naklad. Díky pevné velikosti je lze zpracovávat i v HW. ATM nabizi vyssim vrstvam nekolik trid sluzeb:

CBR (Constant Bit Rate) - garantuje celou kapacitu, emuluje drat – rezervuje pozadovane maximum. Vhodne pro vsechno co generuje konstantni datovy tok: tel. ustredny a nekomprimovana multimedia.

VBR (Variable Bit Rate) - garantuje to, co prenos prave potrebuje - rezervuje max, nevyuzite pasmo vraci (narozdil od CBR), vhodne pro komprimovana multimedia. Třída VBR má dvě varianty: **rt-VBR** (real-time VBR - mohou měnit rychlost bitovéhoproudu (bit rate)) a **nrt-VBR** (non-real-time VBR - pro přenosy vykazující dávkový (bursty) charakter - terminálový přístup do rezervačních systémů)

ABR (Available Bit Rate) - rezervuje min. Pokud je pak požadována kapacita vyšší než MIN, je poskytnuta, pokud jsou dostupné potřebné zdroje. Predpokladane pouziti: site LAN.

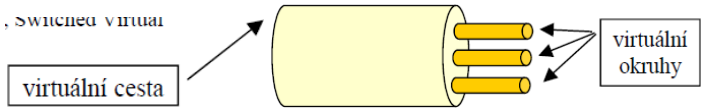
UBR (Unspecified Bit Rate) - negarantuje nic (co zbyde). V zásadě jde o princip „best effort“ z klasických paketových přenosů. Data jsou přenášena na principu FIFO - k odeslání čekají v bufferu až pro ně bude volná buňka.

Vlastnosti ATM

ATM pracuje na **spojovaném principu**. Hlavičky buněk jsou hodně malé ale je v nich prostor na identifikaci spoje (okruhu). Nespojovaný přenos v ATM prakticky nejde protože v hlavičkách malých buněk není prostor pro plnou adresu příjemce. ATM nabízí **virtuální okruhy** které jsou obecně jednosměrné ale lze je vytvářet v párech, pro plně duplexní spojení. Virtuální okruhy mohou být **pevné** (PVC, Permanent Virtual Circuit) nebo **komutované** (SVC, Switched Virtual Circuit). Přenosové služby (virtuální okruhy) nepoužívají potvrzování fungují jako **nespolehlivé** a při zahlcení jsou oprávněny zahazovat buňky ale nejsou oprávněny měnit jejich pořadí.

ATM se snaží maximálně zjednodušit „směrování“ a manipulaci s buňkami v mezilehlých uzlech a důsledkem je **dvouúrovňová hierarchie virtuálních spojů** a jejich adresování:

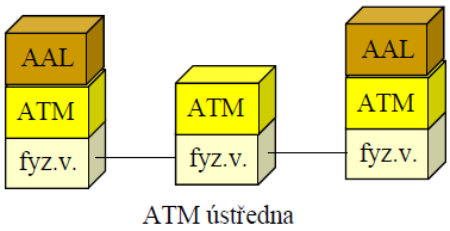
- **virtuální okruhy** (Virtual Circuits, VC)
- **virtuální cesty** (Virtual Paths, VP)

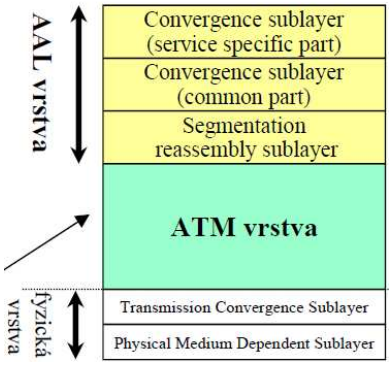


Mezilehlé uzly (ATM ústředny) se při přenosech mezi sebou navzájem budou rozhodovat jen podle cesty. Pouze při doručování koncovým uzlům budou brány v úvahu identifikátory konkrétních okruhů.

Vrstvový model ATM

Základem je **ATM vrstva** má za úkol přenášet jednotlivé buňky. Pod ATM vrstvou je **fyzická vrstva** která zajišťuje vlastní přenos dat, není součástí definice ATM. ATM není vázána na žádnou konkrétní přenosovou rychlost. Pod ATM lze „podstrčit“ různé přenosové technologie. Nad ATM vrstvou je **vrstva AAL (ATM Adaptation Layer)** která má hlavně za úkol přizpůsobování potřebám vyšších vrstev a rozsekavat data na kusy po 44-48B pro prenos pres ATM. AAL je přítomna až v koncových uzlech, nikoli v mezilehlých (nikoli v ATM ústřednách (ATM switches)).





ATM vrstva zhruba odpovídá linkové vrstvě ISO/OSI s podstatným rozdílem: zajišťuje endto-end přenosy (tj. chová se spíše jako síťová vrstva), zatímco linková vrstva přenáší jen k sousedům.

Vrstva AAL připomíná transportní vrstvu. Má vyšším vrstvám zakrýt charakter ATM a dát jim právě to, co chtějí. Nad AAL bývá další (skutečná) transportní vrstva. Poskytuje několik protokolů:

- **AAL1** - je protokol pro přenos dat v reálném čase (malé zpoždění a rozptyl) a konstantní rychlostí (bit rate). V zásadě odpovídá CBR.
- **AAL2** - je protokol pro přenos dat v reálném čase (malé zpoždění a rozptyl) ale s proměnnou rychlostí (bit rate). V zásadě odpovídá VBR.
- **AAL3, AAL4** - jsou protokolmy pro přenosy, které nejsou citlivé na časové závislosti a spíše jim vadí ztráty a chyby. Liší se ve spojovaném (AAL3) a nespojovaném (AAL4) režimu.
- **AAL5** - je reakcí „světa počítačů“ na ostatní AAL. Nabízí spolehlivý i nespolehlivý přenos. Od vyšších vrstev přijímá AAL5 „velké“ bloky až 64 KB. Rámec pak „rozseká“ na kusy á 48 bytů, které vkládá do ATM buněk.

Vyhody: rychla, snazi se byt univerzální.

Nevyhody: tezkopadna, draha.

Priklad: paterni sit ADSL Ceskeho Telecomu (O2).

FDDI

Fiber Distributed Data Interface - **data prenasena optikou**. Pracuje s prenosovou rychlosti **100Mbps**, vhodna pro **paterni site**. Vyvinuta v polovine 80. let. Je znacne robustni, ale dosud velmi draha. Dnes dokaze využívat i kroucenou dvojlinku.

Predpoklada **kruhovou topologii** (fyzicky aj logicky). Pouziva deterministickou pristupovou technologii (**token passing**, ale jinou nez Token Ring). Pocita s velkym dosahem (obvod kruhu az 200 km). Ma zabudovane **automaticke rekonfiguracni mechanizmy** (dokaze kompenzovat jedno přerušeni kruhu).

FDDI kruh

Optická vlákna jsou díky své podstatě vždy jednosměrným přenosovým médiem. Nejlépe jim proto vyhovuje kruhovitá fyzická topologie, v rámci které data "cirkulují" vždy jen jedním směrem dokola. Vypnutí či výpadek kteréhokoli uzlu by však stačily zlikvidovat průchodnost celého kruhu, a tak FDDI počítá s existencí dvou kruhů - jednoho hlavního (či: **primárního**), který postačuje pro běžné fungování za normálních okolností, a kruhu **záložního**, který "nastupuje" až v okamžiku přerušení hlavního kruhu. V případě odpojení či výpadku jednoho uzlu jeho sousedé propojí mezi sebou hlavní a záložní kruh, a tím dosáhnou opětného uzavření souvislé přenosové cesty, a dokáží tedy fungovat dále.

Pripojeni uzlu

Způsob, jakým jsou jednotlivé uzly propojeny v rámci FDDI do kruhu, lze asi nejlépe připodobnit ke vzájemnému propojení zařízení typu opakovačů - každý uzel funguje jako opakovač, a z jedné strany do něj signál vchází a z druhé strany naopak vychází.

DAS (Dual Attached Station) – uzel je pripojen k obema kruhum a účastni se pripadne rekonfigurace. Pouziva se u dulezitych uzlu (servru).

SAS (Single Attached Station) – uzel je pripojen pouze k hlavnimu okruhu a tim padem není schopen účastnit se rekonfigurace. Pouziva sa při pripojeni obyčejných pracovnich stanic.

FDDI koncentrátor – jsou zařízení které jsou samy připojeny k oběma kruhům, a v případě výpadku uzlu napojeného stylem "Single Attachment" jej dokáží uvnitř sebe efektivně odpojit tak, aby byla zachována kontinuita kruhu.

Optical bypass – je zařízení poskytující další možností, jak zachovat spojitost FDDI kruhu i při výpadcích jednotlivých uzlů. Jejich podstatu asi nejlépe vystihuje český překlad "optická výhybka" - v případě výpadku uzlu tato výhybka sama přehodí a propojí části kruhu, které by jinak kvůli výpadkům zůstaly rozpojeny.

Technologie FDDI nabízí dva základní druhy provozu: **synchronní** a **asynchronní**. Základním režimem je přitom režim asynchronní, který umožňuje i použití různých priorit. Standard FDDI připouští také synchronní režim, určený zejména pro přenos hlasu v kódování PCM (Pulse Coded Modulation). Ne všechna FDDI zařízení však tento synchronní režim podporují.

Vrstvy FDDI

PHY (Physical) – implementuje funkce nezavisle na přenosovém mediu (kodovani...)

PMD (Physical media dependent) – zabyva se použitím přenosového media (Fiber-PMD se zabyva použitím prave optického vlakna).

MAC (Media Access Control) – implementuje pristupovou metodu.

SMT (Station Management) – stara se o spravu, rekonfiguraci kruhu…

FastEthernet

Ethernet se zrodil ve středisku PARC (Palo Alto Research Center firmy Xerox). Autoři: Robert Metcalfe, David Boggs. Poprvé zveřejněno v r. 1976 (odborny clanok). První verze Ethernetu (1980) zrychlena na 10 Mbps a vyvinuta ve spolupráci firem DEC, Intel a Xerox (DIX = **DIX Ethernet**). Návrh specifikace byl předložen společnosti IEEE ke standardizaci. Pracovní skupina IEEE 802 návrh přijala, a vydala jako svůj standard, ale poněkud jej pozměnila (např. údaj o délce a obsahu v hlavičce rámce se liší).

Je rozdíl mezi DIX Ethernetem a „IEEE 802.3 Ethernetem“ (DIX Ethernet se dál téměř nevyvíjel). Standardy pocházející od IEEE 802.3 nepoužívají jméno Ethernet protože Xerox si jméno "Ethernet" zaregistroval jako svůj trademark, ale později se jej vzdal, ve prospěch "public domain použití".

Ethernetove Adresy

Uvnitř Ethernetu 48-bitové **MAC** adresy na úrovni **Medium access layer**. Adresy Ethernetového rozhraní by meli být celosvetove unikátni a jsou většinou pevně v hardwaru. Každý výrobce dostane blok (se svým identifikátorem **OUI**), který může přidělovat.

EUI-48 (extended unique identifier): Původně MAC-48. Je to dosavadní 48-bitová Ethernetová adresa. OUI má 24 bitů a 24 bitů je pro sériové číslo. Správce je IEEE, dnes už se nemá používat.

EUI-64: Rozšíření na 64 bitů. Sériové číslo má 40 bitů.

Ethernetove ramce

Rozlišujeme rámce na nižžší úrovni podvrstvy **MAC** a na vyšší úrovni podvrstvy **LLC**. MAC rámcu je v Ethernetu několik druhů, a liší se ve své struktuře (Druhy DIX, IEEE 802.3, Raw a SNAP). Musí obsahovat adresu příjemce, odesilatele a typ obsahu.

Rámec IEEE 802.3 - preambule 7 bytů, 1 byte SFD (start of frame delimiter), 6 bytů adresa příjemce a 6 bytů adresa odesílatele, 2 byty délka rámce (nejvýše 1500B). Na konci 4 byty zabezpečení.

Ramec DIX Ethernetu - preambule má 8 bytů, 2 byty po adresách příjemce a odesilatele udávají typ nákladu. V "nákladové části" může být až 1500 bytů.

Rozlišení DIX a IEEE. Dix má na 13-14 bytu typ rámce > 1500, IEEE tam má delku rámce <= 1500. Různé rámce mohou fungovat na stejném drátě. Dnes se používají IEEE 802.3 IEEE 802.2.

Filozofie Ethernetu

Počítá s všesměrovým charakterem vysílání. Přístupová metoda je CSMA/CD a ta počítá se sběrnicovou topologií sítě po logické stránce. Ve smyslu: vysílání je všesměrové, co jeden uzel vysílá, to „slyší“ všichni ostatní. Tento předpoklad vycházel z původního charakteru kabeláže (koaxiální kabel). Logicky sběrnicová topologie dává Ethernetu sdílený charakter všechny uzly (v rámci stejné kolizní domény) se dělí o jednu společnou přenosovou kapacitu (danou přenosovou rychlostí 10 Mbps). Sdílený charakter (i kolizní doména) „končí“ na nejbližším mostu, přepínači nebo směrovači. Nezaručuje žádnému uzlu, že bude moci odvysílat to, co odvysílat chce. Ethernet poskytuje právo vysílat pouze s určitou pravděpodobností která je velmi vysoká při malé zátěži sítě, ale klesá s rostoucí zátěží.

Pristupova metoda a reseni kolizi

Ethernet pouziva 1-persistentní CSMA/CD pristupovou metodu. T.j. neodmlčí se, čeká na konec aktuálního vysílání. Autori věděli, že 1-persistence nejhůře vytěžuje sdílené přenosové médium, presto si ji vybrali. Duvodem je předpoklad slabeho vytezeni přenosového média a dbali také na latenci (pravděpodobnost, že by uzel čekal na konec vysílání společně s jiným uzlem, a pak se dostali do kolize, je relativně nízká). Uzel se tedy dostane ke slovu rychle a nevzdava to zbytecne. Pokud však ke kolizi predce jen dojde uzel se odmlčí na náhodně zvolenou dobu, a teprve pak se pokouší o vysílání znovu. Pokud by se všechny uzly, zúčastněné v kolizi, zachovaly stejně, pak by zákonitě došlo k další (následné) kolizi. Používá se "zesílení náhody“: zvětšuje se interval, ze kterého si uzel náhodně volí délku svého odmlčení. Při každé následné kolizi se tento interval zdvojnásobí. Při úspěšném odvysílání se zase vrátí na původní hodnotu.

Další vyvoj ethernetu

Sdílený charakter Ethernetu se dnes mění. Nikoli přechodem na „ne-sběrnicovou kabeláž“ ale používáním Ethernetových switchů místo opakovačů. Ethernet se začal používat i s kabeláží, která již není (fyzicky) sběrnicová (kroucená dvoulinka, optická vlákna). Ethernet se tuto „fyzicky nesběrnicovou“ kabeláž snažil stále používat jako „logicky sběrnicovou“ (stále používá metodu CSMA/CD). Nejnovější varianty Ethernetu již opouští metodu CSMA/CD.

Verze enthernetu

10Base5 (ThickWire) - Nejstarší verze Ethernetu počítala s tzv. tlustým (žlutým) koaxiálním kabelem o průměru cca 1 cm. Z něj se dělaly odbočky k jednotlivých uzlům, pomocí tzv. drop kabelů. koaxiální kabel se buď rozpojil a znovu spojil přes tzv. transceiver, nebo byl „nabodnut“ zvláštním nožovým konektorem (tzv. vampire tap). Zakončení kabelu (terminator) bylo nutné proto, aby nedocházelo k odrazům na konci vedení.

10Broad36 - přenos probíhal v přeneseném pásmu jako modulovaný. Dosah jednotlivých kabelových segmentů mohl být větší - až 3,6 km. Dnes se již nepoužívá.

10BaseT – byla snaha využít již existující rozvody „telefonního typu“. Snažila se o to hlavně firma AT&T. Zdokonalování šlo taky cestou zrychlení za cenu zmenšení dosahu. Dosah kabelu: 100 m. Na kroucené dvoulince nelze dělat odbočky a rozvětvení (rozbočení) je proto nutné dělat elektronickou cestou. Kvůli tomu se používají rozbočovače (hub-y). Rozbočení (rozvětvení) může logicky fungovat na úrovni **fyzické vrstvy** (pak se hub chová jako opakovač), **linkové vrstvy** (pak se hub chová jako most, ev. switch) a na urovni **síťové vrstvy**. Pro připojení každého uzlu jsou nutné 2 páry zkroucených vodičů: jeden pro vysílání dat, druhý pro příjem. Kolize je vyjádřena tím, že se na obou párech přenáší data současně.

Standard 10BaseT předpokládá topologii, která již není (fyzicky) sběrnicovitá. Ale je hvězdicovitá, resp. stromovitá. Jednotlivé uzly se již nemusí o své přípojky dělit s ostatními (nemusí je sdílet). Fyzicky hvězdicovitou topologii ale standard 10BaseT používá jako logicky sběrnicovou (tehdy, když se hub chová jako opakovač). Když se hub chova jako switch (most), pak segmenty jsou „jednouzlové“a závada na jednom segmentu ovlivní připojení jen jednoho uzlu, nikoli „všech“ uzlů (na rozdiel od koaxialneho kabelu).

100BaseT (FastEthernet) - je výsledkem snahy zrychlit 10BaseT Ethernet 10x. Prosadily se 2 odlišné přístupy: ponechat vše tak jak je, a pouze vše 10x zrychlit nebo ponechat jen takové vlastnosti, které se ukázaly jako výhodné, ostatní změnit (a 10x zrychlit). V roce 1993 vzniká sdružení Fast Ethernet Alliance, s cílem prosadit jednotný standard Fast Ethernetu. To standardizovalo návrh „Ethernetu beze změn“ – cize se ponechala i nedeterministicka (při vniku kolize) pristupova metoda.
Z navrhu „Ethernet se změnami“ se stal standard 100VG Any-LAN.

10x násobné zrychlení se dosáhlo 10x násobným zkrácením bitového intervalu, zkrácením maximálního dosahu kabelových segmentů, efektivnějším kódováním (10 Mbps: kódování Manchester – 2 změny na 1 bit, 100 Mbps: kódování 4B/5B a NRZI/MLT-3 – 5 změn na 4 bity). Dále se zavedl mechanismus pro detekci rychlosti (umožňuje to vyrábět síťové karty pro 10/100 Mbps, které samy rozpoznají rychlost a přizpůsobí se) a nejkratší možný odstup mezi rámci (IPG, Inter Packet GAP) se zmenšil desetkrát.

Důsledkem zachování metody CSMA/CD je, ze minimální velikost linkového rámce zůstává stejná (64B) ale doba pro detekci kolize nezůstává stejná – stále sice 512 x 1 bitový interval, ale doba přenosu 1 bitu se 10x zkrátila. Maximální dosah se zmenšuje taky (nikoli 10x) - žádný segment z kroucené dvoulinky nesmí být delší než 100 metrů. Drop kabely (MII kabely, mezi transceiverem a kartou) nesmí být delší než 0,5 m.

„Fyzická vrstva“ 100 Mbps Ethernetu se rozdělila na dvě podvrstvy:

- **Medium Independent Interface (MII)** - díky rozdělení fyzické vrstvy je opět možné "osamostatnění" transceiveru. Castěji je ale transceiver integrován na kartě. Protože podvrstva PHY je z velké části analogová. Vrstva MII zajišťuje přizpůsobení mezi PHY a řídícími obvody Ethernetu.

- **Physical Layer Device (PHY) –** umoznuje implementovat analogovy prenos pro vice druhu kabelazi. Je implementována v transceiveru.

Zavedla se možnost používání různých druhů kabeláže: 100 Base TX říká jak provozovat 100Mbps Ethernet nad 2 páry dvoulinky kategorie 5, 100 Base FX dtto, pro optická vlákna, 100 Base T4 říká jak provozovat 100 Mbps Ethernet nad 4 páry dvoulinky kategorie 3 („telefonní“).

Přechod mezi 100BaseT4 a TX/FX je složitější – vyžaduje "dekódování" – převod různě velkých sekvencí bitů a zabere to mnohem více času. Zavedli se 2 druhy opakovačů 100BaseT:

- **Class I (Translational Repeater) -** „dekóduje“ jednotlivé bity, překládá mezi různými druhy kódování (různými médii). Generuje zpoždění v délce 140 bitů - v kolizní doméně smí být jen 1x.
- **Class II (Transparent Repeater) -** „nedekóduje“ jednotlivé bity pouze „vyhlazuje“ signál (nesnaží se jej interpretovat ani na úrovni bitů). Dokáže propojit jen segmenty se stejným způsobem kódování. V kolizní doméně smí být až 2 tyto opakovače.

Rízení toku v Ethernetu

Puvodne nemal žádné mechanismy pro řízení toku (předpokládala se mala zatezenost). Dnes přepínače (switch-e) jsou hodně zatížené a řízení toku na síťové či transportní vrstvě nestačí. Vzniká potřeba dodatečného zapracování mechanismů pro řízení toku primo do ethernetu. Princip fungovani: příliš zatížený přepínač pošle odesilateli dat rámec PAUSE (s parametem: **n** – říká, na kolik časových jednotek se má zdroj odmlčet). Odesilatel pokračuje v odesílání až po vypršení n časových jednotek, nebo po příjmu rámce PAUSE 0. Rámec PAUSE může být poslán jednomu konkrétnímu odesilateli (unicast), všem odesilatelům (broadcast), na speciální multicastovou adresu 01-80-C2-00-00-01 (šíří se jen v daném segmentu, neprochází dál přes mosty/přepínače).

Mechanizmus autodetekce

V 10BaseT existuje mechanismus **NLP (Normal Link Pulse)** - každé zařízení vysílá každých 16 milisekund puls o délce 100 nanosekund (absence tohoto pulsu na druhé straně je interpretována jako přerušení spoje (segmentu)).

100BaseTX používá mechanismus **FLP (Fast Link Pulse)** - dávka 17 až 33 NLP pulsů posílana jen na začátku, když dojde k propojení/zapojení kabelu.

Autodetekce rychlosti: jen 1x NLP = 10 Mbit/s, celá dávka 17-33 NLP = 100 Mbit/s.

Další rozsireni ethernetu

Puvodne poloduplex, ale rozsiren na plne duplexni (vysilani muze probihat obema smery) za predpokladu zadnych opakovacu a jen switchu krome koncovych stanic - uz neni pouzivano sdileneho media - kazda stanice ma primy spoj na switch a tim odpada nutnost CSMA/CD metody. Mozna delka kabelu uz neni omezena technologii na 200m, ale jen utlumem kabelu.

Bezdrátové technologie.

Je k dispozici omezeny rozsah pouzitelnych frekvenci:

Licencovane - (nabizi operatori) provozovatel ma exkluzivni prava na vysilani, zadne ruseni, vydava CTU – gsm (900MHz, 1.8GHz), umts, cdma.

Nelicencovane - musi jen respektovat omezeni o max. vyzařovacím vykonu; jednotlivy provozovatele si mohou rusit signal - wifi (2,4; 5GHz), wimax, bluetooth.

Druhy deleni bezdratovych technologii:

- **Podle druhu mobility**
  - Bez mobility - (FWA,…)
  - S Castecnou mobilitou uzivatele – (Wi-Fi, WiMAX)
  - S plnou mobilitou uzivatele – (GSM, UMTS,…)
  - S mobilitou základových stanic – (systémy LEO)
- **Podle dosahu**
  - Cordless – na vzdálenost v metrech (Bluetooth)
  - Wireless – na vetsi vzdalenosti
  - Satelitni site – GEO (geostacionarni), MEO (mediu earth orbit)
- **Podle zpusobu hospodareni s kmitocty**
  - Bunkove (celularni) – v zadne susedni bunce nesmi byt použita stejna frekvence.
  - Trunkove – frekvencni kanaly jsou zdileny vsemi uzivately v rozsahu cele site.
- **Podle charakteru prenosu**
  - 1:1 – dvoubodovy prenos
  - 1:n – od jedne zakladove stanice k vice terminalum

Bezdratovy ethernet (IEEE 802.11)

Snaha zbavit se závislosti na drátech. původně zamýšleno spíše pro "průmyslové využití" pro propojování různých zařízení (spíše než počítačů). **WLAN** (Wireless LAN) je obecné označení pro bezdrátové sítě LAN. "Bezdrátový Ethernet" (IEEE 802.11) je pouze jednou z technologií, které mohou být použity v rámci WLAN. Wi-fi je "nálepka" (známka) a uděluje se těm produktům, které vyhovují standardům (802.11…) a splňují požadavky na vzájemnou kompatibilitu. Uděluje je organizace Wi-Fi Alliance.

Rok 1997, maximální přenosová rychlost 2 Mbit/s, standard IEEE 802.11 pokrývá:

- **podvrstvu MAC (řízení přístupu)** - varianta DCF, varianta PCF.
- **fyzickou vrstvu (PHY)** - FHSS, DSSS, DFIr

Standard 802.11 byl již při svém schválení zastaralý, ihned začaly práce na jeho vylepšení. Hlavně: zvýšení rychlosti. Vznikají nové pracovní skupiny:

Task Group "A" - snaží se využít nově přidělené pásmo UNII (5,8 GHz). Vyvíjí úplně novou techniku modulace - OFDM, Ortogonal Frequency Division Multiplexing. Dosahuje rychlosti 54 Mbit/s. Vznika standard **802.11a** - 54 Mbit/s v pásmu 5 GHz pouzitelne v USA.

Task Group "B" - snaží se zrychlit dosavadní řešení v pásmu 2,4 GHz. Dosahuje rychlosti 11 Mbit/s. Vznika standard **802.11b** - 11 Mbit/s, v pásmu 2,4 GHz.

Architektura IEEE 802.11

Dva základní režimy fungování:

- **Ad-hoc mode -** kdy dva terminály komunikují přímo mezi sebou. Označováno taky jako peer-to-peer mode.
- **Infrastructure mode -** 1 přístupový bod (**AP**, Access Point), dále n terminálů (**STA**, Station). Terminály nekomunikují přímo mezi sebou ale jen s AP, resp. přes AP. AP je analogií základnové stanice v mobilních sítích. Přenosová kapacita je sdílena všemi právě aktivními terminály.

BSS (Basic Services Set) - nejmenší prvek architektury bezdrátových sítí IEEE 802.11. Dva či více počítačů, které se navzájem rozpoznaly a komunikují spolu. Nemusí obsahovat přístupový bod (AP) pak jde o tzv. **IBSS** (Independent BSS). Častěji BSS obsahuje přístupový bod.

DS (Distribution Systém) – slouzi ke vzájemnému propojení BSS. propojuje na úrovni linkové vrstvy (propouští broadcast). Může být:

- **Drátový (Wired Distribution System) -** častější, nejvíce se používá "drátový" Ethernet (opakovače, přepínače, mosty).
- **Bezdrátový (Wireless Distribution System) -** propojuje dvě "drátové" sítě, bezdrátovým způsobem. Toto řešení bývá v praxi označováno také jako "wireless bridging". Nebo Repeating WDS, aka Wireless Repeater jeden uzel funguje jako běžný přístupový bod (AP) druhé se chová jako AP a snaží se emulovat první AP (stejná MAC adresa, SSID, šifrování atd.) - slouží ke zvětšení dosahu bezdrátové sítě.

DS poskytuje sluzby skrze AP, které jsou součástí DS:

Station Services (SS) - týkají se vztahu/komunikace mezi AP a stanicí. Patri sem: **Authentication** (stanice musí vlastnit sdílený klíč ktery je stejný pro všechny stanice), **Deauthentication**, **Privacy** (symetrické šifrování), **MAC Service Data Unit (MSDU) Delivery**

Distribution System Services (DSS) - týkají se "příslušnosti" stanic do buněk a k distribučním systémům (DS). Patri sem: **Association** (stanice se "asociuje" s konkrétním AP v jedné BSS, teprve pak může stanice přenášet nějaká data z/do AP), **Reassociation** (opakovaná asociace, při přechodu mezi AP v rámci různých BSS (stejné ESS)), **Disassociation** (stanice poté již nemůže přijímat/odesílat data skrze AP), **Distribution** (přenos dat v rámci ESS po DS - odesilatel předá "svému" AP, ten využije DS k distribuci (přenosu) dat k příslušnému AP, které předá koncovému příjemci), **Integration** (přenos dat mimo danou ESS).

ESS (Extended Service Set) - vzniká propojením (pomoci DS) několika BSS. Odpovídá "bezdrátové síti" WLAN (jedna broadcast doména). Umožňuje přechod z jedné buňky (BSS) do druhé (i do jiné ESS). Všechny AP ve stejném ESS mají stejný identifikátor SSID (jméno sítě = 32-znakové jméno sítě) ale mají odlišné BSSID (jméno buňky = má 6 bytů, v rezimu infrastruktury je MAC adresou AP).

Standard IEEE 802.11 požaduje možnost provázání bezdrátových sítí s "drátovými" a přechod realizuje zařízení označované jako **portál**. Měl by fungovat (nejméně) na linkové vrstvě.

Fyzická vrstva IEEE 802.11 je ve skutečnosti rozdělena do dvou podvrstev:

- **PLCP (Physical Layer Convergence Protocol)** - zajišťuje detekci nosné a rozhraní k PMD.
- **PMD (Physical Media Dependent)** - zajišťuje modulaci a kódování signálů.

Bezdrátové sítě dle IEEE 802.11 používají různé druhy rámců:

- **PLCP rámce -** zajišťují funkce související s šířením signálu (zejména indikují rychlost, s jakou jsou přenášena data).
- **MAC rámce** tvoria:
    - **řídící rámce (Control Frames)** - RTS/CTS, pro fungování přístupové metody, ACK, pro potvrzování přijatých datových rámců).
    - **rámce pro správu (Management Frames)** - **Beacon** ("maják" - využívá AP k inzerování své přítomnosti), **Probe, Probe Response** (pro zjišťování přítomnosti a schopnosti uzlů), **Association Request/Response** (žádost/odpověď na asociaci stanice s AP), **Disassociation** (žádost o ukončení asociace stanice s AP), **Authentication** (žádost o autentizaci uzlu vůči AP).
- **Datové rámce** (Data Frames)

Adresy v MAC rámcích IEEE 802.11

Uvádí se až 4 adresy, kvůli tomu, že komunikace může "přestupovat" přes přístupové body (AP) a procházet přes distribuční systém (DS). K rozlišení 4 možností pruchodu slouží dva bity (**To DS**(příjemcem je/není AP/DS), **From DS**(odesilatelem je/není AP/DS)). Je nutne rozlišit následující situace:

- **Přímá komunikace dvou uzlů (ad-hoc rezim) -** To DS = 0, From DS = 0
- **Přenos od AP ke stanici (režim infrastruktury) -** To DS = 0, From DS = 1
- **Přenos od stanice k AP (režim infrastruktury) -** To DS = 1, From DS = 0
- **Přenos přes DS (režim infrastruktury) -** To DS = 1, From DS = 1

Power management

Stanice jsou často napájeny z baterií musí proto šetřit s energií a vysílat jen s takovým výkonem, jaký je skutečně nutný. Je nutne vypínat rádiovou část (transceiver), kdykoli je to jen možné funkce power managementu umožňují zjistit, kdy to možné je a kdy nikoli. Možné stavy zařízení v režimu úspory napájení jsou **awake** (zařízení je "vzhůru" a může přijímat i vysílat) a **sleep** (zařízení "spí" a není schopné přijímat). Když stanice přijme rámec a mají pokračovat další rámce, zůstane "vzhůru" (dozví se to skrze příznak "More Data" v položce Frame Control MAC rámce). Stanice, která "spí", se musí pravidelně probouzet a zjišťovat, zda nejsou pro ni připravena data (odesilatel (AP) musí dočasně bufferovat (uchovávat) data, určená spícím stanicím).

Musí teda existovat mechanismy, které umožňují určit stanicím, kdy se mají "probudit" (**TSF, Timing Synchronization Function**) a určit stanicím, zda jsou pro ně připravena nějaká data k přenosu (v bufferech odesilatele) (**TIM, Traffic Indication Map**).

TSF (Timing Synchronization Function) - počítá s pravidelným rozesíláním speciálních management rámců "beacon frame". V sítích v režimu infrastruktury je rozesílá přístupový bod APOD. V ad-hoc sítích se je snaží vysílat všechny uzly, ale "vítězí" jen jeden. Beacon rámce jsou rozesílány pravidelně interval není pevně stanoven a AP si jej může volit sám (obvykle každých 100 ms). Ne vždy se vyslání beacon rámce podaří přesně včas (kvůli obsazenému médiu). Pak se vyslání rámce může opozdit ale původně naplánované intervaly se nemění (další rámec se snaží odvysílat v původně určeném okamžiku). Beacon rámce by měly přijímat i "spící" stanice - měly by se pravidelně probouzet a přijímat. Beacon ramce obsahuji **beacon interval** (údaj o tom, za jak dlouho je plánováno vyslání dalšího beacon rámce), **timestamp** (slouží k tomu, aby si stanice udržovaly své hodiny v synchronizaci), **TIM** (vektor s informacemi o tom, pro které stanice jsou v AP připravena v bufferu data), **SSID**…

Pristupove metody IEEE 802.11

Rádiové rozhraní uzlů je obvykle pouze poloduplexní. Kvůli tomu vysílající uzel nepozná, že došlo ke kolizi a nedozví se, že by měl přenos opakovat. Rámec se proto odvysílá vždy celý, ale může se poškodit/ztratit i z jiných důvodů, než jen kvůli kolizi (např. kvůli rušení). Příjemce proto musí přijetí rámce explicitně potvrdit zaslanim speciálního potvrzovacího rámce (ACK).

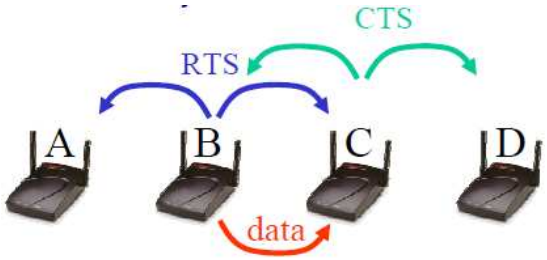
DCF (Distributed Coordination Function) = Nemá žádný centrální prvek/autoritu.

DCF CSMA/CA - varianta je povinná. Zájemce o vysílání sleduje, zda právě probíhá nějaké vysílání, pokud ne, začne hned vysílat sám. Odvysílá celá rámec, aniž by monitoroval eventuelní kolize a čeká na potvrzení (ACK). Pokud právě probíhá nějaké vysílání, odmlčí se hned na náhodou dobu (0-persistence). Pokud během čekání probíhá nějaké vysílání, odpočítávání doby čekání je pozastaveno. Nedokáže garantovat výsledek. Uzel se nemusí dostat "ke slovu. Cas se měří na **sloty** (1 slot je 50μs pro FHSS, resp. 20μs pro DSSS). Není to úplně CA (Collision Avoidance) - ke kolizím může docházet, ale nevyhodnocují se. Fungování je narušováno efektem "skryté stanice" a "předsunuté stanice".

Skryta stanice - médium je "obsazeno", ale uzel se to nedozví. A chce vysílat k B, ale "neslyší" že C právě vysílá k B (signál od C už nedosáhne k A).

Predsunuta stanice - médium je fakticky volné, ale uzel se dozví, že je obsazeno. B vysílá k A, C by chtěl vysílat k D – ale zjistí, že B vysílá a domnívá se, že je médium obsazeno.

DCF CSMA/CA s výměnou RTS/CTS - je volitelná, implementovaná v a vyskytuje se jen v "lepších" produktech a používá se spíše pro "venkovní" komunikaci. Snaha eliminovat problémy skryté a předsunuté stanice. snaha upozornit "ostatní" uzly na to, že po určitou dobu bude probíhat přenos, a že by do něj neměly zasahovat.



Priklad: B chce něco odvysílat k C (snaží se "vyřadit" A a D, aby do toho nevstupovaly).

- B vyšle krátký rámec **RTS (Request To Send)**, určený pro C. Zádá jej o právo vysílat k uzlu C a říká, jak dlouho bude vysílání trvat.
- Tento RTS rámec by měly zachytit ostatní uzly v okolí uzlu B (např. A) a měly by si z něj odvodit, jak dlouho bude B vysílat. Nastaví si "stopky", v podobě vektoru **NAV (Network Allocation Vector).**
- C odpoví krátkým rámcem **CTS (Clear To Send)**. Signalizuje připravenost k příjmu a říká, jak dlouho bude přenos trvat.
- Tento CTS rámec by měly zachytit ostatní uzly v dosahu uzlu C (např. D) a nastaví si svůj NAV.
- Pak probíhá samotný datový přenos. Ostatní uzly čekají na konec vysílání (podle "stopek" v podobě vektoru NAV). Během čekání se ostatní uzly nesnaží samy vysílat.

PCF (Point Coordination Function) - volitelná varianta pristupove metody. AP (přístupový bod) řídí veškerou komunikaci, ke kolizím vůbec nedochází v praxi není (zatím) implementováno. Je ale vázána na režim infrastruktury (vyzaduje existenci AP). AP rozděluje čas na tzv. superrámce (superframe). Každý super-rámec má dvě části: bez soutěže (**contention-free period**) a se soutěží (**contention period**). Během fáze bez soutěže (contention-free period) se AP (koordinátor) explicitně dotazuje jednotlivých uzlů, zda mají co k odeslání (pooling).

Fyzická vrstva IEEE 802.11

Frequency hopping byl použit pouze v původní verzi standardu z roku 1997. Už není použit ve verzích 802.11a,b z roku 1999. PHY vrstva IEEE 802.11b (z roku 1999) používá již pouze techniku DSSS !!! Je zpětně kompatibilní s DSSS z IEEE 802.11 (tj. podporuje rychlosti 2 a 1 Mbit/s).

DSSS (Direct Sequence Spread Spectrum) - používá 8-bitový pseudonáhodný kód. Všechny stanice a AP používají stejnou pseudonáhodnou sekvenci (stejný chipping kód (chip)). Pouziva také jiné kodovani - Complementary Code Keying. Podporuje rychlosti 1 Mbit/s, 2 Mbit/s, 5,5 Mbit/s a 11 Mbit/s. DSSS v IEEE 802.11(b) pracuje v bezlicenčním pásmu 2,4 GHz. Používá frekvenční kanály o šířce 22 MHz. V pásmu 2,4 GHz existují jen 3 vzájemně se nepřekrývající pásma tj. umožňují provozovat 3 sítě WLAN "vedle sebe", na různých kanálech aniž by se vzájemně rušily.

Mobilní komunikace

1. generace (1G) mobilních sítí

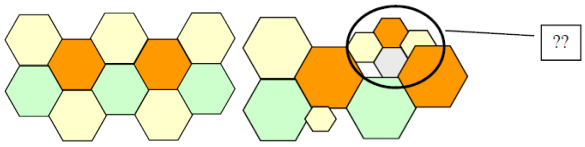
Je ještě analogová. Pro dělení dostupných frekvencí na menší části (kanály) používá techniku frekvenčního multiplexu (FDM). Pro zajištění obousměrného přenosu techniku FDD. Pro modulaci techniku frekvenční modulace. **Pro jednotlivé hovory se používají vždy celé frekvenční kanály**.

2. generace (2G) mobilních sítí

Je již digitální. Pro dělení dostupných frekvencí na menší části (kanály) používá techniku frekvenčního multiplexu (FDMA). Každý frekvenční kanál je dále dělen na principu časového multiplexu (TDMA) nebo celé širší frekvenční kanály se sdílí na principu kódového multiplexu (CDMA). Pro zajištění obousměrného přenosu se používá buď FDD nebo TDD. Pro hovory jsou využívány jen části frekvenčních kanálů (časové sloty). Hlas je přenášen v digitální podobě.

Hospodaření s frekvencemi

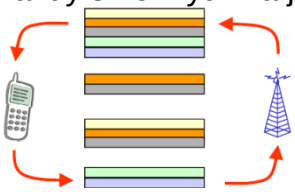
Mobilní sítě zásadně fungují v licenčních pásmech. Frekvence dostávají přidělené na základě individuální licence a tím pádem je počet operátorů omezen dostupností frekvencí. Operátoři dostávají jen určitý počet "frekvenčních kanálů" a proto potřebují pokrýt "neomezeně velkou plochu" ale jen "s omezeným přídělem frekvencí". Řešením je **buňkový (celulární) princip**: plocha k pokrytí je rozdělena na dílčí části (buňky). V sousedních buňkách se nesmí použít stejné frekvence, ale mohou se opakovat v nesousedních buňkách. Struktura buněk mobilní sítě není pevně dána. Vyvíjí se v čase, podle rozvoje sítě a potřeb zákazníků obvykle se zahušťuje (kde je větší provoz, vzniká více (menších) buněk).



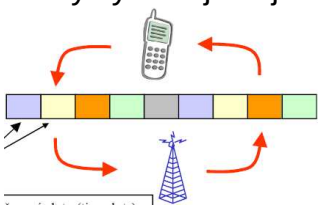
Obousměrná komunikace

Cílem obousměrné komunikace je mít možnost komunikovat v obou směrech současně (přijímat i vysílat současně), avšak není únosné komunikovat obousměrně na stejné frekvenci a ve stejném čase. Proto se využívá:

- Technika **FDD** (Frequency Division Duplexing) - je to analogová technika, kde každý směr využívá jiné frekvence.



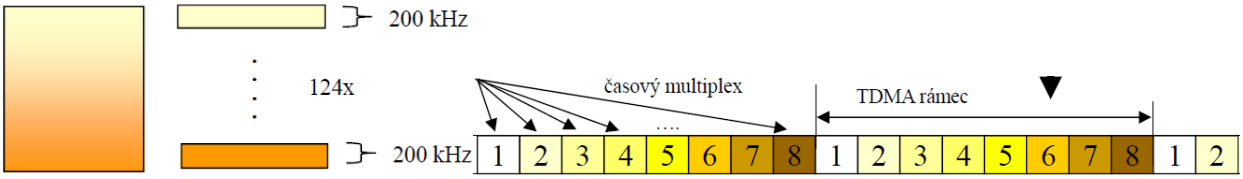
- Technika **TDD** (Time Division Duplexing) - je to digitální technika, kde oba směry využívají stejné frekvence, ale střídají se v čase.



GSM

Byl první komerčně provozovaný systém 2. generace. Vyvinut v 80-tých letech v Evropě pod patronací a za peníze Evropské unie. Dnes je nejrozšířenější standard 2. generace. V Evropě je provozován v pásmu 900 a 1800 MHz od roku 1992 a v USA v pásmu 1900 MHz od roku 1996.

V pásmu 900 MHz GSM používá 124 frekvenčních kanálů pro uplink (890-915 MHz) a 124 frekvenčních kanálů pro downlink (935 až 960 MHz) každý o šířce 200 kHz. To se týka se GSM jako celku! Jednotliví mobilní operátoři dostávají vždy jen **určitou část ze 124 kanálů**. Každý frekvenční kanál je dále dělen (časovým multiplexem) na 8 časových slotů (timeslotů).



Jednotlivé TDMA rámce (každý s 8 timesloty) se střídají s frekvencí 217 Hz. 1 TDMA rámec trvá 4,615 milisekund a 1 timeslot trvá 0,577 ms. Skupina 26 TDMA rámců tvoří 1 "multirámec" a ten trvá 120 ms.

Hlasový hovor zabírá vždy jen 1 timeslot v TDMA rámci ale v každém směru jeden! →Rádiové rozhraní vysílá i přijímá jen 1/8 času a to šetří energii! Vysílání z mobilní stanice (z mobilu) je posunuto vždy o 3 timesloty vzhledem na přijímání aby rádiové rozhraní mělo čas se přepnout.

Kódování hlasu v sítích GSM

Hlas je přenášen v digitální formě. Hlas je snímán 8000x za sekundu a každý vzorek je vyjádřen pomocí 13 bitů, tj. celkově 8000 x 13 = **104 kbit/s**. Následuje komprese RPE/LTP, která sníží datový tok ze 104 kbit/s na **13 kbit/s**. Následuje přidání zabezpečovacích údajů pro detekci a korekci chyb během rádiových přenosů a výsledkem je datový tok **22,8 kbit/s**.

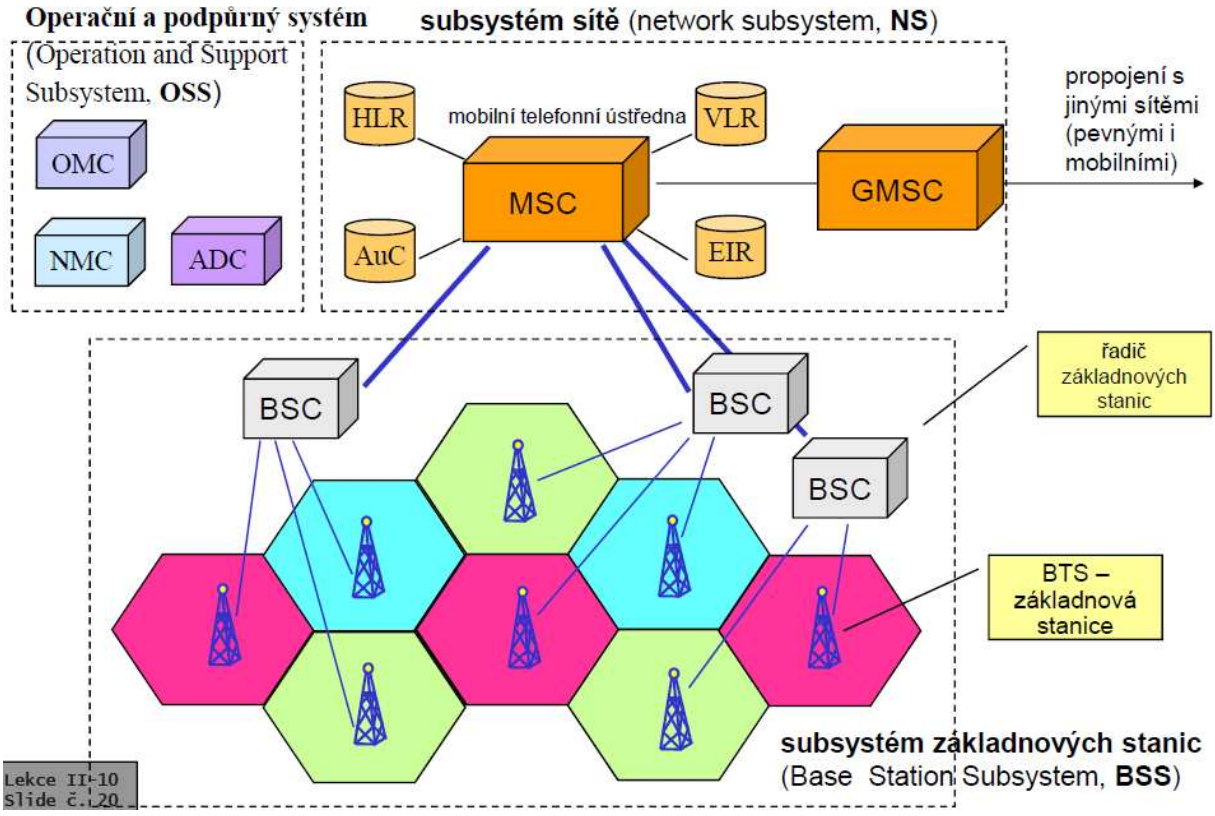
Architektura GSM sítě

Síť GSM je budována na buňkovém principu, tj. plocha, kterou pokrývá, je rozdělena na buňky. V jednotlivých buňkách jsou umístěny tzv. základnové stanice (**BTS**, Base Transceiver Station). Používá se "třísektorová" BTS. Vždy několik BTS je napojeno na jeden společný řadič (**BSC**, Base Station Controller). Páteřní část mobilní sítě řídí

mobilní telefonní ústředna (**MSC**, Mobile Switching Centre). Mobilní ústředna ovládá řadiče BSC a skrze ně jednotlivé BTS.

GSM síť dále obsahuje moduly (registry) v softwarové/hardwarové podobě:

- **HLR** (Home Location Register), který obsahuje informace o uživatelích dané sítě GSM včetně rozsahu předplacených služeb. Uchovává také informaci o tom, kde se mobil nachází (ve které buňce (BSC a BTS)). Každý účastník je registrován vždy jen v jednom HLR!
- **AuC** (Authentication Center) slouží k identifikaci uživatelů. Je součástí HLR a slouží jeho potřebám.
- **EIR** (Equipment Identity Register) obsahuje údaje o odcizených a neoprávněně používaných mobilech (blacklist, whitelist, greylist). Spolupracuje s AUC při ověřování identity a oprávněnosti mobilů ke komunikaci.
- **VLR** (Visitor Location Register) obvykle 1x pro každou ústřednu MSC. Obsahuje údaje o všech uživatelích, kteří jsou právě v dosahu dané ústředny MSC včetně údajů o návštěvnících v rámci roamingu. Jde o jakousi "cache" pro údaje z HLR, tj. dočasné uchování údajů z HLR.
- **OMC** (Operation and Maintenance Center)
- **NMC** (Network Management Center)
- **ADC** (Administrative center) - např. billing, registrace účastníků atd.



Identifikace terminálů (mobilů) v GSM síti

Každý mobilní terminál (MS, mobilní stanice, mobil) je identifikován:

- **IMEI** (International Mobile Equipment Identity number) - 15-ti místní číslo. Identifikuje zařízení jako takové a nemění se.
- **SIM** (Subscriber Identity Module) kartou, která identifikuje uživatele - do které GSM patří, jaké služby má aktivované…SIM karta uchovává údaje: **IMSI** (International Mobile Subscriber Identity) – obsahuje MCC (Mobile Country Code), MNC (Mobile Network Code) a MSIN (Mobile Subscriber Identification Number).
- **MSISDN** (Mobile Subscriber ISDN Number) - skutečné telefonní číslo účastníka uchovává se v HLR a je vztaženo k IMSI.

Přihlašování do GSM sítě

Když mobilní operátor získá nového zákazníka, tak mu přidělí registrační číslo MSIN, stane se součástí IMSI a uloží se na SIM kartu. Přidělí mu telefonní číslo MSISDN např. 420 776 123 456 a uloží se v HLR, spolu s IMSI. Když se mobil (MS) přihlašuje do sítě, předá síti IMEI a IMSI. EIR (Equipment Identity Register) zkontroluje IMEI se svými black/white/grey listem zda je zařízení OK. HLR (Home Location Register) podle IMSI si zjistí MSISDN zapamatuje si polohu MS a předá údaje do VLR. AUC (Authentication Center) vyšle do MS náhodné číslo. MS jej transformuje pomocí klíče na SIM kartě a vrátí výsledek do AUC. AUC tím ověřuje identitu uživatele (SIM karty). VLR (Visitor Location Register) získá údaje od HLR/AUC a přidělí MS dočasné TMSI (Temporary Mobile Subsrcriber Identity). Pod TMSI jej eviduje po dobu pobytu MS v dosahu VLR/MSC.

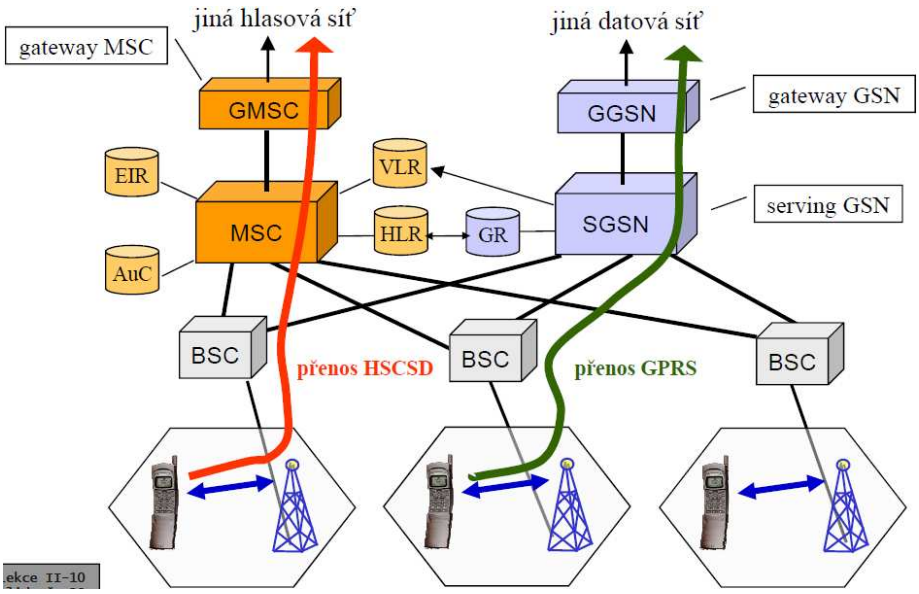
GSM a možnosti přenosu dat

GSM je digitální a teda přenáší i hlas jako data. Existuje několik možností:

CSD (Circuit Switched Data) - místo (zdigitalizovaného) hlasu se budou přenášet obecná data. Fungování GSM sítě se tím pádem nemění. Jde o přenos dat na principu přepojování okruhů. Data se přenáší "hlasovým okruhem", obdobně jako pro hlas. Bez dalších opatření je rychlost CSD to co se vejde do 13 kbit/s (nejbližší nižší normovaná rychlost je 9,6 kbit/s).

HSCSD (High Speed CSD) - varianta CSD, která využívá tzv. channel bundling, tj. využívá více timeslotů současně a rychlost je příslušným násobkem počtu využitých slotů. Stále funguje na principu přepojování okruhů jen je rychlejší. Nevyžaduje změnu HW sítě, stačí jen změna SW.

GPRS (General Packet Radio Service) - HSCSD funguje na principu přepojování okruhů a tedy spotřebovává timesloty po celou dobu existence spojení i když právě nic nepřenáší. To je málo šetrné vůči zdrojům mobilní sítě (vůči timeslotům). GPRS funguje na principu přepojování paketů, tj. když právě nic nepřenáší, nespotřebovává žádné timesloty. Síť dokáže potom obsloužit více uživatelů a díky tomu může být lacinější. GPRS umožňuje trvalé připojení uživatele. Funguje stylem "best effort", proto negarantuje propustnost. Ta je dána momentální zátěží sítě a souběhem požadavků na GPRS přenosy. Nejprve se přidělují timesloty pro hlas, pak pro CSD, a pro GPRS teprve to, co zbude. Mobilní operátor obvykle rezervuje pro GPRS 1 až 2 timesloty v každém TDMA rámci. GPRS vyžaduje zásahy do mobilní sítě: nové prvky sítě a SW upgrade BTS a BSC. GPRS zavádí nové způsoby kódování dat, přenášených mezi MS a BTS.



- **GSN (GPRS Support Node)** - v podstatě jde o směrovače
- **GR (GPRS Register)** - obvykle je součástí HLR. Uchovává data relevantní k GPRS.
- **SGSN (Serving GSN)** - je na stejné úrovni jako (hlasová) ústředna MSC. Je napojen na BSC pomocí datového spoje. Může jich být více. Získává údaje o uživatelích od registru GR. Vede si evidenci toho, kde se MS nachází. Překládá adresy a přenáší data k BSC.
- **GGSN (Gateway GSN)** - zajišťuje vazbu na externí datové sítě. Z jedné strany je napojen na externí datovou síť a z druhé strany je napojen na SGSN pomocí IP spoje. Uživatelská data se mezi SGSN a GGSN tunelují.

Přihlašování v GSM síti k službe GPRS

MS (mobil) se musí nejprve přihlásit do GPRS sítě, tj. předá své IMEI a IMSI uzlu SGSN. SGSN ověří identitu a oprávněnost MS a uživatele. Dále si zkopíruje z HLR údaje o uživateli a přidělí mu dočasné P-TMSI (Packet – temporal Mobile Subscriber Identity). Pod tímto identifikátorem s ním dále pracuje. Nastává tzv. **GPRS attach**. Odhlášení může iniciovat jak MS, tak GPRS síť a nastává tzv. **GPRS detach**. Pro možnost datové komunikace musí MS získat tzv. PDP kontext (Packet Data Protocol context), tj. PDP adresu (obvykle: IP adresu, přidělení je statické či dynamické), dohodnuté parametry QoS… PDP kontext se uchovává v MS, SGSN a GGSN. Přidělením PDP kontextu se MS stává "viditelné" a je dostupné po celou dobu existence (aktivního) PDP kontextu. Směrování se provádí na základě vztahu mezi IMSI a PDP. Jeden uživatel může mít přiděleno více kontextů.

Přístup k Internetu přes GPRS

MS zde navazuje spojení resp. přihlašuje se k "přístupovému bodu" - APN (Access Point Name), který je prostředníkem pro jeho přístup do Internetu. APN obvykle

definuje vlastnosti a parametry připojení např. statické/dynamické přidělení IP adresy, NAT, QoS atd. APN je jakousi bránou do Internetu.

Bezdrátový broadband

Wireless Personal Area Networks

Jsou sítě s velmi malým dosahem. **POS** (Personal Operating Space), znamená do okruhu 10 metrů. Slouží k propojování "drobných" zařízení mezi sebou např. PDA – PDA, k připojování periferií např. hands-free sada k mobilu a mají typicky ad-hoc charakter a dynamickou topologii.

Technologie od ETSI: **Hiper PAN** (High Performance Radio Personal Area Network)
Technologie od IEEE: **Bluetooth** (802.15.1), kompatibilita BT s Wi-Fi (802.15.2).

Bluetooth

Je bezdrátová rádiová technologie určená pro nasazení „na krátkou vzdálenost“ v rámci WPAN, resp. POS. Motivace pro Bluetooth byla snaha vytvořit jednotný standard pro bezdrátovou komunikaci na krátkou vzdálenost - náhrada kabelů. Je to taky šikovnější alternativa k IrDA.

Scénáře využití: **připojení periferií** (klávesnic, myší a head-setů k počítaču nebo PDA), **realizace ad-hoc sítí** (PDA-PDA, PDA-mobil), **propojení sítí** (bridging) - např. mobil se přes GPRS/UMTS atd. připojuje k Internetu, a počítač/PDA přes Bluetooth k mobilu a Internetu.

Požadavky na Bluetooth: snadné nasazení a použití bezlicenčního pásma 2,4 GHz, tj. musí mít velkú odolnost proti rušení. Co nejnižší spotřeba el. energie, co nejmenší rozměry a co nejnižší cena. Podpora přenosu dat i hlasu a možnost peer-to-peer komunikace.

Vlastnosti

Využívá pásmo 2,4 GHz. Rozděluje ho na 79 frekvenčních kanálů o šířce 1 MHz (kanál 0: 2402 MHz, kanál 78: 2480 MHz). Používá techniku Frequency Hopping Spread Spectrum (přeskakuje mezi frekvenčními kanály 1600x za sekundu). Sekvence přeskoků je pseudonáhodná a určuje ji uzel v roli "master". Pracuje na principu TDD (Time Division Duplex), tj. pro rozlišení mezi příjmem a vysíláním se používá dělení v čase (někdy vysílá, jindy přijímá, nikoli současně). Režim úspory napájení jsou **hold mode** (zařízení přijímá jen "hlasovou komunikaci"), **sniff mode** (zařízení spí, ale periodicky se probouzí) a **parked mode** (zařízení je "zaparkováno" (spí, ale master jej může vzbudit)).

Režimy přenosu

- **SCO** (Synchronous Connection Oriented) - tzv. "**Voice Link**".Přenosová rychlost je **64 kbit/s**, plně duplexní dvoubodový spoj, fungující na principu přepojování okruhů. Je určen hlavně pro "hlasové" periferie, například pro komunikaci handsfree sad s mobilem.
- **ACL** (Asynchronous ConnectionLess) - tzv. "**Data Link**". Až **433,9 kbit/s** symetricky, nebo až **723,2 kbit/s / 57,6 kbit/s** asymetricky, na principu přepojování paketů. Topologie point-to-point nebo point-tomultipoint.

nominální rychlost (bez režie):
1 Mbit/s.

Architektura Bluetooth sítí

- **Piconet** - je nejmenší síť. Soustava uzlů, které se sjednotily na stejné posloupnosti přeskakování frekvencí. Piconet tvoří 1x **Master** (určuje posloupnost přeskoků, ostatní se s ním synchronizují) a nejvýše 7 uzlů v roli **(Active) Slave** (spolu s Masterem jsou adresovány 3-bitovou adresou aktivních zařízení).
- **Scatternet** - když se několik piconetů propojí do většího celku. Jednotlivé piconety ve scatternetu mají vlastní posloupnosti přeskoků.

WiMAX (Worldwide Interoperability for Microwave Access)

Je společné označení pro řadu technologií, které vychází ze standardů IEEE 802.16. Technologie jen také, co úspěšně prošli testy sdružení WiMAX. Jde jen o nálepku, podobně jako u WiFi kde testy provádí a nálepku uděluje WiFI Alliance.

WiFi vznikla jako technologie pro koncové uživatele (pro nasazeni v rámci "posledního metru") a pro použití v bezlicenčním pásmu. Typicky indoor, uvnitř budov v rámci bezdrátových lokálních sítí (sítí WLAN).

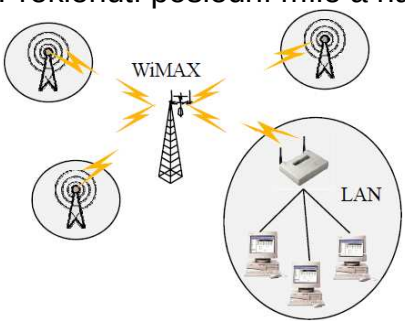
WiMAX vznikl jako technologie pro providery (pro nasazení v rámci "poslední míle") a pro použití spíše v licenčním pásmu. Typicky outdoor, vně budov v rámci bezdrátových metropolitních sítí (sítí WMAN).

Scénáře nasazení

- **Pro "backhaul"** - vytváření páteřních spojů do míst, kde je konektivita dále distribuována. Například napojení Wi-Fi hotspotů na páteřní sítě.

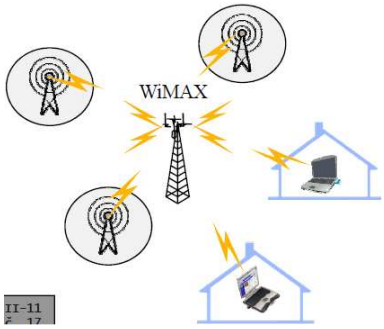


- **Pro připojení koncové sítě LAN** - alternativa/náhrada DSL či kabelu. Překlenutí poslední míle a napojení celé sítě LAN.



- **Pro připojení "nomadického" koncového uživatele** - WiMAX podporuje přímo koncové zařízení, používané uživatelem. Uživatel se může přemisťovat,

ale během komunikace se nepohybuje.



- **Pro připojení plně mobilního uživatele (pouze "mobilní WiMAX")** - WiMAX podporuje přímo koncové zařízení uživatele. Uživatel může komunikovat i v době, kdy se pohybuje.



Verze WiMAX-u

IEEE 802.16 – 2004 (Fixed WiMAX) - nepředpokládá se mobilita uživatele. Ten se může i přemisťovat, ale jen v rámci nomadicity. Tj. během přemisťování nekomunikuje s WiMAX sítí. Frekvenční pásma 2 až 11 GHz. Šířka frekv. kanálu max. 25 MHz. Duplexnost FDD nebo TDD. Max. rychlost až **75 Mbit/s**. Očekávaný dosah až 30 mil (až cca 48 km). Reálný dosah 3 až 5 mil (cca 5 až 8 km). Umožňuje NLOS (Non-Line of Sight, tj. nemusí být přímá viditelnost).

IEEE 802.16e (Mobile WiMAX) - předpokládá se mobilita uživatele až do jízdy autem. Využívají se užší frekvenční kanály a také nižší frekvence. Frekvenční pásma jen licenční, do 6 GHz. Šířka frekv. kanálu 1,25 až 20 MHz. Duplexnost jen TDD. Max. rychlost až 30 Mbit/s (10 MHz kanál). Očekávaný dosah až 3 míle (až cca 5 km). Umožňuje mobilitu (rychlost pohybu až 150 km/h) a NLOS.

Společné vlastnosti WiMAX technologií

WiMAX není "jedna technologie"ale je to několik různých technologií které nejsou vzájemně kompatibilní. Už jen kvůli tomu, že pracují v různých frekvenčních pásmech a používají různě široké frekvenční kanály. Společné vlastnosti WiMAX technologií:

- Používají centralizované řízení - o tom, kdo a kdy bude komunikovat, rozhoduje centrální autorita (základnová stanice). Dotazuje se stanic, zda chtějí něco přenést.
- Předpokládají pouze spojovaný způsob komunikace terminál (**SS, Subscriber Station**) může komunikovat až tehdy, když od základnové stanice (**BS, Base Station**) dostane přidělený kanál.
- Podporují QoS - díky spojovanému charakteru.

RM ISO/OSI

Sítový model je ucelená predstava o tom, jak mají být síte rešeny (obsahuje: pocet vrstev, co má která vrstva na starosti; neobsahuje: konkrétní predstavu jak která vrstva plní své úkoly - tedy konkrétní protokoly). Príkladem je referencní model ISO/OSI (konkrétní protokoly vznikaly samostatne a dodatecne).

Sítová architektura navíc obsahuje konkrétní protokoly - napr. rodina protokolu TCP/IP.

Referencný model ISO/OSI (International Standards Organization / Open Systems Interconnection) bol pokusom vytvorit univerzálnu sietovú architektúru – ale skoncil ako sietový model (bez protokolov). Pochádza zo „sveta spojov“ – organizácie ISO, a bol „oficiálnym riešením“, presadzovaným „orgánmi štátu“; dnes už prakticky odpísaný - prehral v súboji s TCP/IP. ISO/OSI bol reakciou na vznik proprietárnych a uzavretých sietí. Pôvodne mal model popisovat chovanie otvorených systémov vo vnútri aj medzi sebou, ale bolo od toho upustené a nakoniec z modelu ostal len sietový model (popis funkcionality vrstiev) a konkrétne protokoly pre RM ISO/OSI boli vyvíjané samostatne (a dodatocne zaradované do rámca ISO/OSI). Model vznikal maximalistickým spôsobom - obsahoval všetko co by mohlo byt v budúcnosti potrebné. Vdaka rozsiahlosti štandardu sa implementovali len jeho niektoré podmnožiny - ktoré neboli (vždy) kompatibilné. Vznikol GOSIP (Government OSI Profile) urcujúci podmnožinu modelu, ktorú malo mat implementované všetko štátne sietové vybavenie. Naproti tomu všetkému TCP/IP vzniklo naopak - najprv navrhnutím jednoduchého riešenia, potom postupným obohacovaním o nové vlastnosti (tie boli zahrnuté až po preukázaní „životaschopnosti“).

Reseni od "zeleneho stolu". Nebylo plne dodelano, protokoly se dodelavaly az pozdeji postupne. Zacal jako vetsina SW projetku na MFF "velkyma ocima", ze kterych bylo postupne ustupovano kvuli neimplementovatelnosti celeho modelu. Ze zacatku hodne prosazovan statnimi institucemi, ktere ve svych zakazkach vyzadovali kompatibilitu s rm iso/osi.

RM ISO/OSI obsahuje sedm vrstev:



Kazda vrstva ke svemu fungovani pouziva vrstvy primo pod ni na stejnem pocitaci, mezi pocitaci fakticky komunikuje jen fyzicka vrstva.

Fyzicka vrstva - prenasi **bity** (poskytuje sluzby typu: odesli, prijmi bit). Kvuli tomu resi kodovani, modulaci, synchronizaci,… Na jeji urovni se rozlisuje paralelni a seriovy prenos, synchronni a asynchronni,…

Linkova vrstva - prenasi ramce (cele bloky dat) k primym sousedum. Kvuli pretizenosti se rozpadla na dve podvrstvy - **LLC** (multiplexing, rizeni toku) a **MAC** (pristup ke sdílenému mediu). Zajistuje synchronizaci na urovni **ramcu** (detekce zacatku a konce ramcu). Muze fungovat spojovane i nespojovane, spolehlive i nespolehlive, best effort i QoS. **Ridi tok dat** (aby odesilatel nezahltil prijemce) a **pristup ke sdilenemu mediu** (resi konfilkty při vicenasobnem pristupu). Každý uzel vidi jen prime sousedy na urovni linkovej vrstvy.

Sitova vrstva - prenasi **packety** pomoci **smerovani** do cele site. Muze pouzivat ruzne routovaci algoritmy. Je to posledni vrstva, kterou **musi mit smerovace** (routery). **Routing** - hledani cesty k adresatovi, **forwarding** - poslani packetu tou cestou. Každý uzel vidi celou topologii site.

Transportni vrstva – Zajistuje **end-to-end komunikaci** (napr. jako u TCP/IP: TCP prenos a UDP prenos) mezi koncovými uzly. Zajistuje potrebne přizpůsobeni nizsich vrstev potřebám vyssich. **Muze pridavat sluzby** - napr. nizsi vrstvy (patrici nekomu jinemu) nabizi jen nespolehlivou komunikaci a transportni vrstva doda (pripadne zvysi) spolehlivost. Transportní vrstva již **rozlišuje konkrétní entity** v rámci každého uzlu (tj. jednotlivé procesy, démony, úlohy rozlišuje obvykle neprímo, skrze prechodové body (body SAP, porty) ke kterým jsou tyto entity asociovány – napr. cisla portu v TCP/IP).

Relacni vrstva - muze zajistovat sifrovani, podporu transakci. Je to nejvice kritizovana soucast RM ISO/OSI. V TCP/IP neni zastoupena.

Prezentacni vrstva - prezentacní vrstva má na starosti potrebné konverze, tak aby obe strany interpretovaly prenášená data stejne (formát císel, kódování znaku, …) a prevod prenášených dat z/do takového tvaru, v jakém je možné je prenášet (napr. je nutné "zlinearizovat" vícerozmerná pole, vícerozmerné datové struktury).

Aplikacni vrstva - puvodne mela obsahovat cele aplikace (ty by ale museli byt všechny standardizovany), ale nakonec jen jadro aplikaci, ktere ma smysl standardizovat - napr. prenos el. posty.

Kritika

Vyroben od zeleneho stolu, prilis slozity, nerespektuje požadavky a realitu bezne praxe, nektere cinnosti zbytocne opakuje na kazde vrstve, upřednostňuje spolehlive a spojovane prenosové služby (možnost nespolahlivého/nespojovaného spojenia bolo pridané do štandardu až dodatocne).

Aktivní prvky (bridge, routery)

Rozbočovač (hub)

Je obecně aktivně fungující propojovací zařízení, bez apriorního určení úrovně na které pracuje. Může fungovat jako opakovač, jako most i jako směrovač. Jeho funkce záleží na tom, jaké moduly se pořídí a instalují do šasi.

Opakovac (repeater)

Zajišťuje propojení na fyzické vrstvě. Propojuje úseky kabelů (kabelové segmenty). Opakovač je pouze digitální zesilovač, který zesiluje a znovu tvaruje přenášený signál. Kompenzuje zkreslení, útlum a další vlivy reálných obvodových vlastností přenosových cest pritom nezesiluje šum. Opakovač „nevnímá“, že určité skupiny bitů patří k sobě a tvoří přenosový rámec. Nedokáže rozpoznat ani adresu odesilatele a příjemce dat (rámce). Všechna data rozesílá („*opakuje*“) do všech stran (segmentů), ke kterým je připojen. Funguje v reálném čase a proto nemá žádnou vnitřní paměť pro bufferování dat. Může propojovat jen segmenty se stejnou přenosovou rychlostí.

Opakovač v Ethernetu

Opakovač je obecně nezávislý na protokolech linkové vrstvy (když funguje na fyzické vrstvě). Ale je závislý na specifikacích fyzické vrstvy, které typicky úzce souvisí s protokoly linkové vrstvy (existují např. opakovače pro Ethernet). V Ethernetu nesmí být opakovačů příliš mnoho! **Mezi žádnými dvěma uzly nesmí být více jak dva opakovače.** Duvodem je metoda CSMA/CD, která u 10 Mbit/s vyžaduje aby se kolize rozšířila „*z jednoho konce na druhý konec*“ nejdéle do pevně dané doby **t = 51,2 µs**.

V Ethernetu je možné aby více uzlů vysílalo současně (není to žádoucí = dochází k tzv. kolizi). Opakovač v Ethernetu **musí šířit i kolize**, aby i uzly v jiných segmentech poznaly, že k ní došlo. Všechny segmenty, propojené opakovačem (opakovači), tvoří tzv. **kolizní doménu**. Ta končí až na nejbližším mostu, přepínači nebo směrovači.

Most (bridge), prepinac (switch)

Nevýhodou opakovačů je skutečnost, že "propouští" veškerý provoz z jednoho segmentu do druhého, i když by to vůbec nebylo nutné. Výhodnější by jistě byl případ, kdyby propojovací zařízení dokázalo správně rozpoznat, která data mohou zůstat "uvnitř" příslušné části sítě (segmentu), a nezatěžovat jimi provoz v ostatních segmentech. K tomu je ale nutné, aby propojovací zařízení "vnímalo" celé přenášené rámce a znalo jejich formát natolik, aby dokázalo zjistit fyzickou adresu jejich odesilatele a příjemce (a podle toho je pak propustit do sousedního segmentu či nikoli). To ovšem znamená, že takovéto propojovací zařízení, které se pak nazývá **most** (**bridge**), musí **pracovat na úrovni linkové vrstvy** (resp. na úrovni podvrstvy řízení přístupu k médiu (vrstvy MAC) v případě lokálních sítí dle standardů IEEE 802), kde již lze fyzické adresy příjemce a odesilatele rozpoznat. Díky bufferování ("store and forward") může propojovat segmenty s různými přenosovými rychlostmi. Propojovací uzel není pro ostatní uzly viditelný - odesilatel neví o propojovacím uzlu, odesílaný rámec adresuje koncovému příjemci (v dané síti). Za normálních okolností

by mu neměly být přímo adresovány (MAC adresa) žádné rámce. Bridge nemá vlastní adresu na úrovni síťové vrstvy (např. IP adresu).

Most (bridge), prepinac (switch) v Ethernetu

Díky bufferování není nutné kolize propagovat. Pravidlo o max. počtu opakovačů se „zastavuje“ na nejbližším mostu, přepínači či směrovači. Na úrovni linkové vrstvy se musí propouštět a šířit všesměrové vysílání (broadcasting-u) do všech segmentů.

Most, prepinac musí mít dostatečné informace o skutečné topologii sítě v dosahu přímého spojení, k nejbližším směrovačům. Rozsah informací, které potřebuje, je relativně malý. Most i přepínač je schopen (nějak) fungovat i tehdy, když tyto informace nebude mít k dispozici (bude fungovat jako opakovač, a rozešle všechno na všechny strany). Lze tedy připustit, aby si most sám získával potřebné informace ze svého okolí (učil se) a do doby, než se „naučí“, fungoval neefektivně. Propojovací uzly, fungující na linkové vrstvě (mosty, přepínače) mohou být tedy zařízení typu „plug & play” - není nutné je konfigurovat.

Zpětné učení v Ethernetu

Most (přepínač) začíná fungovat bez jakekoliv informace o topologii v tomto stavu se chová jako opakovač (na úrovni linkové vrstvy). Je to neefektivní ale lze to připustit – netrvá to dlouho, čím větší je provoz tím dříve to skončí. Most (přepínač) průběžně sleduje z jakých adres mu přichází jednotlivé rámce.

Když dostane rámec od uzlu A pro uzel B ze směru X, odvodí si že „**A leží ve směru X**“ a rámec rozešle do všech směrů (kromě X). Z případné odpovědi se „dozví“ umístění uzlu B - "**B leží ve směru Y**". Příští rámec od A pro B již pošle cíleně jen do směru Y, ve kterém se B skutečně nachází (obdobně pro rámec od B pro A).

Proces samoučení nebude fungovat, když v síti budou cykly (smyčky). Pak most (přepínač) přijme jeden rámec z více různých směrů. Inteligentní mosty a přepínače se dokáží vzájemně domluvit a cyklus přerušit - aplikují algoritmus STA (Spanning Tree Alg.) a vytvoří kostru grafu.

Most v sítích Token Ring

Používají se mosty fungující na principu „source routing“. Každý jednotlivý rámec si v sobě nese úplný seznam uzlů, přes které má projít. Tento seznam sestavuje odesílající uzel (proto „source“ routing). Má to blíže k síťové vrstvě než k vrstvě linkové. Znalost o topologii sítě vezme odesílající uzel před prvním odeslanim jekehokoli jineho ramce z průzkumneho paketu (spíše rámec), ktery se šíří záplavově (jako lavina), až dorazí ke svému cíli. Po dosažení cíle se průzkumný paket vrací a nese v sobě údaj o cestě, kterou se k cíli dostal. Najde skutečně „nejkratší“ cestu není to ale příliš adaptivní.

Rozdil mezi mostem (bridge) a prepinacem (switch)

Neliší se principem fungování. Oba fungují na úrovni linkové vrstvy. oba se rozhodují podle informací dostupných na linkové vrstvě. Oba zajišťují filtering i forwarding.

Most (bridge) je starší typ zařízení, má méně portů pro připojení dílčích (sdílených) segmentů. Není u něj důraz na výkonnost (na rychlost provádění forwardingu) spise na filtering. Jeho vnitřní fungování je často řešeno programovými prostředky. mosty vznikly v době, kdy Ethernet používal koaxiální kabely a měl skutečně sběrnicovou topologii. Segmenty měly typicky více uzlů. Mosty se snažily udělat maximum pro využití dostupné kapacity.

Přepínač (switch) je novější typ zařízení, má více portů než typický most. Je optimalizován na výkonnost a celkovou propustnost (pro potřeby forwardingu). Jeho "přepojovací stroj" (switching engine) je typicky realizován v hardwaru. V Ethernetu funguje na principu samoučení (i když kvůli optimalizaci může pracovat i se statickou konfigurací). Je dulezity pro zvýšení propustnosti (mikrosegmentace), protože je forwarding rychly. Když každý dílčí (sdílený) segment bude "obydlen" jen jedním uzlem pak nebude existovat žádný lokální provoz a nebude žádný filtering. Veškerý provoz bude třeba cíleně forwardovat do příslušného cílového segmentu. Ethernet totiž přešel na použití kroucené dvoulinky, která není sdíleným médiem. Topologie se ze sběrnicové změnila na stromovitou. Přepínač (switch) je vlastně řešením, které naplno využívá možností stromovité topologie. Zrodil se tzv. "**přepínaný Ethernet**".

Typy prepinacu

Store&forward přepínač - přepínač nejprve načte celý rámec a uloží jej do svého bufferu (store), pak se rozhodne co s ním a event. ho předá dál (forward). Má větší průchozí zpoždění.

Cut-through přepínač - přepínač načeká na načtení rámce ale rozhodne se ihned po načtení jeho hlavičky (a začne rámec přeposílat okamžitě dál). Má menší zpoždění (tzv. latenci).

Propojování různých síťových segmentů

Propojování různých síťových segmentů, které používají různé přenosové technologie na úrovni linkové vrstvy nelze spojit pomocí opakovačů (např. kvůli různým přenosovým rychlostem ale i kvůli dalším odlišnostem). Pomocí mostů/přepínačů někdy ano, někdy ne (ale je to komplikované).

Příklad: Ethernet a Token Ring. Problémy jsou v rozdílné povaze informací o topologii (Ethernetový most chce znát adresy sousedních uzlů, Token Ring-ový most chce znát cesty v síti), v různém způsobu vyjádření stejných informací (status rámce, adresy a zabezpečení jsou vyjádřeny jinak), v neexistenci ekvivalentů (např. v Token Ring-u mohou mít některé rámce vyšší prioritu, v Ethernetu neexistuje analogie), v rozdílné max. velikosti rámců (Ethernet připouští max. 1500 bytů, Token Ring 4000 až 17800 bytů).

Možné řešení: Preklad - data obsažená v rámci jednoho typu se „přeloží“ do jiného tvaru (odpovídajícího jinému typu rámce) - ne vždy je to možné, např. kvůli velikosti.

Možné řešení: „zapouzdření“, "tunelování" - rámec jednoho typu se vloží (jako data) do rámce jiného typu, přenese, a opětně „vybalí“. Jde o obecně použitelnou techniku, lze ji aplikovat na různých úrovních (lze vkládat rámce do rámců, rámce do paketů, pakety do paketů …).

Router (smerovac)

Zarizeni fungujici na urovni sitove vrstvy. Propojuje dve nebo vice siti (ma dve nebo vice sitovych rozhrani - adres). Funguje v prostredi s prepojovanim packetu. Stara se o smerovani packetu (routing) i jejich posilani (forwarding).

Sklada se z **prepojovaciho pole** (switching fabric) a **smerovaciho procesoru** (rating procesor). Procesor urcuje smer dalsiho putovani packetu, pres prepojovaci pole se packet fyzicky prenese ze **vstupni fronty** do **vystupni fronty**.

Smerovac pro urceni dalsi cesty pouziva **smerovaci tabulky** (routing tables – informace o topologii site), pripadne je nepouziva u "nouzovych" metod typu flood nebo random. Adaptivni algoritmy prubezne upravuji smer. tabulky, neadaptivni ne a teda je nutne naplnit tabulky dopredu - staticky.

Smerovaci tabulka obsahuje trojice <do jake site, jakou cenou (v nejake predem stanovene metrice), pres jakeho primeho souseda>.

Při propojení různých segmentů, které používají různé přenosové technologie, na úrovni síťové vrstvy nevznikají "kvalitativní" problémy. Pomocí směrovačů ano a jde o standardní řešení. Síťový paket je vybalen z jednoho typu linkového rámce a vložen do jiného typu linkového rámce. Mohou ale vznikat problémy "kvantitativní". např. tzv. **fragmentace** (když se paket nevejde do max. velkého linkového rámce a musí být sám rozdělen). Protokol IP je na to připraven a dokáže fragmentaci řešit.

Klasifikace algoritmu smerovani

Centralizované smerování - routing (rozhodování o smeru) provádí jeden centralizovaný uzel (route server). Smerovace provadi jen formwarding. S výpadkem route serveru prestává sít fungovat (v praxi se proto moc nepouziva).

Izolované smerování - smerovace provádí routing i forwarding, jednotlivé uzly nespolupracují na hledání optimálních cest.

Distribuované smerování - smerovace provádí routing i forwarding, uzly vzájemne spolupracují na hledání optimálních cest a na aktualizaci svých smerovacích informací.

Hierarchické smerování - soustava sítí se rozdelí na více relativne samostatných oblastí. Uvnitr oblastí se smerování reší samostatne. "Mezi" oblastmi se reší jen "na hrubo".

Zaplavove smerovani (flooding) - varianta izolovaného smerování posli vsem krome zdroje. V praxi se používá jako mechanismus šírení "speciálních" dat. Pro aktualizacni informace, pripadne ve vojenskych sitich, kde je nutna robustnost (vždy najde cil). Problémy jsou s eliminací nadbytecných paketu (dojde z viacerych smerov).

Random - posli komukoli. Pouziva se kdyz se mi zacinaji zahlcovat buffery a nejde to poslat tam, kam to ma jit.

Distribuovane smerovani - nejcasteji používaná varianta smerování. Jednotlivé smerovace vzájemne spolupracují (posílají si aktualizacní informace o zmenách v síti). Dnes prevazne **link state routing** metoda - každý uzel pravidelne monitoruje pruchodnost spojení ke svým sousedum. Každý uzel má úplnou informaci o topologii

celé síti a o pruchodnosti všech spoju. Každý uzel pocítá "za sebe", prípadná chyba neovlivní ostatní uzly. Protokol **OSPF (Open Shortest Path First)** - pomocí paketu protokolu **HELLO** (na astenie suseda) /**ECHO** (na astenie doby odozvy). Každý uzel pravidelne **sestavuje paket**, do kterého dá „namerené“ hodnoty odezvy svých prímých sousedu (ohodnocení hran). Tento paket **rozešle všem ostatním uzlum** (tyto pakety stací rozesílat jen pri zmene!). Každý smerovac postupne „naakumuluje“ zprávy o stavu všech spoju v síti. Výpocet nejkratších cest probíhá lokálne (Dijkstra).

Hiearchicke smerovani - ani algoritmy „link-state“ nejsou vhodné pro opravdu velké síte. Rozdelim sit na bloky, ze kterych se nebudou sirit detailni smerovaci informace, ale jen informace typu informace ve smyslu: "pres mne jsou dostupné síte X.Y.Z až X.Y.W". Tuto sumarizaci provadi "hranicni smerovac". Pres paterni sit je nutno pouzit exterior gateway protokol, v "moji" siti nejaky interior gateway protokol.

Síťový model TCP/IP

Historie

Požadavky na mezipočítačovou komunikaci (od ministerstva obrany USA): Musi to byt - decentralizovane a nejak fungovat i kdyz cast fungovat prestane (bude znicena treba). Musel byt teda robustny. Nebylo pozadovano zabezpeceni, mobilita…

Koncepce paketového přenosu vzniká v 60. letech. Vzniká ARPANET budovaný akademickou sférou za vojenské peníze, aby tuto myšlenku vyzkoušel. Používá experimentální protokol NCP. Po ověření předán akademikům zcela. V roce 1983 přechod na TCP/IP. Z ARPANETU se vyvinul Internet.

Vývoj protokolů na seminářích V. Cerfa pod agenturou ARPA, potom ICCB, IAB, ISOC a W3C, ICANN. Rozděleno do mnoha pracovních skupin. Dnes místo vývoje standardů spíše výběr z hotových.

Dnešní vývoj za peníze komerční sféry, podílí se na něm stále širší odborná veřejnost pomocí standardizací.

Standardizace

RFC = Request for Comment, publikovali doktorandi, původně rozesíláno listovní poštou. Dnes standardy, informace, výzvy, výsledky experimentů…

Nikdy se nemění, ale mohou být zastarány novější verzí. Číslovány pořadovými čísly. Dva druhy: **standard tracks** (standardy) a **off-tracks** (ostatní dokumenty). Aktuálním standardem je vždy dokument, který řeší danou problematiku a má nejvyšší pořadové číslo. Než se něco stane standardem, prochází třemi úrovněmi:

Proposed standard – min. 6 měsíců. 2 nezávislé implementace → **Draft Standard** – min. 3 měsíce, provozní zkušenosti → **Internet Standard**

Filozofie TCP/IP vs. RM ISO/OSI

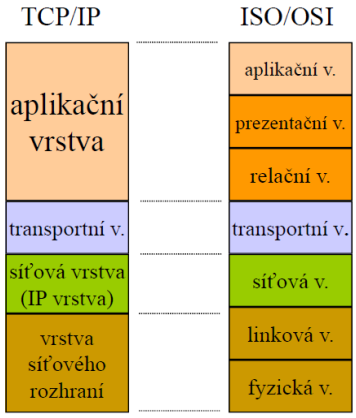
ISO/OSI: sami vymyslíme a standardizujeme, od složitého k jednoduššímu, nejdříve standard a pak výzkum, zda je to prakticky realizovatelné.

Inteligence má být v síti, tedy spolehlivost řešena na úrovni síťové vrstvy, inteligence ve směrovačích, drahé a nepružné, nedává možnost výběru.

TCP/IP: převezmeme rozumné věci, co už vymysleli ostatní (např. jak provozovat IP nad Ethernetem, která standardizovalo ISO), od jednoduššího ke složitějšímu, nejprve praktické ověření a pak až standardizace, nespojované, nespolehlivé, maximální snaha, ale nezaručený výsledek -> aby byla síť vysoce odolná, snaha, aby to fungovalo nad vším.

Inteligence má být uzlech, spolehlivost řešena v transportní vrstvě, aplikace si může vybrat, zda spolehlivost chce přenosová vrstva se má starat o přenos dat, jak nejlépe může a nezdržovat se zbytkem. **Best effort** – nepracuje s prioritami, snaží se co nejrychleji, když nemůže krátí všem stejně (alternativou je garance služeb QoS). Levnější a rychlejší, ale vadí multimediálním aplikacím.

Vrstvy TCP/IP vs. RM ISO/OSI



Vrstva sitoveho rozhrani

TCP/IP ji **nedefinuje**. TCP/IP nezajímá, co je na úrovni vrstvy síťového rozhraní (zda např. Ethernet, Token Ring...). Zajímá ji pouze, jak propojit síťovou vrstvu s vrstvou síťového rozhraní.

Sitova vrstva

Autoři se rozhodovali, zda vytvoří jednotnou nadstavbu nad soustavou vzájemně propojených sítí (přenosový protokol na úrovni síťové vrstvy bude mít stejné vlastnosti všude a bude poskytovat stejné služby), nebo zda nadstavba nebude všude stejná. To by mohlo být efektivnější, ale zase bychom se museli zabývat různými odlišnostmi…

Autoři se rozhodli se pro **jednotnou pokličku**. Jde o jednotnou nadstavbu, která je **tvořena přenosovým protokolem IP**:

- Je nespojovaný, nespolehlivý, s maximální snahou.
- Jednotné adresování 32b IP adresami (nemají žádný reálný vzor).
- IP protokol i vyšší vrstvy "vidí" maximální velikost linkového rámce (skrz parametr **MTU**, Maximum Transfer Unit) a měli by jej respektovat tak aby nedocházelo ke zbytečné fragmentaci při přenosech – to je jedina výjimka, ked sa IP moze spravat v ruznych sietiach ruzne a netvori teda jednotnu poklicku.

Transportni vrstva – **End-to-end komunikace**. Zajišťuje multiplex/demultiplex pomocí čísel portů. Nabízí dva protokoly:

- **TCP** - funguje spojovaně, spolehlivě, je velmi adaptivní (dokáže se průběžně přizpůsobovat různým podmínkám přenosu) a komplikovaný. Aplikace dostává bytový proud, ale data jsou přenášena po blocích.
- **UDP** - je pouze jednoduchou nadstavbou nad síťovým protokolem IP, funguje nespojovaně, nespolehlivě. Data přicházejí po blocích.

Aplikacni vrstva

Obsahuje protokoly (**RPC a XDR**) zastupujuce relacnu a presentacnu vrstvu (jako v ISO/OSI), ale ich pouzitie je nepovinne a nie velmi caste. Prakticky vsetky aplikacie sú založené na architekture **klient/server**.

Objevují se ale i aplikace, pro které není TCP/IP principiálně vhodný. Např. Vysílání - stejná data z jednoho zdroje na mnoho klientů, tj. Multicasting. To TCP/IP neumí, pokusně to zkoušela služba MBONE, ale neúspěch. Zavedeno do Ipv6.

Přístupy ke QoS v TCP/IP

Multimediální aplikace potřebují dostávat data s malým a stejným zpožděním. Tedy nutná podpora QoS. Možná řešení:

- **Kvantitativní zvyšování kapacity (linky, směrovače)** - fungování na principu maximální snahy zůstává, pouze se zmenšuje pravděpodobnost, že dojde k nedostatku zdrojů.
- **Kvalitativní – zavedení podpory QoS** - drahé a obtížné, ale zlepšení je garantované. Možné řešení:
    - Prioritizace: přenosy s vyšší prioritou dostávají kvalitnější obsluhu, DiffServ, nebo MPLS.
    - Rezervace: vyhrazení přenosové a přepojovací kapacity, IntServ, nebo RSVP.
    - Hrubá síla: princip max. snahy se nemění, pouze se předimenzují kapacity.
    - Na úrovni transportní vrstvy: RTP, RTCP.
    - Na úrovni aplikační vrstvy: bufferování.

Řešení mobility a bezpečnosti

IP adresy nejsou mobilní – nelze je přenášet mezi sítěmi (nelze je vytrhnout z mateřské sítě, protože se směruje podle síťové části adresy). Lze řešit přidělením nové adresy, skrze agenty a tunely (přesměrování na novou adresu).

IP Mobility Support – pomocí agentů, neviditelné pro vyšší vrstvy, nutnost statické adresy agenta. Určen pro občasný přesun, ne pro neustálé pohybování.

TCP/IP neposkytuje zabezpečení, protože to nebylo v původním zadání. Data nejsou chráněna proti odposlechu, ztrátě, ani změně (poslední dvě jen UDP). Zabezpečení si musí zajistit sama aplikace – aplikační úroveň.

Časem vznikl **IPSEC** – bezpečnost na síťové vrstvě. Definován v několika RFC, není to jeden protokol. Původně vznikl pro IPv6, ale začal se používat už teď. Zajišťuje:

- Důvěrnost
- Integritu
- Výměnu klíčů
- Obranu před opakováním odposlechnutých zpráv

Má dva režimy fungování: **transport mode** (zabezpečovací údaje se vloží do hlavičky IP datagramu) a **tunnel mode** (celý IP datagram se vloží do zabezpečeného datagramu).

IP adresy

Predpokladaji se dva typy uzlu v siti - koncove (pc, tiskarny, servery,...) a routery, ktere spojuji jednotlive "site" (dvě síťová rozhrání – dve adresy). Každý uzel musí mít originální adresu, aby mohl komunikovat (dnes platí pouze pro počítače na internetu, ne ty za firewallem). IP adresy mají 32b a mají všude stejný tvar. Jejich tvar vychází vstříc potřebám TCP/IC protokolu, směrování (část adresy určena sítí, kde se počítač nachází) a představě katenetového modelu.

IP adresa = síťová část + adresa uzlu

Síťovou část tvoří vyšší bity, adresu uzlu nižší. Dříve byla hranice mezi nimi pevně dána (3 : 1), dnes je volitelná.

Přidělování IP adres

Musí být respektováno rozdělení na sítě, uzly ve stejné síti musí mít stejnou síťovou část, uzly v různých sítích musí mít různé síťové části adresy. IP adresy patří rozhraním, ne uzlům. Nutnost přidělovat IP adresy po blocích, bez ohledu na to, kolik se jich využije. IP adresy nemají žádný ekvivalent v linkových adresách, překládají se např. pomocí **ARP** protokolu. Žádná adresa nesmí být přidělena dvakrát (dnes existují výjimky). Existuje centrální autorita, která je přiděluje. IANA rozdělila bloky adres regionálním přidělovatelům.

Autoři původně předpokládali malý počet velkých sítí, střední počet středních sítí a velký počet malých sítí. Takže velikost síťové části adresy má 3 polohy:

- Třída **A** - velké sítě, 8:24 – 1-126.x.x.x (adresa začíná 0)
- Třída **B** - středně velké sítě, 16:16 – 128.0.x.x – 191.255.x.x (adresa začíná 10)
- Třída **C** - malé sítě, 24:8 – 192.0.0.x – 223.255.255.x (adresa začíná 110)

Existují ještě dvě speciální třídy:

- Třída **D**: multicasting 224.0.0.0 až 239.255.255.255. (adresa začíná 1110)
- Třída **E**: vyhrazeno pro budoucí použití 240.0.0.0 až 255.255.255.255. (adresa začíná 1111).

Možnosti řešení úbytku IP adres

Hrozilo **vyčerpání** adres, protože se muselo přidělovat po blocích, které byly zbytečně velké. Začal akutně hrozit nedostatek adres, hledalo se řešení, počítalo se, jak dlouho adresy ještě vydrží.

Zjemnění přidělování by se dalo dosáhnout posunem hranice mezi sítovou a uzlovou adresou, ale na to nebyly původní mechanismy připravené. Tedy vznikla **maska**.

Řešení:

- Přidělování násobků největších nižších kvant (např. 4 x třídu C).
- Subnetting – lokální dělení skupin IP adres v podsítích.
- Privátní IP adresy – ven pomocí NAT nebo firewallů.
- CIDR – přidělování adres po libovolných kvantech, jejichž velikost je násobek 2 (dnes používáno).

IPv6

Naslednik IPv4 (ten prevazne pouzivany nyni). Asi nejdulezitejsi zmenou je rozsireni adresoveho prostoru z 32 bitu na 128 a tim vyreseni problemu s nedostatkem adres "na vzdy". Obsahuje ale i jine nove moznosti. Nejedna se o zasadni zmeny od IPv4, spise, podle filosofie protokolu TCP/IP o vylepseni, ktere si "zadaji uzivatele":

- Podpora QoS, multicast, anycast, mobilitu, autokonfigurace.
- Nemá checksum v hlavičce (kde se neustále mění TTL/hop limit), místo toho spoléhá na vrstvy okolo. Routery tak nemusejí neustále počítat měnící se součty.
- Pakety se nefragmentují na cestě, ale jen u odesílatele, předpokládá se použití Path MTU discovery.

Adresace v IPv6

IPv6 je trvalejším riešením nedostatku adries – zatial sa ale rozširuje velmi pozvolna. Adresa v IPv6 má dlžku 128 bitov (oproti 32), co znamená cca. 6×10^{23} IP adries na $1m^2$ zemského povrchu – umožnuje teda, aby každé zariadenie na zemi malo vlastnú jednoznacnú adresu. Adresa IPv6 sa zapisuje ako osem skupín po štyroch hexadecimálnych číslach (napr. 2001:0718:1c01:0016:0214:22ff:fec9:0ca5) – pricom úvodné nuly v číslach je možné vynechat. Ak po sebe nasleduje niekolko nulových skupín, je možné použit len znaky :: – napr. ::1 miesto 0000:0000:. . . . . .:0001. Toto je možné použit len raz v zápise adresy. RFC 4291 zavádza 3 typy adries:

- **Individualne / Unicast adresy** - reprezentuje jednotlivé síťové rozhraní. Paket zaslaný na unicast adresu je doručen konkrétnímu počítači.
- **Skupinove / Multicast adresy** - adresy jsou používány k definování množiny rozhraní obvykle patřících různým uzlům, nikoli pouze jednomu. Paket zaslaný na multicast adresu je protokolem doručen všem rozhraním určeným touto adresou. Multicast adresy mají prefix FF00::/8 a jejich druhý oktet určuje dosah adresy, tzn. rozsah v jakém je multicast adresa zviditelněna. Běžně využívány jsou rozsahy místní linky (0x2), místní stránky (0x5) a globální (0xE).
- **Vyberove / Anycast adresy** - adresy jsou také přiřazeny více než jednomu rozhraní, patřící rozdílným uzlům. Nicméně paket vyslaný na anycast adresu je obvykle doručen pouze jednomu z členských rozhraní, typicky „nejbližšímu“ vzhledem k představě směrovacího protokolu o vzdálenosti. Anycast adresy nemohou být snadno identifikovány, mají strukturu bežné unicast adresy a liší se pouze zaváděním do směrovacího protokolu na více místech v síti.

IPv6 neobsahuje všesmerové (broadcast) adresy. Byly nahrazeny obecnejším modelem skupinových adres a pro potreby dorucení dat všem zarízením pripojeným k urcité síti slouží speciální skupinové adresy (napr. ff02::1 oznacuje všechny uzly na dané lince).

IPv6 zavádí také koncepci dosahu (scope) adres. Adresa je jednoznacná vždy jen v rámci svého dosahu. Nejcastejší dosah je pochopitelne globální, kdy adresa je jednoznačná v celém Internetu. Krome toho se casto používá dosah linkový, definující jednoznacnou adresu v rámci jedné linky (lokální síte, napr. Ethernetu).

Propracovanou strukturu dosahu mají skupinové adresy (viz níže). Adresní prostor je rozdelen následovne:

prefix	**význam**
::/128	neurcená
::1/128	smycka (loopback)
ff00::/8	skupinové
fe80::/10	individuální lokální linkové
ostatní	individuální globální

Výberové adresy nemají rezervovánu svou vlastní cást adresního prostoru. Jsou promíchány s individuálními a je otázkou lokální konfigurace, aby uzel poznal, zda se jedná o individuální ci výberovou adresu.

Strukturu globálních individuálních IPv6 adres definuje RFC 3587. Je velmi jednoduchá a de facto odpovídá (až na rozmery jednotlivých cástí) výše uvedené strukture IPv4 adresy.

n bitu	**64-n bitu**	**64 bitu**
globální smerovací prefix	adresa podsíte	adresa rozhraní

Globální smerovací prefix je de facto totéž co adresa síte, následuje adresa podsíte a pocítace (presneji sítového rozhraní). V praxi je adresa podsíte až na výjimky 16bitová a globální prefix 48bitový. Ten je pak pridelován obvyklou hierarchií, jejíž stávající pravidla jsou:

- první dva bajty obsahují hodnotu 2001 (psáno v šestnáctkové soustave)
- další dva bajty prideluje regionální registrátor (RIR)
- další dva bajty prideluje lokální registrátor (LIR)

Reálná struktura globální individuální adresy tedy vypadá následovne:

16 bitu	**16 bitu**	**16 bitu**	**16 bitu**	**64 bitu**
2001	prideluje RIR	prideluje LIR	adresa podsíte	adresa rozhraní

Adresa rozhraní by pak mela obsahovat modifikovaný EUI-64 identifikátor. Ten získáte z MAC adresy jednoduchým postupem: invertuje se druhý bit MAC adresy a doprostred se vloží dva bajty obsahující hodnotu fffe. Z ethernetové adresy 00:14:22:c9:0c:a5 tak vznikne identifikátor 0214:22ff:fec9:0ca5. Adresy zacínajúce hodnotou ff sú tzv. ”skupinové adresy” – štyri nasledujúce bity v nej obsahujú príznaky, dalšie štyri potom dosah (napr. interface-local, link-local, admin-local, site-local, organization-local, global. . .) IPv6 dalej podporuje QoS a bezpecnost (IPsec).

DNS

Symbolická jména se používala již v ARPANETu. Existovala centrální autorita, která vedla evidenci symbolických jmen a převodní tabulku. Zajišťovala aktualizace a distribuovala tabulku všem zájemcům. Tato symbolická jména mohla být jednorozměrná (**plochý systém** – jednorozměrnost a centralizace), nemusela být dělena na složky. Celé to mohlo fungovat pro malý počet uzlů a málo změn. Pro větší sítě nutná změna – necentralizované a dostatečně škálované.

Dnes existuje hierarchie (strom) dílčích jmenných prostorů – těm se říká **doména**. (doména je prvek v rámci hierarchického členění jmenného prostoru). Je to okruh působnosti někoho, kdo má právo přidělovat symbolická jména. Není předepsáno, čím musí být ohraničena, kromě domén nejvyšší úrovně. Ty jsou ohraničeny podle států, nebo charakteru subjektu.

Data by měl být uchovávána co nejblíže místu, kde „žijí“ - kde se mění. Neměl by to být jeden centrální prvek, kvůli bezpečnosti a vytížení, dotazy by měly být zodpovězeny tam, kde se data nachází.

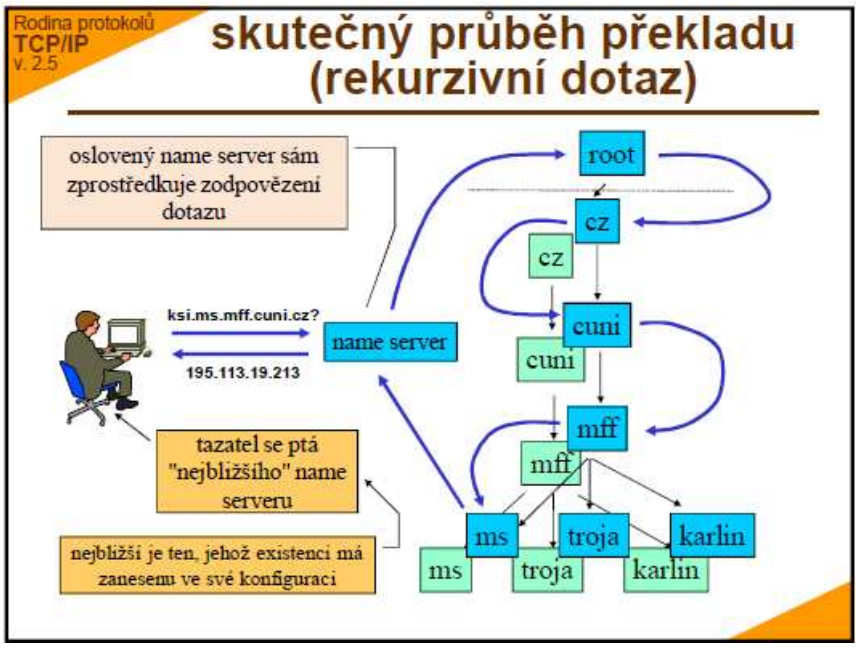
Zóna je oblast tvořená skupinou domén, nad kterou má konkrétní subjekt autoritu. Všechny údaje týkající se jedné zóny jsou na jednom místě, v jednom souboru (zone file). Zóny lze rozšiřovat dceřinými doménami, nebo zmenšovat delegováním dceřiných domén jinam. Struktura name serverů logicky odpovídá struktuře domén, prakticky struktuře zón (1 zóna 1 name server).

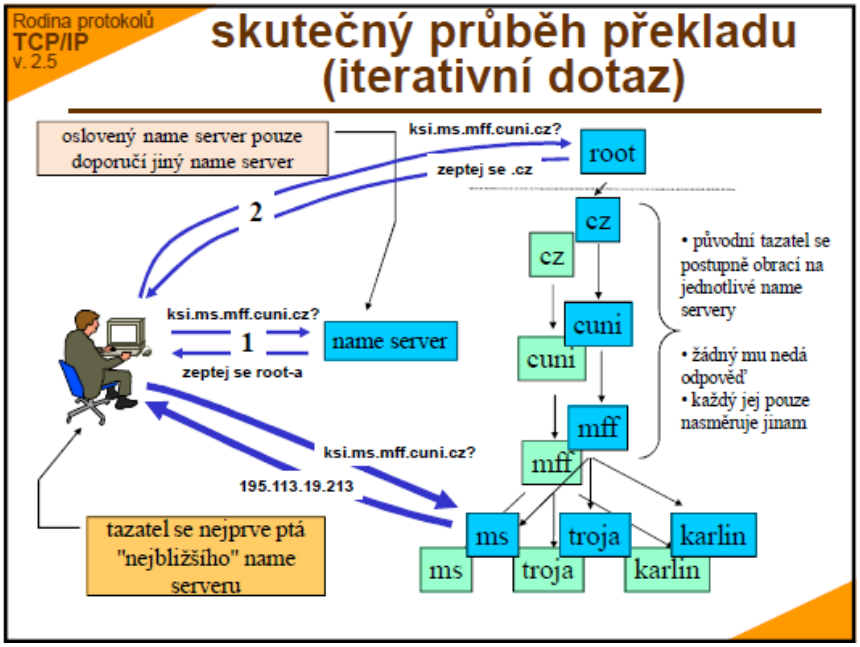
V rámci každé domény se pamatují informace o propojení jmen a adres. Tyto informace nejsou uchovávány centrálně, ale distribuovaně. Name server vždy přísluší nějaké doméně, má k dispozici data odpovídající této doméně a odpovídá na dotazy týkající se jí. Každá doména má svůj name server, ale jeden name server může sloužit pro více domén.

- **Kořenové name servery** jsou ty, které si pamatují adresy všech name serverů domén nejvyšší úrovně, je jich na světě 13 na různých platformách a v různých zemích.
- **Primární name server** má přímo k dispozici data o doméně. **Sekundární server** má data zprostředkovaně od primárního name serveru, měl by být umístěn v jiné síti.

Princip překladu symbolického doménového jména na IP adresu

Nejdříve se zeptám kořenového name serveru, jehož adresa je všeobecně známa. Postupně je odkazován na další name servery „níže“, až se dostane k odpovědi: Skutečný průběh může být dvojího typu: **iterativní dotaz** a **rekurzivní dotaz**.


Rodina protokolů
TCP/IP
v. 2.5
skutečný průběh překladu
(rekurzivní dotaz)
oslovený name server sám zprostředkuje zodpovězení dotazu
ksi.ms.mff.cuni.cz?
195.113.19.213
name server
tazatel se ptá "nejbližšího" name serveru
nejbližší je ten, jehož existenci má zanesenu ve své konfiguraci
root
cz
cz
cuni
cuni
mff
mff
ms
troja
karlin
ms
troja
karlin


Rodina protokolů
TCP/IP
v. 2.5
skutečný průběh překladu
(iterativní dotaz)
oslovený name server pouze doporučí jiný name server
ksi.ms.mff.cuni.cz?
zeptej se .cz
2
ksi.ms.mff.cuni.cz?
1
zeptej se root-a
name server
ksi.ms.mff.cuni.cz?
195.113.19.213
tazatel se nejprve ptá "nejbližšího" name serveru
root
cz
cz
cuni
cuni
mff
mff
ms
troja
karlin
ms
troja
karlin
• původní tazatel se postupně obrací na jednotlivé name servery
• žádný mu nedá odpověď
• každý jej pouze nasměruje jinam

Přenosové služby počítačových sítí: spolehlivé a nespolehlivé

Spolehliva sluzba

Ten, kdo data prenasi zodpovida za doruceni (proto je nutna detekce chyb, zadosti o opetovny prenos).

Nespolehliva sluzba

Je ji jedno, zda se to doruci nebo ne. Se spolehlivosti je spojena nenulova rezie, ktera muze byt nezadouci. Napr. multimedialni prenos potrebuje dodavat pravidelne hodne dat a pripadna ztrata casti z nich boli mene nez zdrzeni vetsiny kvuli znovuposlani.

Best effort

Typ prenosu, kdy se sit "snazi" a pokud to uz nejde, tak jsou pozadavky stejnomerne kraceny. Napriklad sitovy protokol IP.

Quality of Service

Obecné oznacení pro variantu, kdy prenosová sluzba dokáže rozlišovat mezi jednotlivými prenosy a nabízet jim ruznou "kvalitu prenosu" (QoS). Muze nabizet garanci sluzeb (resi se rezervaci zdroju) a take nemusi (vytvori len prioritnu frontu).

Zajištění spolehlivosti

Může být realizováno na kterékoli vrstvě kromě fyzické. (TCP/IP: řeší se až na transportní vrstvě - protokol TCP, RM ISO/OSI: očekává se od všech vrstev, počínaje linkovou vrstvou). Podmínkou je schopnost detekce chyb schopnost rozpoznat, že došlo k nějaké chybě při přenosu (musí být použit vhodný detekční mechanismus). Realizace opravy bud pomoci použití **samoopravných kódů** (např. Hammingovy kódy, problémem je velká míra redundance, která zvyšuje objem přenášených dat - používá se jen výjimečně) nebo **pomocí potvrzování** (příjemce si nechá znovu zaslat poškozená data).

Detekce chyb

K přenášeným datům se připojí "zabezpečovací údaj". Odesilatel podle obsahu přenášeného bloku vypočítá "zabezpečovací údaj", který připojí k datovému bloku a přenese. Příjemce znovu vypočítá "zabezpečovací údaj" (stejným postupem) a porovná jej s přijatým zabezpečovacím údajem.

Parita

Paritní bit - bit přidaný navíc k datovým bitům. Několik typu: **Sudá parita** (paritní bit je nastaven tak, aby celkový počet 1 byl sudý), **lichá parita** (aby byl lichy), **příčná parita** (po jednotlivých bytech/slovech), **podélná parita** (parita ze všech stejnolehlých bitů všech bytů/slov).

Kontrolní součet

Jednotlivé byty/slova/dvojslova tvořící přenášený blok se interpretují jako čísla a sečtou se (muzou se i XOR-ovat) a výsledný součet se použije jako zabezpečovací údaj. Obvykle se použije jen část součtu, například nižší byte či nižší slovo.

CRC (Cyclic Redundancy Check)

Posloupnostbitů, tvořící blok dat, je interpretována jako **polynom nad tělesem charakteristiky 2**, kde jednotlivé bity jsou jeho koeficienty. Tento polynom je vydělen jiným polynomem, tzv. **charakteristickým polynomem** (např. CRC-16: $x^{16}+ x^{15}+ x^2+1$). Výsledkem je podíl a zbytek(ten se pouzije v roli zabezpečení). Spolehlivost CRC kódů se opírá o silné teoretické výsledky z algebry. Samotný výpočet CRC-kódu (zbytku po dělení) je velmi jednoduchý a může být snadno implementován v HW, pomocí XOR-hradel a posuvných registrů.

Potvrzování (acknowledgement)

Jde o obecnější mechanismus, který slouží (může sloužit) více účelům současně: **zajištění spolehlivosti** (umožňuje, aby si příjemce vyžádal opakované zaslání poškozeného rámce) a **řízení toku** (aby příjemce mohl regulovat tempo, jakým mu odesilatel posílá data). Existuje více možných způsobů jak realizovat potvrzování: **kladné a záporné potvrzování** (potvrzují se správě resp. chybně přijaté bloky), **jednotlivé a kontinuální potvrzování** (podle toho, zda odesilatel vždy čeká na potvrzení nebo odesílá "do fronty"), **samostatné a nesamostatné (piggybacking) potvrzování** (zda potvrzení cestuje jako samostatný rámec/paket, nebo je vnořeno do datového rámce/paketu), **metoda okénka**…

Stop & Wait ARQ - jde o samostatné jednotlivé potvrzování. Odesilatel odešle datový rámec a čeká na jeho potvrzení (kladné či záporné). Příjemce odešle potvrzení. Podle druhu potvrzení odesilatel buď opakuje přenos téhož rámce, nebo vyšle další rámec. Nebo čeká na vypršení timeoutu, které interpretuje stejně jako záporné potvrzení.

Kontinuální potvrzování (continuous ARQ)

Idea: odesilatel bude vysílat datové rámce „dopředu“, a příslušná potvrzení bude přijímat průběžně, s určitým zpožděním. Existuji dve varianty jak se má odesilatel zachovat, když dostane záporné potvrzení a už mezitím odeslal několik dalších rámců:

- **Selektivní opakování -** odesilatel znovu vyšle jen ten rámec, který se poškodil. Další rámce, které se mohly úspěšně přenést, se nevysílají znovu. Příjemce musí úspěšně přijaté rámce ukládat do bufferů.
- **Návrat zpět -** řeší situaci kdy záporné potvrzení (informace o poškození určitého rámce) přijde se zpožděním - v době kdy již byly odeslány nějaké další rámce. Reší se „zahozením“ již odeslaných rámců (zacnem vysielat znova od poskodeneho ramca). Příjemci stačí čekat na nový přenos poškozeného rámce, a pak pokračovat v příjmu dalších rámců.

Metoda okénka (sliding window)

Idea: spojit potvrzování s řízením toku na úrovni rámců. Odesilatel si udržuje vysílací „okénko“. Velikost okénka udává, kolik rámců smí vyslat "dopředu" aniž by je měl ještě potvrzené. Velikost okénka určuje odesilatel, dle „chování“ a vlastností sítě (většinou stanoví maximální velikost) a příjemce, podle svých možností (např. dostupnosti bufferů). Ovlivňováním velikosti okénka může příjemce efektivně regulovat tok dat směrem k sobě! Zmenšením okénka na nulovou velikost lze zastavit vysílání.

Přenosové služby počítačových sítí: Spojované a nespojované

Spojovana komunikace

1. Nejdrive se navaze spojeni v rámci **navázání spojení** je nalezena (a vyznacena) trasa penosu.
2. **Probiha komunikace** (po trase (ceste), nalezené při navazování spojení.
3. Na konci je treba **spojení zase ukoncit** (rozvázat) a vrátit pridelené zdroje, zrušit vytycenou trasu.

Komunikace je stavova (alespon neni spojeni x je spojeni). Je treba osetrit prechod mezi jednotlivymi stavy a nestandardni situace (jeden ucastnik spadne apod.) Zachovava poradi (packety se nemohou sprehazet, kdyz cestuji stejnou cestou)

Analogie - telefonni hovor (vytocim, mluvim, polozim).

Nespojovana komunikace

Nenavazuji spojeni, neoveruji, zda druha strana existuje, neni treba ukoncovat spojeni. Komunikace probiha formou posilani zprav – diagramu. Je bezestavova. **Datagram** obsahuje v hlavicce plnou adresu příjemce nezarucuje poradi doruceni trasa pro datagram je hledana vzdy znovu.

Analogie - listovni posta.

Spolehlivost, Flow control, Congestion kontrol a protokol TCP/IP

Kedže TCP/IP funguje nad obecne nespojovanými a nespolahlivými médiami, spolehlivost ktorú TCP poskytuje nie je „skutocná“, ale len „softvérovo emulovaná“ – medzilahlé uzly o spojení nic nevedia, fungujú nespojovane (pre komunikáciu sa používa sietová vrstva, transportná „existuje“ iba medzi koncovými uzlami). Je teda nutné ošetrit napr. nespolahlivost infraštruktúry (strácanie dát, duplicity – pricom stratit sa môže aj žiadost o vytvorenie pripojenia, potvrdenie. . .) a reboot uzlov (uzol stratí históriu, je potrebné ošetrit existujúce spojenia. . .).

Používa sa celá rada techník, kde základom je kontinuálne potvrdzovanie: príjemca posiela kladné potvrdenia; odosielatel po každom odoslaní spúšta casovac a ak mu do vypršania nepríde potvrdenie, posiela dáta znovu. Potvrdzovanie nie je samostatné ale vkladá sa do paketov cestujúcich opacným smerom – **piggybacking**.

TCP priebežne kontroluje „dobu obrátky“ a vyhodnocuje vážený priemer a rozptyl dôb obrátky. Cakaciu dobu (na potvrdenie) potom vypocítava ako funkciu tohoto váženého priemeru a rozptylu. Výsledný efekt je potom ten, že cakacia doba je tesne nad strednou dobou obrátky. V prípade konštantnej doby obrátky sa cakacia doba približuje strednej dobe obrátky; ak kolíše, cakacia doba sa zväcšuje.

Dáta v TCP sa príjímajú/posielajú po jednotlivých byteoch – interne sa však bufferujú a posielajú až po naplnení buffera (pricom aplikácia si môže vyžiadat okamžité odoslanie – operácia PUSH). TCP si potrebuje oznacovat jednotlivé byty v rámci prúdu (kedže nepracuje s blokmi) – napr. kvôli potvrdzovaniu; používa sa na to 32-bitová pozícia v bytovom prúde (zacína sa od náhodne zvoleného císla).

TCP sa snaží riadit tok dát – aby odosielatel nezahlcoval príjemcu a kvôli tomu nedochádzalo k stráte dát. Podstata riešenia je tzv. **metóda okienka**. Okienko udáva velkost volných bufferov na strane prijímajúceho a odosielatel môže posielat dáta až do „zaplnenia“ okienka. Príjemca spolu s každým potvrdením posiela aj svoju ponuku – údaj o velkosti okienka (**window advertisment**), ktorý hovorí kolko ešte dát je schopný prijat (naviac k práve potvrdeným). Znovu – používa sa metóda kontinuálneho potvrdovania.

Väcšina strát prenášaných dát je skor kvoli zahlteniu ako chybám HW a transportné protokoly môžu nevhodným chovaním zhoršovat dôsledky. TCP každú stratu dát chápe ako dôsledok zahltenia – nasadzuje opatrenia proti zahlteniu (congestion control). Po stráte paketu ho pošle znovu ale neposiela dalšie a caká na potvrdenie (tj. prechod z kontinuálneho potvrdzovania na jednotlivé) vysiela menej dát ako mu umožnuje okienko). Ak príde potvrdenie vcas, zdvojnásobí množstvo odosielaných dát – a tak pokracuje kým nenarazí na aktuálnu velkosti okienka (postupne sa tak vracia na kontinuálne potvrdzovanie).

Dôležitou vlastnostou je aj korektné chovanie pri naväzovaní a rušení spojenia (v prostredí, kde môže dôjst k spomaleniu, strate, duplicite. . .) – používa sa tzv. **3-fázový handshake:**

1. Klient pošle serveru SYN paket (v pakete je nastavený príznak SYN) spolu s náhodným sequence number (X).
2. Server tento paket prijme, zaznamená si sequence number (X) a pošle spät paket SYN-ACK. Tento paket obsahuje pole Acknowledgement, ktoré oznacuje dalšie císlo (sequence number), ktoré tento host ocakáva (X+1). Tento host rovno vytvorí spätnú session s vlastným sekvencným císlom (Y).
3. Klient odpovie so sekvencným císlom (X+1) a jednoduchým Acknowledgement císlom (Y+1) – co je sekvencné císlo servera+1.

Pak už spojení považováno za navázané. Rušenie spojenia funguje podobne, posílají se pakety FIN (finish), FIN+ACK a ACK. Pokud více než nejaký urcitý počet pokusu o odeslání (po spocítaných time-outech) jednoho z 3-way handshake paketu selže (druhá strana neodešle to, co melo následovat), spojení se považuje za prerušené (i u navazování, i u rušení).

Dalsie Pojmy

Proudovy prenos (streaming)

Predavano po jednotlivych bitech/bytech. Data nemusi mit hlavicku, prijemce je ten na druhe strane kanalu. Predpoklada se spojovana komunikace.

Blokovy prenos

Prenaseno po blocich. Prenost muze byt spojovany i nespojovany. Blok je nazyvan podle vrstvy, ve ktere je prenos realizovan, zpusobu prenosu nebo velikosti bloku:

Packet - sitova vrstva, **Ramec** (frame) - linkova vrstva, **Zprava** - aplikacni vrstva

Datagram - obecny nespojovane prenaseny blok (ramec je datagram).

Velikost bloku je promenna, ale shora omezena.

Prepojovani okruhu

Je zpusob prenosu. Jednotlive uzly site "vyriznou" z dostupne kapacity tolik, o kolik si komunikujici strany reknou (a taky to pak nauctuji :)). Vyrabi iluzi jednoho dratu mezi zdrojem a prijemcem, ktery ma vsude stejnou kapacitu. Odhadnutelne zpozdeni na ceste, vyuziva se ve svete telekomunikaci, protoze je vhodne pro multimedia, malo ve svete pocitacu (snad jen seriove komunikace). Prenos muze byt proudovy i blokovy.

Prepojovani packetu

Nic se nevyhrazuje, stale je snaha vyuzit vsechny dostupne zdroje (best effort). Pouziva se prakticky ve vsech sitich, vhodne pro datove prenosy. Datagramy museji obsahovat adresy odesilatele a prijemce. Ma smysl jen blokovy prenos, proudovy se emuluje. Diky slozitejsi logice zpracovani v uzlech ma vetsi prenosove zpozdeni nez prepojovani okruhu a je nerovnomerne. Uzly funguji na principu Store & Forward Muze fungovat spojovane (virtualni okruhy) i nespojovane(datagramova sluzba) efektivni. Prenos muze byt pouze blokovy.

Přenos a sdílení dat

Přenos souborů

Jde o netransparentní řešení. Uživatel/aplikace si **uvědomuje**, že soubor se nachází na vzdáleném počítači. **Musí** proto explicitně **znát umístění** souboru na vzdáleném počítači. **Musí explicitně podnikat** určité kroky ke zpřístupnění souboru. V rámci TCP/IP řeší protokol **FTP**.

Sdílení souborů

Jde o transparentní řešení. Uživatel/aplikace si **neuvědomuje**, že soubor se nachází na vzdáleném počítači. **Nemusí** proto explicitně **znát umístění** souboru na vzdáleném počítači. **Nemusí ani podnikat** žádné explicitní kroky ke zpřístupnění souboru. V rámci TCP/IP řeší protokol **NFS**. Alternativy: AFS (Athena File System), CIFS (Common Internet File System).

FTP (File Transfer Protocol)

Implicitně chápe soubor jako dále nestrukturovaný, označováno jako **file structure** (alternativy: **record structure** (posloupnost stejne velkych zaznamu), **page structure** (mnozina stranek)). Implicitně je obsah souboru přenášen jako spojitý proud dat, oznacovano jako **stream mode** (alternativy: **block mode** (je mozne medzi bloky vkladat zarazky a při vypadku spojeni se vratit k posledni), **compressed mode** (pouzita jednoducha metoda komprese)).

Vychazi z modelu **klient/server**. Klient je typicky aplikačním programem. Server obvykle systémovým procesem. Zajištění potřebných funkcí v rámci FTP je rozděleno mezi dva subjekty (procesy běžící jak u klienta, tak u servru):

- **Interpret protokolu** (**PI, Protocol Interpreter**) - existuje trvale.
- **Přenosový proces (DTP, Data Transfer Process)** - vzniká až na základě konkrétního požadavku.

Používají se dvě různá spojení:

- **Řídící** (pro přenos příkazů) - je jedno pro všechny přenášené soubory. Iniciuje a navazuje ho klient ze svého (dynamicky přiděleného) portu na port 21. Toto spojení spracovává (má na starosti) interpret protokolu.
- **Datové** (pro přenos souborů) - iniciuje a navazuje ho server ze svého portu 20 na port klienta, ze kterého bylo navázáno řídící spojení. V **passive mode** datové spojení nenavazuje server, ale klient (kvuli firewallum). Toto spojení má na starosti přenosový proces.

Uživatelé a FTP

FTP je “**user aware**” - uvědomuje si existenci uživatelů. Server při přístupu k místním souborům vždy vystupuje jménem konkrétního uživatele. Uživatelé se v rámci "FTP relace" musí přihlásit místním uživatelským jménem a prokázat platným heslem. Má-li být něco veřejně přístupné, uživatelé se hlásí jako "**anonymous**" a heslo není významné.

Řídící jazyk

FTP definuje vlastní řídící jazyk (příkazy řídícího jazyka jsou přenášeny řídícím spojením). Prikazy lze rozdelit na: **řízení přístupu** (např. pro zadání uživatelského jména a hesla)**, nastavení parametrů** (např. pro změnu implicitních čísel portů, pro nastavení režimu přenosu apod)**, výkonné příkazy.** Každý prikaz vyvola alespoň jendu odpoved, které jsou ciselneho charakteru (trojmístné číslo). První číslice vyjadřuje celkový charakter odpovědi, druhá číslice upřesňuje odpověď a třetí ji ještě blíže specifikuje. “Hloupý” klient či server se může spokojit jen s první číslicí.

NFS (Network File System)

V rámci TCP/IP není jediný, ale je nejrozšířenější. Pochází od firmy Sun Microsystems. Připouští, aby klient i server stáli na různých platformách. NFS je bezestavovým protokolem a teda připouští pouze použití **idempotentních** (lze je opakovat se stejnym efektem) operací mezi klientem a serverem. Bezestavovost NFS je klíčem k **velké robustnosti** a úspěchu NFS.

Problémy NFS

Není však možné realizovat všechny požadavky na přístup k souborům pomocí idempotentních operací. Např. provést OPEN (otevření), a soubor ponechat otevřený, obdobně pro CLOSE. NFS to řeší tak, že pro potřeby vyřízení každého jednotlivého požadavku soubor nejprve otevře, a pak jej zase ihned zavře. Některé případy však nelze ani obejít jako v případě OPEN a CLOSE. např. **APPEND** (přidání za aktuální konec souboru) **NFS řeší zákazem takovýchto operací.**

NFS si klade za cíl být použitelný na různých platformách proto se nemůže vázat na konvence žádné specifické platformy. Při odkazech na soubory nemůže používat specifikace přístupových cest typu /usr/bin/neco.txt (které jsou závislé na platformě). **Přístupové cesty sestavuje vždy až klient** podle místních konvencí, server používá vždy jen "jednorozměrná" jména souborů a adresářů. Počáteční "přimontování" (mount) části adresářového stromu není bezestavové - **NFS řeší vyčleněním do tzv. mount servru** (ten dělá vše co není bezstavové).

Systémová identifikace - handle

Pro odkazy na konkrétní soubory se v rámci NFS používají tzv. systémové identifikace (**file handles**):

- jednoznačně identifikuje buď soubor, nebo adresář.
- má absolutní povahu (a nikoli relativní, NFS nezná pojem aktuálního adresáře).
- Přiděluje ji server, klient je pouze používá.
- Pro klienta je identifikátorem (dále nedělitelnou posloupností bitů). Pro server obsahuje tři složky identifikující: **systém souborů**, **soubor** a **instanci souboru**.

Fungování NFS z pohledu klienta

Když klient vlastní systémovou identifikaci adresáře, může si vyžádat výpis jeho obsahu. Součástí výpisu je seznam znakových řetězců, popisujících jednotlivé soubory a podadresáře. Klient si může vyžádat na serveru systémovou identifikaci zadaného souboru v adresáři. Serveru přitom musí předat systémovou identifikaci adresáře, a znakový řetězec popisující soubor či podadresář.

Klient musí získat alespoň jeden vstupní bod do systému souborů serveru (alespoň jednu systémovou identifikaci). Pak má k dispozici prostředky pro procházení celého stromu. Poskytnutí "prvního" handle ale není v silách NFS serveru. **První handle poskytuje MOUNT server.**

Mount server

MOUNT server řeší další činnosti, které také nemůže zajišťovat NFS server. Vede evidenci zpřístupňovaných (exportovaných) adresářových stromů, počáteční "přimontování" adresářového stromu včetně ověření identity uživatele a jeho přístupových práv. Je typicky řešen na aplikační úrovní a nezáleží na jeho rychlosti.

Implementace NFS

Při implementaci protokolu byl poprvé významněji použit koncept tzv. vzdáleného volání procedur prostřednictvím protokolu **RPC** (programátor nemusí řešit asynchronní komunikaci, ale pouze volá předem připravené procedury knihovního charakteru). Dále je využit také protokol **XDR** (definuje jednotný způsob reprezentace přenášených dat, nezávislý na konkrétní architektuře příjemce a odesilatele).

Elektronická pošta

Je to sluzba. Existují různé "koncepce" elektronické pošty (smtp, MS mail, X.400 - liší se formátem zpráv, adresami, přenosovými mechanismy, ...) a obecně jsou vzájemně nekompatibilní. Pro možnost vzájemné spolupráce vyžadují existenci poštovních bran. V internetu se pouziva **SMTP**. Krome protokolu pro prenos zprav mezi servery a klientem (smtp), je treba dodrzovat jeste spravny format (RFC 822 - hlavicka, telo, pripadne prilohy), zajistit zpusob stahovani zprav ze schranky na serveru (imap, pop3), rozsireni pro napr. narodni abecedy, urceni typu prilohy (MIME).

Architektura SMTP pošty

Vychází z modelu klient/server. **Poštovní server (mail server)** - zajišťuje transport zpráv a shromažďuje zprávy pro ty účastníky, kteří nejsou momentálně dostupní. **Poštovní klient** - umožňuje číst, psát a jinak zpracovávat jednotlivé zprávy a vytváří uživatelské rozhraní.

Umístění poštovní schránky

- **Na poštovním serveru** - poštovní klient musí mít ke schránce trvalý a dostatečně rychlý přístup. Výhoda: uživatel si může "sednout" ke kterémukoli počítači v síti, a mít plnohodnotný přístup ke své poště. Dříve to bylo praktické pouze v prostředí lokální sítě (sítě LAN), skrze sdílení souborů. Dnes lze používat i na dálku, skrze protokol **IMAP**.
- **Je rozdělena** - již přijaté, ale dosud nevyzvednuté zprávy se nachází na poštovním serveru již vyzvednuté zprávy se uchovávají u uživatele, na jeho počítači.
- **Schránka i klient jsou na serveru** -prostřednictvím WWW stránek (webmail).

Zpráva elektronické pošty

Původní pošta v rámci TCP/IP definovala pouze syntaxi a sémantiku hlavičky (**RFC 822**). Tělo brala jako “černou skříňku” a přílohy neuvažovala vůbec. Dnes existuje rozšíření (standard MIME), které zčásti specifikuje i formát těla zprávy a zavádí možnost použití příloh. Každa zpráva má (může mít):

- **Hlavicka (header)** – obsahuje adresu příjemce (tvaru něco@domena.com). Existují 3 kategorie: **To** ("hlavní příjemce", obdoba adresáta úředního dopisu), **Cc** (obdoba dopisu, který má jiného hlavního adresáta (v kategorii **To**)) a **Bcc** (příjemce **Bcc** kopie se **dozví** o příjemcích v kategorii **To** a **Cc**, ale příjemci v kategoriích **To** a **Cc** se **nedozví** o příjemcích v kategorii **Bcc**). Dále obsahuje adresu odesilatele, datum vzniku/odeslání, předmět zprávy (subject). Každá položka začíná na nové řádce a je uvozena klíčovým slovem (zakončeným dvojtečkou), za kterým následuje vlastní obsah položky. Pořadí položek v hlavičce není předepsáno.
- **Telo** - vlastní text spravy. RFC 822 předpokládal ASCII text.
- **Priloha** - cokoli, co lze "zabalit" do podoby souboru.

Prenos zprávy (zhruba):

1. Sestavení zprávy, příkaz k odeslání
2. Upload (odeslání) zprávy na poštovní server (pomocí SMTP)
3. Přenos zprávy mezi poštovními servery (pomocí protokolu SMTP, zpráva končí v poštovní schránce (mailboxu) příjemce).
4. Stažení zprávy z poštovní schránky do poštovního klienta (pomocí protokolu POP3).
5. Ctení přijaté zprávy, v rámci poštovního klienta příjemce.

Adresy v elektronické poště

Dříve byly poštovní adresy typu schránka@počítač, tj. byly pevně vázány na konkrétní počítač, a docházelo k velkým problémům při „stěhování“ uživatelů, při změně poštovního serveru apod. Dnes se používají spíše adresy typu **uživatel@doména**, kde „uživatel“ je jakýkoli alias (v podstatě jakýkoli text) a „doména“ je jméno DNS domény.

Doručování elektronické pošty

Odesílající poštovní server si z DNS MX záznamu zjistí, kterému (cílovému) poštovnímu serveru má zprávu odeslat. Na cílovém poštovním serveru se pošta rozděluje podle zde umístěného **seznamu aliasů** do správných schránek příjemců. Při použití doménového koše jdou na cílovém poštovním serveru všechny zprávy pro příjemce v doméně …@doména do stejné schránky a tudíž není třeba žádný seznam aliasů. Uživatel si může sám vytvářet další pravidla třídění pošty (obdobu seznamu aliasů), a tím vytvářet neomezený počet vlastních emailových adres. Doménový koš a schránky jednotlivých uživatelů se nemusí nacházet na stejném uzlu (poštovním serveru).

Přenos zprávy prostřednictvím SMTP má v zásadě on-line charakter. Odesílající uzel komunikuje přímo s koncovým příjemcem pošty a očekává, že tento je ke komunikaci připraven, proto musí být příjemce trvale dosažitelný. Pokud není, tak je pošta přesměrována na vhodný **mail spool** - řeší se pomocí MX záznamů. MX záznamy mají priority - **nižší číslo = vyšší priorita**. prostřednictvím priorit se stanoví pořadí poštovních serverů, tak aby vždy byl některý z nich dosažitelný on-line.

SMTP dialog

Odesilatel naváže spojení s určeným příjemcem na port 25 (kde čeká SMTP server). Příjemce je určen dle MX záznamů v DNS. Pokud se nedaří jej určit z DNS, snaží se odesilatel interpretovat část adresy za zavináčem jako jméno konkrétního počítače. Poté dochází ke vzájemnému dialogu. Obě strany si předávají důležité "identifikační" údaje “nápisy na obálce“ a teprve pak je přenesena vlastní zpráva. Příkazy SMTP mají textový charakter a odpovědi jsou zásadně číselné (obdobně jako v případě protokolu FTP).

Rozumné omezení (opatření proti spammingu) SMTP serveru je aby přijímal zprávy z jeho síte od jeho uživatelů a určené jen jeho síti pro jeho uživatele.

Problémy s netextovými přenosy

Původně byla SMTP pošta byla určena jen pro přenos krátkých textových zpráv v "čistém ASCII", tj. bez háčků, čárek, bez formátování a bez různých druhů písma. Přenosové mechanismy (protokol SMTP) jsou koncipovány tak, aby garantovaly přenos textových zpráv složených ze 7-bitových znaků. Pokud se někdo pokusí přenést něco jiného než 7-bitové znaky, není garantováno jak to dopadne.

Problém je s hlavně přílohami: pokud by k textové zprávě byl přiložen datový soubor, nemusel by "projít", protože datový soubor je obecně tvořený 8-bitovými byty. Problém je ale také i s národními abecedami, protože ty je nutné kódovat do 8 bitů a taky s formátováním, protože formátovací znaky jsou také 8-bitové.

MIME (Multipurpose **I**nternet **M**ultimedia **E**xtensions**)**

Řeší problém příloh i otázku použití národních abeced a formátování zpráv. Protokol je podporován většinou novějších poštovních klientů. Definuje **kódování** (2 způsoby převedení 8-bitových dat do 7-bitové podoby), **typování dat** (aby bylo možné definovat co jsou data zač a bylo možné odvodit, jak mají být zpracována) a **rozšíření formátu zprávy** (zavádí nové položky do hlavičky a umožňuje aby tělo zprávy mělo více složek).

Kódování v MIME

- **Quoted-printable** - tisknutelné ASCII znaky ponechává tak jak jsou. ostatní kóduje do trojice znaků: rovnítko a hexadecimální kód znaku v použité znakové sadě (např. =C8). Je to vhodné tam, kde je málo netisknutelných znaků.
- **Base64** - kóduje všechny znaky. Vezme binární kódy všech znaků, seřadí je do posloupnosti, rozdělí je na šestice a tím získá čísla od 0 do 63 a nakonec čísla použije jako indexy do převodní tabulky A,B,C..Z,a,b…..,8,9,+,/.

IMAP vs POP3 - IMAP dovoluje pracovat s daty na serveru, je urcen pro stala pripojeni typu pevna linka/adsl. POP3 umi zpravy stahovat na lokal - vhodne pro modemisty.

Služby pro zpřístupnění informací (WWW, proxy, peer-to-peer sítě)

WWW

Původně to byla jen služba pro zpřístupnění textů. Dnes podporuje celé řady multimediálních formátů. Vychází z myšloenky hypertextu.

Hypertext je (původně lineární) text, uzpůsobený na přeskakování „z myšlenky na myšlenku“. Přeskakování v hypertextu je označováno jako (browsing, to browse). Informace jsou členěny na (relativně malé) jednotky, zvané **stránky** (pages). V rámci stránek jsou informace uspořádány (víceméně) lineárně. Stránky se mohou nalézat na různých (libovolných) místech - je to plně distribuované. Mezi stránkami mohou existovat libovolné **vazby**.

Architektura WWW

Vychází z architektury klient/server. WWW server uchovává jednotlivé WWW stránky (dokumenty napsané v jazyku HTML), na (explicitní) žádost je poskytuje svým klientům. WWW prohlížeč si „vyzvedává“ stránky od serverů (přenos probíhá přes protokol HTTP), zobrazuje je podle svých grafických možností uživateli.

Dokumenty HTML mohou být:

- **Staticke** - na servery v nemennej podobe sa aj přenášejí. Může být indexován fulltextovými vyhledávači.
- **Dynamicke** - je generovany na strane serveru. Neexistuje sám o sobě a tedy nemůže být indexován a nemá smysl jej cache-ovat.
- **Aktivne stranky** - dotvaraju sa na strane klienta – applety, ActiveX. Existuje staticky, ale některé jeho části vznikají dynamicky, účinkováním výkonných částí které dotváří výslednou podobu.

HTTP

Přenáší data v textovém tvaru. Používá transportní služby protokolu TCP. Server přijímá požadavky na dobře známém portu 80. Funguje bezestavově. Navazuje samostatné spojení pro každý objekt v rámci WWW stránky (HTTP 1.0).

Bezestavovy protokol (request/response) – prinasa problemy v situacích, kde si je nutne si pamatat historiu predchadzajucej komunikacie (cookies). Otazky v podstate **Get** (url), **Head** (url), **Post** (data z formulre). Odpovedi maji ciselny charaker o stavu (404- page not found), nasledovany daty z prislusneho url v pripade uspechu (200-ok).

- **HTTP 0.9** - velmi jednoduchý přenosový protokol. Neuměl přenášet nic jiného, než jen hypertextové dokumenty.
- **HTTP 1.0** - více možností "vzájemné domluvy" mezi klientem a serverem pomocí hlaviček. Lze přenášet i jiné typy objektů, než jen WWW stránky.
- **HTTP 1.1** – Zavádí virtuální WWW servery, persistentní transportní spojení (jedno TCP spojení může být využito pro přenos více různých objektů), pipelining (klient může posílat více požadavků za sebou a teprve pak dostávat odpovědi), lepší podporu pro cache a proxy paměti, lepší zabezpečení.

Virtuální WWW servery - někdy je zapotřebí, aby na jednom počítači "běželo" více samostatných WWW serverů. **IP-based** - každý virtuální WWW server má svou vlastní IP adresu. **Name-based** - všechny virtuální WWW servery (na stejném fyzickém počítači) mají stejnou IP adresu a požadavek klienta musí obsahovat hlavičku HOST. Bez té není možné rozlišit name-based virtuální WWW server.

Cookies - bezestavový charakter komunikace WWW klienta a serveru přináší některé problémy. Neumožňuje pamatovat si historii předchozí komunikace. Možné řešení: stavová informace se ukládá do URL nebo použití cookies. Cookie je malý datový údaj, který generuje server, a uchovává jej klient. Server uloží do cookie vše co si potřebuje pamatovat o aktuální transakci, a pošle cookie klientovi (pomocí hlavičky **Set-Cookie**). Při příštím požadavku klienta na stejný server je k požadavku přidáno příslušné cookie (pomocí hlavičky **Cookie**). Server si na základě přijatého cookie "připomene" předchozí historii.

Proxy

Proxy server je jakysi prostrednik mezi klientem a serverem, ktery vyrizuje pozadavek. Klient posle pozadavek proxy serveru (o cemz uzivatel ani nemusi vedet a typicky si nemuze vybrat, zda proxy server pouzije ci nikoli) a ten jej vyridi bud ze sve cache nebo dotazem na dany server a pak preposlanim odpovedi.

Pouziva se z duvodu bezpecnostnich (jediny bod site pristupny z internetu, vsechny lokalni pocitace jsou az "za nim"), vykonovym (cache-ovanim muze usetrit spoustu prenosove kapacity), pripadne jinym (napr. povoleni jen urcitych stranek - pokus o pristup na jine proxy server nedovoli).

Moznosti cache-ování

- **V rámci klienta** - klient sám si uchovává objekty ve své cache paměti. Je to velmi efektivní (jsou nejrychleji k dispozici) ale je to cache jen pro něj, nikoli pro jiné klienty/uživatele.
- **Proxy caching** - cache paměť funguje na uzlu uprostřed celého řetězce. Typicky na proxy serveru. Může to být společné pro více uživatelů/klientů.
- **V rámci serveru** - například pokud generuje stránky dynamicky, jejich uchováváním v cache šetří svůj výkon a zrychluje odpovědi.

Peer to peer site (Peer2Peer, P2P)

Všechny uzly si jsou rovny. Obecně tedy všechny uzly mohou potenciálně vystupovat jako servery (a něco poskytovat), i jako klienti (něco požadovat).
Funkčností jde nejčastěji o vymenne datove site (bittorrent, Gnutella, Napster, DirectConnect...), kde si data predavaji dva (vice) koncovych uzivatelu, nejsou stahovana ze zadneho serveru.

Gnutella - neexistuje žádný centrální server (centrální index všech dostupných souborů). Klient, který hledá nějaký soubor, se dotáže svého souseda (max. 7 svých sousedů). Soused který nezná odpověď se sám dotáže max. 7 svých sousedů. Maximální "hloubka" dotazování je 10. Po nalezeni se stahuje přímo od vlastnika.

Bezpečnost síťového přístupu

Bezny sitovy provoz je tovren v "dobre vire". Napr. TCP/IP od pocatku neresil zadnou bezpečnost – bola by to aj v niektorych pripadoch zbytocna/neziaduca rezia. Kdokoli je na trase, tak muze do zprav nakukovat, i je menit. To samozrejme neni pro nektere typy komunikaci dostatecne. Aby byla sitova komunikace bezpecna, mela by zajistovat tyto veci:

- **Identifikace a autentikace** - zjisteni identity ucastniku - ten druhy je tim, za koho se vydava.
- **Autentizace** - overeni pristupovych prav k urcitemu zdroju, ci k provedeni urcite aktivity - nelez kam nemas.
- **Duvernost** - aby data byla přístupná pouze určitému okruhu subjektů (například pouze příjemci) a nikomu jinému. sifrovani komunikace - nikdo si to neprecte.
- **Integritu** - možnost spolehlivě zjistit, zda od vytvoření dat jejich původním autorem došlo či nedošlo k nějaké jejich změně. Taky sifrovani - nikdo to nezmeni.
- **Neodmítnutelnost** (non-repudiation) - pro věrohodnost nejrůznějších úkonů realizovaných prostřednictvím Internetu je nezbytné mít možnost prokázat, že jejich autor (původce) je skutečně vykonal, a nemohl příslušný úkon popřít (odmítnout jeho "autorství").
- **Dostupnost** (availability) - je požadavek na to, aby služby i zdroje, které uživatelé potřebují, jim byly skutečně k dispozici, vždy když to oni potřebují.
- **Žurnálování** (auditing): pro řešení nejrůznějších bezpečnostních incidentů je velmi žádoucí, aby po každé činnosti zůstávala určitá "stopa", umožňující kdykoli později detekovat, kdo a co dělal.

Zabezpečené protokoly

Protokoly pro bezpecny prenos informaci - z datoveho toku neni poznat, co je prenaseno. Resi se sifrovanim. Pro "uplne" (kazda sifra muze byt prolomena) bezpecne pouziti je nutne identifikatory ucastniku (fingerprinty, jmena/hesla) prenest bezpecnou cestou (rucne lokalne opsat, nikoli stahnout nezabezpecene z webu). Bezpecnost zavisi na bezpecnosti pouzite sifry a korektni implementaci algoritmu na obou stranach.

Identita ucastniku a navazani spojeni se resi pomoci asymetrickeho sifrovani, vlastni vymena pak pomoci symetrickeho. Identita je dana public klicem ucastnika (nebo jeho fingerprintem). To se typicky deje jen pro server a uzivatel svou identitu serveru "prokazuje" pozdeji pomoci jmena a hesla.

IP Security (IPSec)

- Je to celý rámec (framework). Nejde o (jeden) konkrétní protokol, ale o soustavu vzájemně provázaných opatření a dílčích protokolů pro zabezpecení komunikace pomocí IP protokolu, funguje na sítové vrstve a teda není závislý na protokolech vyšších vrstev jako je TCP a UDP (napr. SSL protokol pracuje na transportní vrstve)
- **funguje na síťové úrovni!!!**
- původně vznikl pro IPv6 (ale začal se používat i pro stávající IPv4)
- Zajistuje (**důvěrnost** – sifruje prenasena data a **integritu** - přenášená data nejsou při přenosu změněna)
- muže být implementováno na bráne (security gateway, lokální sít je považována za bezpecnou) nebo na koncových zarízení
- Umožňuje aby si komunikující strany dohodly algoritmy a klíče pro zabezpečení svých přenosů.
- Má dva režimy fungování:
    - **transport mode** - "zabezpečovací údaje" se vloží přímo do IP datagramu - do jeho hlavičky a za ni. IP hlavicka je nechránená (jeden z duvodu je užívání systému NAT), telo paketu je šifrováno (data vyšších protokolu). Použitelné jen na koncových stanicích.
    - **tunnel mode** - pakety jsou celé (vcetne hlavicky) zašifrovány, vloženy do dalšího paketu a na druhé strane rozbaleny. Povinné pro security gateways, volitelné pro koncové stanice. Ve vnejší IP hlavicce se jako príjemce uvádí security gateway na hranici cílové síte.

IPsec protokoly

AH (Authentication Header)

- komunikující strany se dohodnou na klíci
- k datum se pripojuje hash
- chrání také pred replay attack
- provádí autentizaci a kontrolu zmeny dat, neprovádí šifrování

ESP (Encapsulating Security Payload)

- provádí autentizaci a také šifruje obsah
- pro šifrování používá 3DES, Blowfish aj. (puvodne DES, již není považováno za bezpecné)

Dohoda na klícech

Pred použitím protokolu AH ci ESP si musí strany dohodnout klíce:

- manuální konfigurace
- automatická konfigurace – IKE (Internet Key Exchange) protokol

SSL (secure socket layer)

Protokol v rodine tcp/ip (vyvinutý firmou Netscape) slouzici k bezpecnemu provozu jinych protokolu. Výhodou tohoto přístupu je i skutečnost, že vše se dá zařídit tak, aby se používané aplikace nemusely nějak výrazněji měnit. Zabudované zabezpečovací mechanismy totiž mohou vytvářet jednotlivým aplikacím iluzi toho, že pracují nad takovou přenosovou infrastrukturou, která je již sama zabezpečená a sama se stará o bezpečnost přenášených dat. Nevýhodou je pak skutečnost, že takováto "společná" míra bezpečnosti nemusí všem aplikacím postačovat - ze stejných důvodů, o kterých jsme se již zmiňovali výše.
Je zalozen na sifrovani a muze pouzit nekolik ruznych algoritmu. Pouziti se sestava ze tri kroku: 1. Dohodnuti se na algoritmech, 2. autentikace ucastniku pomoci verejnych klicu, 3. vymena dat pomoci symetrickych sifer.

HTTPS - secure obdoba http protokolu.

SSH (secure shell) - pro bezpecne terminalove spojeni se se vzdalenym serverem.

SFTP (ssh FTP) - pro bezpecny prenos souboru pomoci protokolu FTP.

Překlad adres

NAT (Network Address Translation)

Používá se na rozhraní mezi privátní sítí a veřejným Internetem. **Překládá** lokální (privátní, vícenásobně použitelné) adresy na veřejné (unikátní) adresy. **Poskytuje zabezpečení** - lokální adresy "nejsou vidět" z veřejné sítě. **Setří IP adresy** - pokud jen část lokálních uzlů potřebuje komunikovat s vnějším světem!!!

NAT je funkce sítového routeru pro zmenu IP adres packetu procházejících zarízením, kdy se zdrojová nebo cílová IP adresa prevádí mezi ruznými rozsahy. Nejbežnejší formou je tzv. maškaráda (maskování), kdy router IP adresy z nejakého rozsahu mení na svoji IP adresu a naopak – tím umožnuje, aby pocítace ve vnitrní síti (LAN) vystupovaly v Internetu pod jedinou IP adresou. Router si drží po celou dobu spojení v pameti tabulku prekladu adres.

Preklad sítových adres je funkce, která umožnuje prekládání adres. Což znamená, že adresy z lokální síte preloží na jedinecnou adresu, která slouží pro vstup do jiné síte (napr. Internetu), adresu prekládanou si uloží do tabulky pod náhodným portem, pri odpovedi si v tabulce vyhledá port a pošle pakety na IP adresu prirazenou k danému portu. NAT je vlastne jednoduchým proxy serverem (na sietovej vrstve).

Implementace

Klient odešle požadavek na komunikaci, smerovac se podívá do tabulky a zjistí, zdali se jedná o adresu lokální, nebo adresu venkovní. V prípade venkovní adresy si do tabulky uloží císlo náhodného portu, pod kterým bude vysílat a k nemu si priradí IP adresu. Behem preposílání „ven“ a zmeny adresy v paketu musí NAT také prepocítat CRC checksum TCP i IP (aby pakety nebyly zahazovány kvuli špatnému CRC, protože zmenená adresa je jejich soucástí). Výhodami NAT su umožnenie pripojenie viacerých pocítacov do internetu cez jednu zdielanú verejnú IP adresu, a zvýšenie

bezpecnosti pocítacov za NATom (aj ked je to security through obscurity a nie je dobré postavit bezpecnost iba na NATe). Nevýhodami potom sú nefungujúce protokoly (napr. aktívne FTP) – co je zrejmé z fungovania NATu.

Pokud: vztah mezi "vnitřními" a "vnějšími" IP adresami je 1:1

Statický NAT - vztah mezi "vnitřními" a "vnějšími" IP adresami je pevně dán. Vnitřní uzly mohou být přístupné z vnější sítě. Lze využít například při "stěhování" mezi ISP (bez nutnosti měnit IP adresy dle CIDR).

Dynamický NAT - "vnitřním" IP adresám jsou přiřazovány "vnější" IP adresy dynamicky, podle potřeby. Vnitřní uzly nejsou obecně dostupné z vnějšku. Lze ušetřit "vnější" IP adresy (pokud některé vnitřní uzly nepotřebují).

Pokud: vztah mezi "vnitřními" a "vnějšími" IP adresami je 1:N:

PAT (Port Address Translation alias. Overloading)

Všechny "vnitřní" IP adresy se mapují do 1 vnější IP adresy. "Rozlišující" informace se uchovávají v číslech portů. Dokáže využít 1 veřejnou adresu pro "hodně" uzlů v privátní síti. Nelze navazovat spojení směrem dovnitř. Pro některé aplikace/služby nemusí NAT fungovat vůbec: Pro takové, které přenáší IP adresy i jinde než v hlavičce (kde o tom NAT neví a nemůže adresy měnit - např. IPSEC). **Inteligentní NAT** se snaží rozpoznat konkrétní protokoly, které skrz něj prochází, a mění IP adresy i v těle IP paketů.

Krome techto "vlastnosti", ktere se projevi pri kazdem pouziti NATu, lze tuto techniku pouzit i pro dosazeni jinych cilu:

LoadBalancing - nekolik fyzickych serveru za jednou ip adresou a "NATovadlo :)", ktere rozdeluje pozadavky.

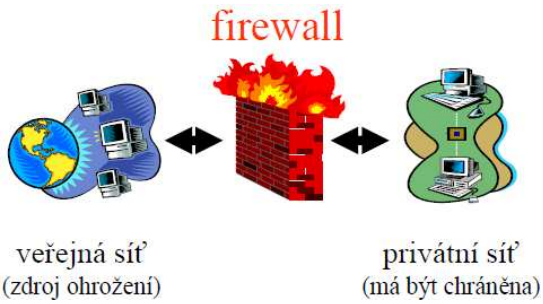
Failover - server a jeho backup za jednou ip adresou. Kdyz server selze, tak se zacne pouzivat backup, aniz by o tom kdokoli vedel.

Priklad - dnesni domaci routery

Firewally

Jde o obecné označení řešení, které sleduje zajištění bezpečnosti. firewall může být realizován různými způsoby, např. pouze organizačními opatřeními, jen v SW (vhodným nakonfigurováním smerovacu), kombinací SW a HW (nejcasteji).

Každý firewall obecně má **část zajišťující blokování** (zabraňující v přístupu), **část zajišťující prostupnost toho, co má být povoleno**.

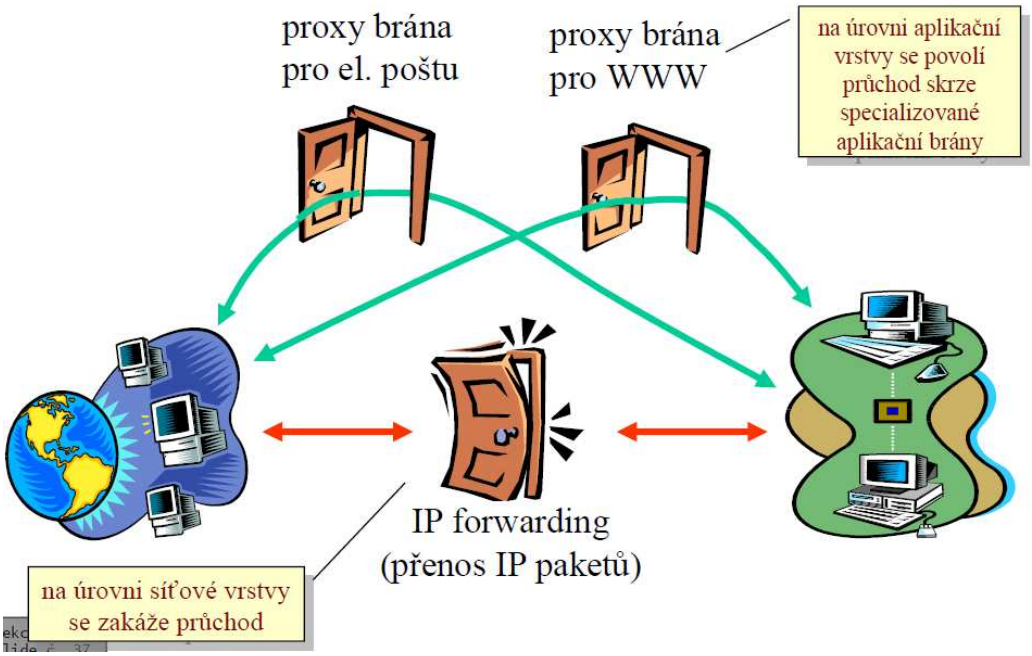


Možnosti blokování

Lze realizovat například **nakonfigurováním směrovačů**. Tak aby nepropouštěly určitý druh provozu, resp. nepropouštěly nic kromě explicitně povoleného provozu. Problémem je, ze ne všechny běžné směrovače dávají takové možnosti, aby bylo možné nakonfigurovat vše co je třeba.

Existují i specializované produkty, tzv. **paketové filtry** které mají výrazně posíleny nejrůznější „povolovací“ a „blokující“ možnosti, v závislosti na mnoha různých faktorech. Mohou to být řešeny na bázi vlastního HW a SW, nebo i řešení čistě SW (běžící na normálním počítači nebo i na směrovači). Paketove filtry jsou relativně laciné a snadné řešení. Fungují na úrovni síťové vrstvy, eventuelně i transportní (nefungují na úrovni aplikační vrstvy) musí se rozhodovat na základě síťových adres a čísel portů. Jejich možnosti rozpoznání útoku jsou omezené, kvůli tomu že "nevidí" až na aplikační úroveň. Mohou být typu "**statefull**" (každý paket posuzují s uvážením historie) nebo "**state-less**" (každý paket posuzují bez uvážení historie).

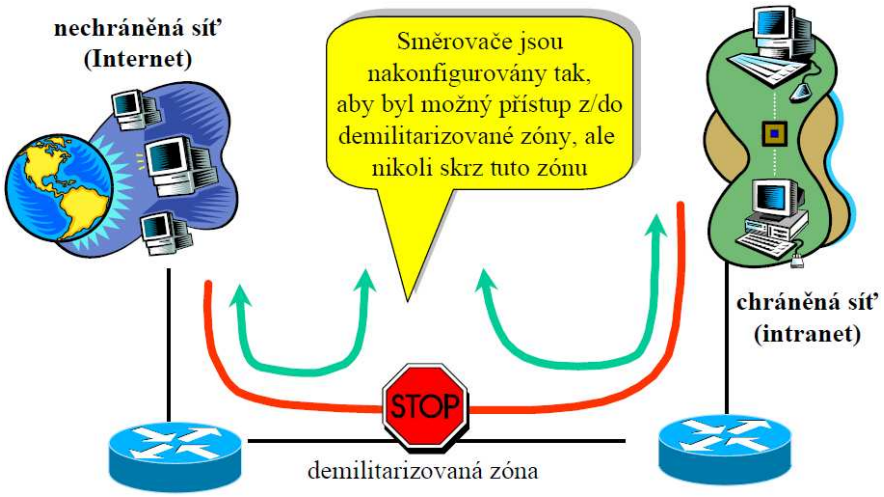
Princip řešení na aplikační úrovni


proxy brána
pro el. poštu
proxy brána
pro WWW
na úrovni aplikační
vrstvy se povolí
průchod skrze
specializované
aplikační brány
IP forwarding
(přenos IP paketů)
na úrovni síťové vrstvy
se zakáže průchod

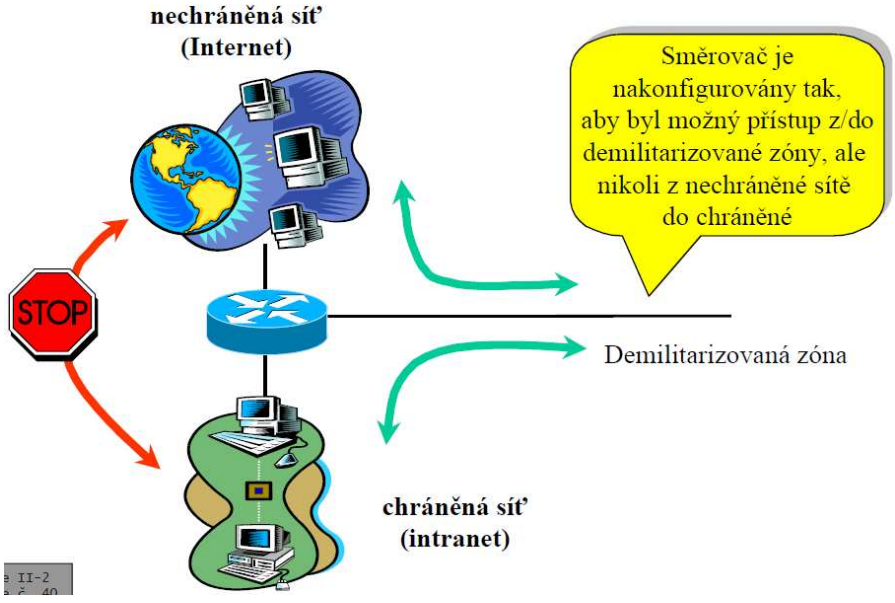
Vše může být implementováno čistě softwarovými prostředky. Zákaz IP forwardingu (dual-homed host nefunguje jako směrovač). Aplikační (proxy) brány běží na daném uzlu jako aplikace. Relativně laciné řešení ve formě "kompaktních" aplikací, např. WINROUTE.

Rešení s tzv. demilitarizovanou zónou

Se 2 směrovači



S 1 směrovačem



Idea: do DMZ se umístí „přestupní stanice“, přes které půjde veškerý provoz který má být povolen a přestupní stanice jej dokáží účinně kontrolovat. „Přestupní stanice“ budou ve skutečnosti aplikační brány specializované na určitý druh provozu, který dokáží dobře kontrolovat například poštu, WWW apod. Jsou to tzv. proxy brány. Jde o stejný princip, jaký byl využíván již u středověkých hradů: vodní příkop měl bránit

tomu, aby se do hradu dostal někdo jinou cestou než hlavní branou a u ni stál hlídač a ten každého zkontroloval.

Demilitarizovaná zóna je “vidět” z obou stran, ale není “průhledná skrz”. Z nechráněné sítě (Internetu) jsou “vidět” pouze uzly přímo v DMZ. Celá chráněná síť za DMZ (princip) je z vnějšku neviditelná. Může dokonce používat adresy, které by “venku” byly nepřípustné. Do DMZ se umístí takové servery které mají být „vidět“ (WWW, FTP server). Ostatní (chráněné) servery mohou být „schovány“ v chráněné síti (např. skutečný poštovní server).

Firewall je zařízení slouzici ke kontrole a případnému **povolení/zakázání** nějakého síťového provozu. V podstatě odděluje vnější síť (typicky Internet) od vnitřní sítě (LAN nebo i jeden počítač). Lze řešit v HW i SW. Ve firemní síti firewall třeba zakazuje veškerý HTTP provoz všemu kromě proxy serveru, a zabraňuje navazování spojení zvenku. Dále se filtruje třeba SMTP, nebo některý druh přístupu pouze z důvěryhodných sítí,… Stavový firewall (stateful) sleduje vztahy mezi pakety a ohlíží se na historii.

Umoznuje sledování a filtrování komunikace na síti (paketové filtry – napr. na routeru):

- blokování – zabranuje neoprávnenému prístupu
- prostupnost – propouštení povoleného toku

Firewall muze existovat (byt nasazen) na ruzných vrstvách:

- **sítová** – pouze dle zdrojových a cílových adres a protokolu
- **transportní** – také podle portu
- **aplikacní** – dle obsahu (dat)

Demilitarizovaná zóna (DMZ): Je jiné rešení bezpecnosti. Prístup ven pouze pres specializovaná zarízení (proxy, brány), nelze prímo. To platí pro oba smery.

Na jednom PC je firewall vacsinou software který kontroluje príchozí a odchozí spojení na úrovni aplikací - můžu povolit spojení ven prohlížeči, ale už ho třeba zakázat malwaru.

Certifikáty

Asymetrické šifrování (asymmetric cryptography)

Při této technice šifrování se pracuje se dvěma nestejnými klíči, z nichž pouze jeden musí zůstat utajen (jako tzv. **privátní klíč**), zatímco druhý klíč není potřeba utajovat - právě naopak, kvůli snazšímu využití celého asymetrického šifrování, je vhodné jej zveřejnit tak, aby byl každému snadno dostupný (proto je také tento klíč označován jako "**veřejný**"). S distribucí takovéhoto veřejného klíče pak nejsou žádné problémy, je možné jej dokonce "rozvěsit na každém rohu". odstatné přitom je, že to co je "zamknuto" (zašifrováno) jedním z klíčů, lze "odemknout" (odšifrovat) právě a pouze druhým z obou klíčů. Této důležité vlastnosti asymetrického šifrování lze využít dvěma odlišnými způsoby:

- **Použitím privátního klíče k zašifrování a veřejného klíče k odšifrování**. Jelikož privátní klíč má k dispozici jen jeho oprávněný vlastník, může toto udělat pouze on. Naproti tomu veřejný klíč má k dispozici v zásadě každý, takže odšifrování může realizovat také každý - výsledkem je splnění požadavků na identifikaci a autorizaci (protože nikdo jiný než skutečný autor nemohl data zašifrovat privátním klíčem), dále požadavku na integritu (nikdo jiný nemohl data odšifrovat, změnit a zase zašifrovat privátním klíčem), i na neodmítnutelnost (data nemohl zašifrovat nikdo jiný než vlastní privátního klíče). Za zdůraznění jistě stojí, že tímto způsobem není naplněn požadavek na důvěrnost - vzhledem k veřejnému charakteru klíče se k datům může dostat skutečně kdokoli.

- **Použitím veřejného klíče k zašifrování, a privátního klíče k odšifrování**. Jelikož k veřejnému klíči má přístup kdokoli, může zašifrování provést také kdokoli. Ovšem odšifrovat takto zašifrovaná data může pouze vlastník privátního klíče a nikdo jiný. To v praxi znamená naplnění požadavku na důvěrnost (nikdo jiný než vlastník privátního klíče je nemůže "odemknout", tj. odšifrovat). Naproti tomu zde není splněn ani požadavek na autentikaci a neodmítnutelnost (zašifrovat mohl kdokoli). Požadavek na integritu v zásadě také není splněn, protože kdokoli mohl původní data zahodit (nemůže je odšifrovat), a nahradit je vlastními daty, korektně zašifrovanými veřejným klíčem.

Certifikaty

U asymetrického šifrování, které je používáno pro potřeby elektronických podpisů, existuje kritické místo, které ovlivňuje důvěryhodnost a celkovou spolehlivost celého tohoto mechanismu. Jde o to, že s veřejným klíčem musí být dostatečně spolehlivým způsobem spojena identita osoby, které tento klíč patří. To je totiž velmi podstatné - představme si příjemce, který elektronickou poštou dostane nějakou zprávu s elektronickým podpisem, a na této zprávě je napsáno že odesilatelem je pan Jan Novák. Příjemce použije veřejný klíč, který mohl získat skutečně kdekoli (např. dostat jej spolu se zprávou, nebo jej stáhnout odněkud z nějakého internetového serveru apod.), a prostřednictvím tohoto veřejného klíče ověří, že vše je v pořádku (že nový otisk zprávy souhlasí se zašifrovaným otiskem atd.). Z čeho pak ale příjemce může odvodit závěr, že autorem zprávy je skutečně Jan Novák?

Odpověď je jednoduchá: z toho, **jaká identita je přiřazena k veřejnému klíči**, který příjemce použil pro ověření podpisu na přijaté zprávě. Pokud by s použitým veřejným klíčem byla spojena identita pana Josefa Novotného, měl by příjemce právo si odvodit, že skutečným autorem je Josef Novotný. Způsob, jakým je s veřejným klíčem spojena identita určité konkrétní osoby, je tedy nesmírně důležitý - jakákoli možnost podvodu či zneužití tohoto vztahu mezi veřejným klíčem a identitou vlastníka by mohla mít opravdu nedozírné následky.

Konkrétní způsob spojení veřejného klíče s identitou určité osoby je označován jako certifikát. Lze si jej neformálně představit jako visačku (k veřejnému klíči), na které je napsáno komu tento klíč patří. Velmi pečlivě přitom musí být "pohlídáno", jakým způsobem je nápis na této visačce vytvořen (aby odpovídal skutečnosti) a aby nebylo možné jej změnit. Zvláště pak s ohledem na to, že právě certifikáty (a nikoli samotné veřejné klíče jako takové) mohou a mají být šířeny úplně volně - doslova mohou být rozvěšovány na Internetu na každém jeho rohu. Jak potom zajistit, aby jejich obsah korektní a bylo možné se na něj spolehnout? Možností, jak zajistit tolik potřebnou důvěryhodnost certifikátů, existuje více - v dnešní době se lze v praxi setkat se dvěma značně odlišnými přístupy:

První z nich je založen na předpokladu, že certifikát si vystaví sám jeho vlastník, a následně vyzve osoby které jej znají aby stvrdili jeho platnost (aby jeho certifikát sami podepsali svým elektronickým podpisem). Tím tyto další osoby v zásadě přenáší svou vlastní důvěryhodnost na další osobu, jako kdyby říkali: "věříte-li mému certifikátu, větře i certifikátu tamté osoby, já ji znám". Tímto způsobem pak každý certifikát může kumulativním způsobem získávat vyšší a vyšší důvěryhodnost, úměrně tomu kolik dalších lidí příslušný certifikát podpoří svým vyjádřením důvěryhodnosti (tím že jej podepíše). Podíváme-li se na to z obecnějšího pohledu, vzniká tímto způsobem celé husté předivo vzájemného předávání důvěry, které není nadarmo přirovnáváno k pavučině. Vedle "kladných" odkazů zvyšujících důvěryhodnost přitom v takovéto pavučině nutně musí existovat i odkazy negativní, ve smyslu "nedůvěřujte tomuto certifikátu, něco s ním není v pořádku".

Celý systém takovéto "pavučiny důvěry" je poměrně snadno zranitelný, a pro skutečně spolehlivé komunikace, kde je třeba "sázet na jistotu", toto řešení není vhodné. V praxi je toto řešení používáno v rámci systému PGP (Pretty Good Privacy), kde naopak docela dobře vyhovuje - jde o řešení, které má za cíl zvýšit důvěryhodnost jinak málo důvěryhodné elektronické pošty, navíc způsobem který nevyžaduje žádné významnější náklady na svou realizaci, lze jej vybudovat v zásadě "svépomocí" a nevyžaduje žádnou explicitní součinnost či podporu např. ze strany zákona či orgánů státu atd. (ostatně, PGP je řešení, které nasadila a uvedla do praxe sama komunita uživatelů Internetu pro potřeby své vzájemné komunikace). Pro aktivity, které vyžadují vyšší úroveň důvěryhodnosti a zabezpečení - například pro obchodování, uzavírání smluvních vztahů apod. toto řešení nepostačuje.

Alternativou, která je schopna poskytnout podstatně vyšší míru důvěryhodnosti - byť za cenu opravdu nemalých nákladů, součinnosti se zákony atd. - je varianta "stromu důvěry". Na rozdíl od "pavučiny důvěry", charakteristické pro PGP, zde pro stvrzení důvěryhodnosti postačuje jediná (dostatečně důvěryhodná) třetí strana, která řekne něco jako "ručím za to, že tento certifikát je v pořádku". Takováto třetí strana přitom nemusí být žádným známým či přítelem toho, komu certifikát patří, a své "potvrzení pravosti certifikátu" nevydává z lásky k bližnímu, ale jako běžnou (typicky placenou) komerční službu. Jde o tzv. certifikační autoritu (CA, též: poskytovatele certifikačních

služeb), která certifikát dotyčné osobě nejen vydá (vystaví), ale především se za něj postaví, resp. zaručí se za obsah certifikátu (neboli za to, že veřejný klíč, uvedený v příslušném certifikátu, skutečně patří osobě která je v certifikátu uvedena jako její vlastník). Fakticky takovéto "zaručení se" má opět podobu toho, že certifikační autorita sama podepíše vystavený certifikát (svým vlastním elektronickým podpisem). O pravosti takovéhoto certifikátu (včetně jeho neporušenosti, resp. integrity atd.) se může přesvědčit každý - stačí k tomu aby měl k dispozici veřejný klíč této autority coby třetí důvěryhodné strany. Kde jej ale vezme?

Odpověď již zahrnuje "strom" místo "pavučiny" - certifikačních autorit může existovat více, a mohou si mezi sebou poskytovat důvěru, tím že jedna certifikační autorita vyjádří důvěru jiné autoritě, což realizuje podepsáním jejího certifikátu. Tím vlastně daná certifikační autorita zařadí tu, které důvěru vyjádřila, do stromovité struktury pod sebe. Z pohledu uživatelů to pak má tu výhodu, že jim stačí znát veřejný klíč k nejvyššímu bodu stromu (kořeni), a pak si mohou ověřit pravost certifikátu vydaného certifikační autoritou kdekoli v podstromu.

Činnost certifikačních autorit je přitom podepřena zákonem, který na tyto subjekty klade přísné požadavky a také jejich dodržování kontroluje. Je to ale nezbytné k tomu, aby zákon mohl udělovat elektronickým podpisům, vytvořeným pomocí takto vydaných certifikátů (a při splnění dalších podmínek, jako např. použití bezpečných prostředků pro vytváření podpisů, použití certifikátů definovaných zákonem atd.) stejný statut jaký mají podpisy vlastnoruční.

Digitální certifikát

Obecné označení pro údaje, týkající se určitého subjektu a stvrzené jiným subjektem, který se zaručuje za jejich pravost (tzv. certifikační autoritou - u nas 1.CA, ve svete VeriSign). Nejčastěji je v certifikátu obsažen veřejný klíč vlastníka certifikátu, který má být veřejně přístupný (ale mohou zde být obsaženy i další údaje). Údaje v certifikátu jsou chráněny pomocí asymetrických šifrovacích technik - jsou zašifrovány privátním klíčem vydavatele certifikátu (certifikační autority), a mohou být dešifrovány s použitím veřejného klíče certifikační autority (který je veřejně známý). Význam a věrohodnost certifikátu jsou závislé jak na věrohodnosti samotné certifikační autority, tak i na způsobu, jakým tato autorita získává a ověřuje údaje, které svým certifikátem stvrzuje.

Treba pro certifikat pouzitelny pro digitalni podpis pro komunikaci se statni spravou musite jit osobne s obcankou do 1.CA. Nikdo jiny zatim nema "licenci" na to, aby tyto certifikaty vydaval. Naproti tomu Verisign vydava vice "druhu" certifikatu, podle toho jak byla overena identita.

Pro uplnou funkci je treba pouzivat i casove znamky (timestamp). Jinak bych u cehokoli mohl tvrdit, ze to bylo podepsano nekym jinym mym certifikatem az potom, co jsem jej prohlasil za neplatny a vygeneroval si jiny.

VPN (Virtuální privátní sít)

Z fyzickeho pohledu jde o součást jiné sítě. Z logického pohledu se jedná o samostatnou sit. Muze mit vlastni adresovaní, z jinych siti pristup jen pres branu.

Duvod vzniku - uzivatel chce mit vlastni sit, ale nevyplati se mu ji fyzicky budovat. Jeji "simulace" je levnejsi. Provoz ve VPN je sifrovan, aby byl bezpecny. Poskytuje typicky sluzby:

Identifikace a autentizace - u každého "vstupu" dochází ke zjištení a overení identity uživatele, ten se pak muže "volne pohybovat" po celé VPN.

Duvernost - komunikace v rámci VPN je chránena proti odposlechu i když fakticky vede pres sdílenou a verejnou infrastrukturu. Reší se pomocí **šifrování**.

Integritu - komunikace v rámci VPN je chránena i proti tomu, aby ji nekdo neoprávnene pozměnil.

Priklady pouziti - uzivatel muze z domu do firemni site. Nekolik pobocek dohromady se tvari jako jedna sit. Ve svete telekomunikaci se vyskytuji sluzby jako "volani zdarma v ramci firmy", jejiz telefony tvori VPN.

Programovací jazyky a prekladace

Struktura kompilátoru a navazujících nástroju (linkery, loadery, debuggery, knihovny, preprocesory)

Co je to překladač?

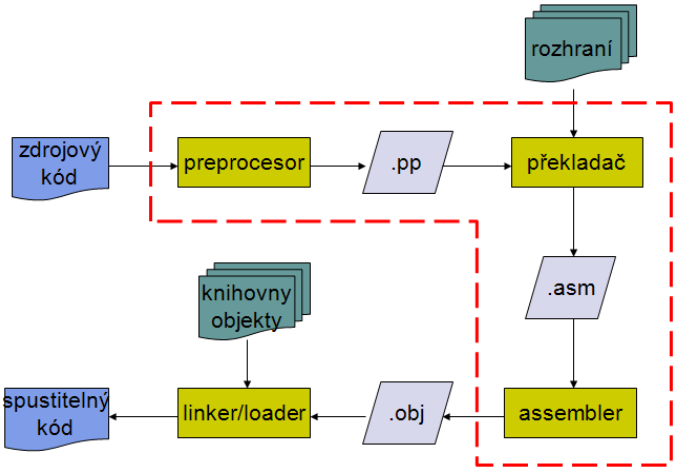
Mějme vstupní jazyk L_{in} generovaný gramatikou G_{in}. Dále mějme výstupní jazyk L_{out} generovaný gramatikou G_{out} nebo přijímaný automatem A_{out}.

Překladač je zobrazení $L_{in} \rightarrow L_{out}$, kde $\forall\, w_{in} \in L_{in}\ \exists\, w_{out} \in L_{out}$. Pro $w_{in} \notin L_{in}$ zobrazení neexistuje.

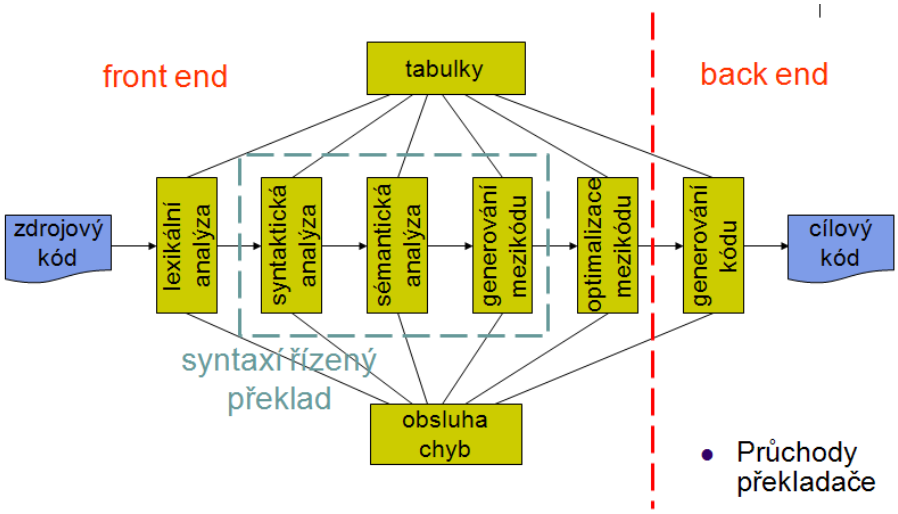
Teda provadi preklad zdrojového souboru do **relativniho (objektoveho) kodu** pocitace – vznika .OBJ soubor.

Relativni kod je téměř hotovy program. Slovo relativni znamena, ze adresy proměnných nebo funkci ještě nejsou znamy (napr. proto, ze jsou ulozeny v knihovne) a jsou tedy v .OBJ souboru zapsany relativne.

Preklad programu



Fáze překladače



Preprocesor

Zpracuje makrodefinice, príkazy pro podmínený preklad nebo príkazy pro vlození textu z jiného souboru do zdrojového programu.

Linker

Neboli *sestavovací program* prideli relativnimu kodu (respektive jeho relativnim adresam) absolutni adresy a provede všechny odkazy (najde cesty) na dosud nezname identifikátory (napr. na volane knihovni funkce uložené v souborech .LIB). Vysledkem práce linkeru je primo spustitelny program (.EXE). Zaroven spoji jeden nebo více objektových souborů vygenerovaných překladačem do jediného spustitelného souboru.

Funkce linkeru

Objektové soubory jsou moduly programu obsahující strojový kód a informace pro linker. Tyto informace se v nich vyskytují ve formě dvou různých typů symbolických definic:

- ***Definované*** nebo ***exportované*** symboly jsou funkce nebo proměnné, které jsou zastoupeny v modulu reprezentovaném objektovým souborem. Tyto symboly by měly být dostupné ostatním modulům.
- ***Nedefinované*** nebo ***importované*** symboly jsou vnitřně nedefinované funkce nebo proměnné, které jsou volány, nebo na které se objekt odkazuje.

Ve zkratce, účelem linkeru je řešení referencí nedefinovaných symbolů vyhledáváním symbolů v ostatních objektových souborech, které je definují. Nedefinovaný symbol je nahrazen adresou symbolu, který jej definuje.

Knihovny

Linker často pracuje s objekty, které jsou uspořádány ve větších celcích, tzv. knihovnách. Některé linkery neobsahují na výstupu celou knihovnu, ale obsahují pouze její symboly, na které je odkazováno z objektových souborů případně jiných knihoven. Existuje celá řada knihoven majících různý účel. Jedna nebo více systémových knihoven bývají obvykle připojovány v linkovacím procesu standardně.

Sdílené knihovny

Moderní prostředí operačních systémů umožňuje tzv. **dynamické sestavování**, řešící některé nespecifikované symboly až po spuštění programu, z čehož vyplývá, že spustitelný soubor stále obsahuje nespecifikované symboly a seznam objektů a knihoven obsahujících jejich definice. Tyto objekty a knihovny se načtou až při spuštění programu, kdy se provede konečné sestavení.

Tento přístup má dvě výhody:

Často používané knihovny (např. Standardní systémové knihovny) nemusí být obsaženy v každém adresním prostoru každého programu zvlášť, ale pouze jednou, přičemž programy je mohou sdílet.

Pokud je chybně fungující knihovna nahrazena, všechny programy, které k ní dynamicky přistupují mohou po svém restartu z této změny těžit. Programy, které se na funkci obsaženou v knihovně odkazují staticky musí být nejprve přelinkovány.

Debugger

Prekladac například musí zajistit generování dostatecných informací pro symbolické ladení programu (jména a umístení promenných a procedur, odkazy na zacátky zdrojových rádku apod.) nebo musí do generovaného programu vkládat volání speciálních sluzeb pro vyhodnocování statistik o cinnosti programu.

Loader

Je súčasťou operačného systému, ktora je zodpovedna za jednu zo základných fáz v procese načítania a spustenia programu. Vykonava teda spustenie programu prečítaním obsahu spustiteľných súborov (**spustiteľné súbory** - súbory obsahujúce text programu) do pamäte. Potom vykonáva ďalšie potrebné prípravné úlohy, po ktorom programový kód môže konečne bezat. Je spustený ked operačný systém odovzdá riadenie nacitanemu programu. Specialne systemy, ktore maju program (programy) pevne dane na nejakej ROM pamati, neportebuju loader.

Start programu z pohledu OS

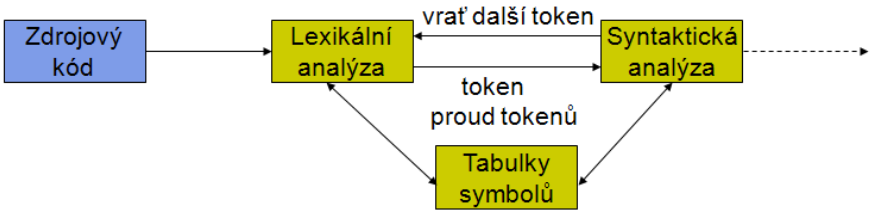
- „Někdo“ (shell,...) požádá OS o spuštění programu:
  - Jméno spustitelného souboru
  - Parametry (MS: vcelku, unix: po částech)
- OS (loader)
  - vytvoří proces, vyhradí základní kvantum paměti
  - načte instrukce (a inicializační data) ze souboru do paměti
  - doplní referencované DLL, pospojuje odkazy a relokuje
  - oživí proces na definovaném místě
- Proces běží a prostřednictvím systémových volání žádá o
  - interakci s okolím (soubory, uživatelská rozhraní, komunikace,...)
  - přidělení další paměti, vytvoření dalších vláken, ...
  - ukončení (s návratovým kódem)
- Proces je zlikvidován (na vlastní žádost nebo po pádu), zabit, ...
  - nadřazenému procesu je předán návratový kód
  - význam návratového kódu je závislý na konvencích

Start programu z pohledu C++ knihoven

- OS (loader)
  - oživí proces na definovaném místě v kódu standardní knihovny
- Knihovní kód
  - požádá OS o první kvantum paměti pro dynamickou alokaci
  - inicializuje struktury dynamické alokace
  - naváže standardní vstupy a výstupy na OS
  - inicializuje všechny globální proměnné
  - zkonvertuje parametry programu
  - zavolá *main*
  - uklidí globální proměnné
  - zavře všechny otevřené soubory
  - požádá o ukončení s návratovým kódem vráceným z *main*
- Proces je zlikvidován

Konecné automaty a lexikální analýza

Lexikální analyzátor je první fází prekladace. Jeho hlavním úkolem je císt znaky ze vstupu a na svuj výstup dávat symboly, které dále pouzívá syntaktický analyzátor. Lexikální analýza zabírá 60-80% doby překladu. Znamou implementaciou generatoru lexikálních parserov je Lex.



Pojmy

Token - Výstup lexikální analýzy a vstup syntaktické analýzy. Na straně syntaktické analýzy se nazývá **terminál**. Muze byt i množina řetězců, které produkují stejný token. Token muze mit i ruzne **atributy** (typicky jeden) – pokud je rozpoznavan vice patterny, nebo pokud je literal.

Pattern – Pravidla, která popisují množinu řetězců pro daný token**.** Obvykle se využívá **regulárních výrazů (-> regularni jazyky -> rozpoznavany konečnými automaty**). Automat je restartovan po kazdem rozpoznanem tokenu.

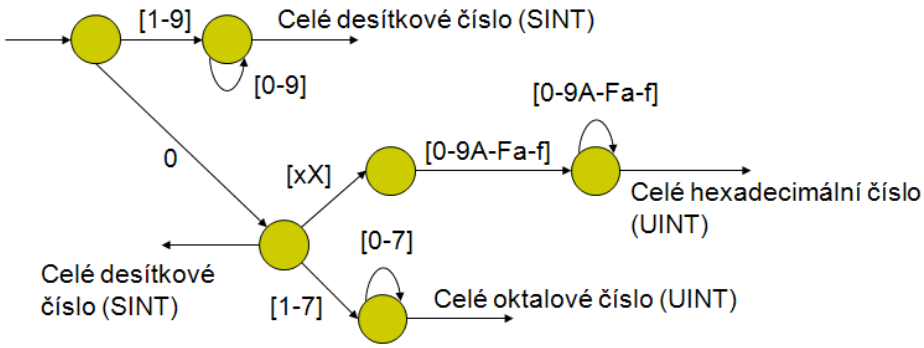
Lexém, lexikální element - Sekvence znaků ve zdrojovém kódu, která odpovídá nějakému patternu nějakého tokenu. Některé lexémy nemají výstup jako token (komentář).

Literál - Konstanta, má svoji hodnotu.

Konecne automaty

Konečným automatem nazýváme pětici **A = (Q,X,δ,q0,F)** kde:

- **Q** - konečná neprázdná množina stavů (stavový prostor)
- **X** - konečná neprázdná množina symbolů (vstupní abeceda)
- **δ** - zobrazení Q × X → Q (přechodová funkce)
- **q0** ∈ Q (počáteční stav)
- **F** ⊆ Q (množina koncových stavů)



Lexikalni chyby

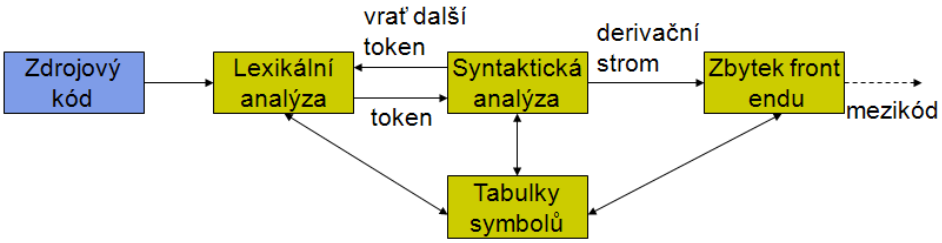
Chyby, které nastanou v okamžiku, kdy konečný automat nemůže pokračovat dál a není v konečném stavu - neznámý znak, neukončený řetězec do konce řádky. Zotaveni z chyb muze probihat **ignorovanim** chyby, nebo **domyslet si chybějící znak(y)**. Lexikalni chyby muzou dost rozhodit syntaktickou analyzu.

Bufferování vstupu

Jedno z možných zrychlení: čtení vstupního souboru po blocích (bufferech) a práce automatu v paměti bufferu. Tento přístup má však potíže s vložením souboru (#include), protože to znamená „vnoření“ bufferu.

Syntaktická analýza

Behem syntaktické analýzy se prekladac snazí zjistit, zda zdrojový text tvorí vetu odpovídající gramatice prekládaného jazyka. K tomu vyuzívá posloupnost lexikálních symbolu získanou jako výsledek lexikální analýzy. Pokud text obsahuje nejaké chyby, prekladac je nahlásí a obvykle provede urcité zotavení tak, aby i pri výskytu chyb mohl pokracovat dále v cinnosti a odhalit prípadné dalsí chyby. Dulezitymi ulohami jsou i **syntaxi rizeny preklad** (ridi cinnost celeho prekladace) a **stavba derivačního stromu**. Pri implementaci prekladace se obvykle pouzívá jednoho ze dvou základních prístupu prekladu **shora dolu** nebo **zdola nahoru**. Uvedeným dvema prístupum odpovídají také dve základní trídy gramatik, konkrétne **LL** a **LR gramatiky**.



Gramatika

Generativní gramatikou nazýváme čtveřici **G = (V_N,V_T,S,P)**:

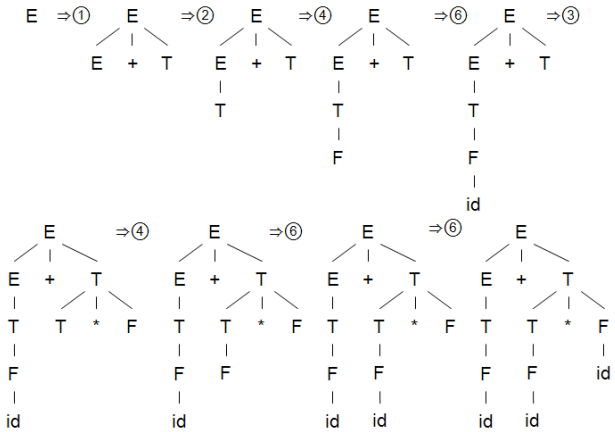
- **V_N** - konečná množina neterminálních symbolů
- **V_T** - konečná množina terminálních symbolů
  Obě abecedy jsou neprázdné a disjunktní!
- **S ∈ V_N** - počáteční neterminální symbol
- **P** - **systém produkcí (pravidel)** ***u→v***, kde *u, v* ∈ $(V_N \cup V_T)^*$ a *u* obsahuje alespoň jeden neterminální symbol.

Jazyk L(G) generovaný gramatikou **G** definujeme takto:

$L(G) = \{w \mid w \in V_T^* \ \& \ S \Rightarrow^* w\}$

Derivacni strom

Je **grafická reprezentace derivací** (odvozeni, prepsani nekterym pravidlem) použitím stromů, kde uzly stromu jsou neterminály i terminály a hrany vedou od neterminálu z levé strany pravidla na všechny symboly z pravé strany.



LL techniky

Pokud je α řetězec symbolů gramatiky, pak **FIRST(α)** je množina terminálů, kterými začínají všechny řetězce derivované z α. Pokud α může zderivovat na Λ, pak Λ je také ve FIRST(α).

Definujme **FOLLOW(A)** pro neterminál A jako množinu terminálů, které se mohou vyskytovat těsně za A v nějakém řetězci, který vznikl derivací z počátečního neterminálu gramatiky (S ⇒* αAaβ, pro nějaká α a β). Pokud je A nejpravější symbol v nějakém přepisu, pak i $ je ve FOLLOW(A).

Analyza shora dolu (LL technika)

Je pokus najít nejlevější derivaci pro vstupní řetězec nebo pokus o zkonstruovani derivačního stromu pro daný vstup počínaje kořenem a přidáváním uzlů do stromu v preorderu. Je rešena obvykle **rekurzivním sestupem** (rekurzivni volani procedur, jedna procedura, funkce pro každý neterminal gramatiky) nebo **nerekurzivní analýzou s predikcí** (automat s explicitním zásobníkem). Každé z těchto řešení má (okrem ineho) potíže s levou rekurzí v gramatice. Pro analyzu s hora dolu se proto zavadi speciální tvar gramatiky LL(1).

LL(1) gramatika

Bezkontextová gramatika G = (T,N,S,P) je **LL(1) gramatika**, pokud pro každá 2 pravidla A→α, A→β ∈ P, kde α≠β, a každé 2 **levé větné formy** uAγ, vAδ, kde u,v∈T* a γ,δ ∈ (T∪N)*, platí FIRST(αγ) ∩ FIRST(βδ) = ∅.

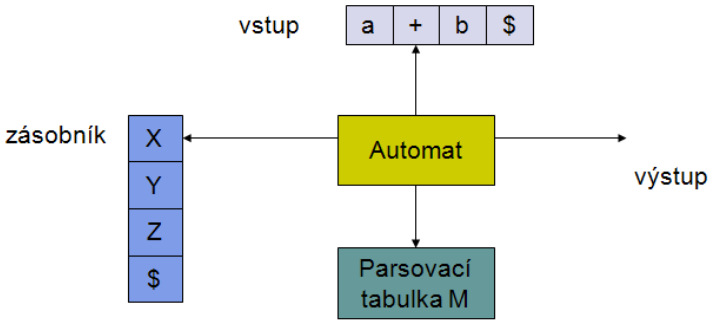
Prevod gramatiky na gramatiku typu LL(1)

Ne vždy se použitím nasledujicich pravidel prevod zdari!

- **Odstranění nejednoznačnosti** – Lze sestrojit různé derivační stromy pro stejné vstupní slovo. Je třeba si vyjasnit který derivační strom je ten správný a prispusobit tomu i pravidla gramatiky (**dangling else**).
- **Eliminace levé rekurze** - Gramatika, která obsahuje levou rekurzi, nemuze být typu LL(1) a je teda problémem pro analýzu shora-dolů. Obecne muzeme zleva rekurzivní pravidlo zapsat jako:
  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$, retezce β_i nezacínají nonterminálem A.
  Takové pravidlo muzeme prepsat zavedením nového nonterminálu A' jako
  $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$
  $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \Lambda$
- **Levá faktorizace** - Když není jasno, které ze dvou možných variant si vybrat je nutne přepsat ekvivalentně gramatiku s tím, že odložíme rozhodnutí na pozdější dobu, až bude vidět, které z pravidel si vybrat:
  $A \rightarrow \beta\alpha_1 \mid \beta\alpha_2 \mid \ldots \mid \beta\alpha_n$.
  Muzeme provést jejich vytknutí opet zavedením nového nonterminálu A'
  s pravidly:
  $A \rightarrow \beta\ A'$
  $A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

Nerekurzivní analýza s predikcí – automat

Analyzátor pracující metodou shora dolu muzeme popsat jako zásobníkový automat tvorený **vstupní páskou** (tokeny), **zásobníkem** (obsahuje symboly gramatiky), **výstupní páskou** (informace o pouzitem pravidlu, nebo chyba) a **rozkladovou (parsovaci) tabulkou** (dvojrozměrná tabulka **M[A, a]**, kde A je neterminál a a je terminál).



Funkce automatu

Počáteční konfigurace - Vstupní ukazatel (cteci hlava) ukazuje na začátek vstupu (pasky). Na zásobníku je počáteční neterminál gramatiky nad symbolem $.

V každém kroku se rozhoduji podle symbolu X na vrcholu zásobníku a terminálu **a**, který je právě na vstupu:

- Pokud je **X=a=$**, pak se parser s úspěchem zastaví
- Pokud je **X=a≠$**, pak se vyzvedne X ze zásobníku a ukazatel vstupu se přesune o terminál dále
- **Je-li X neterminál**, pak rozhodne položka M[X, a]. Pokud je tam přepisovací pravidlo, pak se nahradí na zásobníku X pravou stranou přepisovacího pravidla (s nejlevějším symbolem na vrcholu). Zároveň je generován výstup použití příslušného pravidla. Pokud je v tabulce **error**, pak se nahlásí chyba.

Konstrukce rozkladove (parsovaci) tabulky LL automatu

Pro každé přepisovací pravidlo A→α gramatiky proveď následující kroky:

- Pro ∀ a∈FIRST(α) přidej A→α do M[A, a]
- Pokud Λ∈FIRST(α), pak přidej A→α do M [A, b]
  ∀ b∈FOLLOW(A). Pokud navíc $∈FOLLOW(A), přidej A→α do M[A, $]

Pro každé prázdné políčko M nastav **error.**

LR techniky

Pokud je α řetězec symbolů gramatiky, pak **$FIRST_k(\alpha)$** je množina slov terminálů o délce nejvýše k, kterými začínají všechny řetězce derivované z α. Pokud α může zderivovat na Λ, pak Λ je také ve $FIRST_k(\alpha)$.

Definujme **$FOLLOW_k(A)$** pro neterminál A jako množinu slov terminálů o délce nejvýše k, které se mohou vyskytovat těsně za A v nějakém řetězci, který vznikl derivací z počátečního neterminálu gramatiky ($S \Rightarrow^* \alpha A u \beta$, pro nějaká α a β). Pokud je A nejpravější symbol v nějakém přepisu, pak i $ je ve $FOLLOW_k(A)$.

Analyza shora dolu (LR technika)

Analyza je známa jako syntaktická analýza typu **presun – redukce**. Syntaktická analýza presun-redukce se pokousí budovat derivacní strom ze vstupního retezce od listu smerem ke korenu. Na tento postup muzeme pohlízet rovnez jako na redukci vstupní vety w na startovací symbol gramatiky. Pri kazdém z redukcních kroku je podřetězec odpovídající pravé straně pravidla gramatiky nahrazen neterminálem z levé strany pravidla.

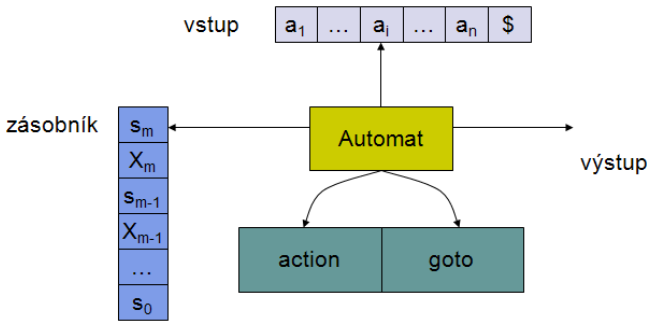
Používáno známými generátory parserů (Bison). Dá se implementovat stejně efektivně jako metody shora dolů. Třída rozpoznávaných jazyků LR(1) je vlastní nadmnožina LL(1).

LR(k) gramatika

Bezkontextová gramatika G = (T,N,S,P) je **LR(k) gramatika** pro k≥1, pokud pro každá 2 pravidla $A \rightarrow \alpha$, $A \rightarrow \beta \in P$, kde $\alpha \neq \beta$, a každé 2 **pravé větné formy** $\gamma A u$, $\delta A v$, kde $u,v \in T^*$ a $\gamma,\delta \in (T \cup N)^*$, platí $FIRST_k(u) \cap FIRST_k(v) = \emptyset$.

Automat pro LR parser

Skládá se ze **vstupu, výstupu, zásobníku, rídicího programu a rozkladové tabulky**, která má dve cásti (**akce a prechody**). Rídicí program je stejný pro všechny LR analyzátory, mení se pouze zpusob konstrukce rozkladové tabulky. Rídicí program cte znaky ze vstupní vyrovnávací pameti jeden po druhém. Uzívá zásobník k ulození retezce ve tvaru $s_0X_1s_1X_2s_2 \ldots X_ms_m$, kde s_m je na vrcholu zásobníku. X_i oznacuje **symbol gramatiky** a s_i je symbol nazvaný **stav**. Kazdý stavový symbol sumarizuje informace obsazené v zásobníku pod ním a kombinace stavového symbolu na vrcholu zásobníku a aktivního vstupního symbolu se uzívá pro indexování prvku rozkladové tabulky urcujícího provádenou **akci**.



Konfigurace LR analyzátoru je dvojice, jejíz první slozkou je obsah zásobníku a druhou slozkou je dosud neprectená cást vstupního retezce:
$(s_0X_1s_1X_2s_2 \dots X_ms_m, a_ia_{i+1} \dots a_n\$)$

Akce analyzátoru je urcena stavem na vrcholu zásobníku s_m, a aktuálním vstupním symbolem a_i. Provádená akce je funkcní hodnotou **akce[s_m; a_i]** získané z rozkladové tabulky.

Funkce LR automatu

Pocáteční konfigurace - Ukazatel vstupu (cteci hlava) ukazuje na zacátek vstupu (pasky). Na zásobníku je pocáteční stav s_0.

V každém kroku podle s_m a a_i adresuji action[s_m, a_i]. Akce muze byt:

- **Posun (shift) s**, kde s je **nový stav**. Posune pásku o 1 terminál, na zásobník se pridá a_i a s.
  Analyzátor prejde do konfigurace ($s_0X_1s_1X_2s_2 \dots X_ms_ma_is$; $a_{i+1} \dots a_n\$$).
- **Redukce (reduction)** podle pravidla gramatiky A→α. Zruší se ze zásobníku r=|α| dvojic (s_k, X_k), na zásobník se pridá A a stav goto[s_{m-r}, A] (s_{m-r} je stav, co zbyl na vrcholu zásobníku po odmazání).
  Analyzátor prejde do konfigurace ($s_0X_1s_1X_2s_2 \dots X_{m-r}s_{m-r}As$; $a_ia_{i+1} \dots a_n\$$)
- **Accept** - Vstupní slovo je úspešne rozpoznáno. Generuje výstup.
- **Error** - Vstupní slovo neodpovídá gramatice.

Tabulka pro povestnou gramatiku

stav	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Funkce LR automatu na povestne gramatice

Zásobník	Vstup	Akce
0	**id+id*id$**	s5
0 **id** 5	**+id*id$**	r6: F→**id**
0 F 3	**+id*id$**	r4: T→F
0 T 2	**+id*id$**	r2: E→T
0 E 1	**+id*id$**	s6
0 E 1 **+** 6	**id*id$**	s5
0 E 1 **+** 6 **id** 5	***id$**	r6: F→**id**
0 E 1 **+** 6 F 3	***id$**	r4: T→F
0 E 1 **+** 6 T 9	***id$**	s7
0 E 1 **+** 6 T 9 * 7	**id$**	s5
0 E 1 **+** 6 T 9 * 7 **id** 5	**$**	r6: F→**id**
0 E 1 **+** 6 T 9 * 7 F 10	**$**	r3: T→T * F
0 E 1 **+** 6 T 9	**$**	r1: E→E **+** T
0 E 1	**$**	acc

Konstrukce rozkladove (parsovaci) tabulky LR automatu

Rozšíření gramatiky G=(T,N,S,P) je gramatika G'=(T,N',S',P'), kde N'=N∪{S'}, P'=P∪{S'→S}. Není třeba provádět, pokud S je na levé straně jednoho pravidla a není na žádné pravé straně pravidel. Cílem je pomoci parseru s rozpoznáním konce parsování.

Oteckovana pravidla

Otečkované pravidlo gramatiky G je pravidlo, které má na pravé straně na nějaké pozici speciální symbol tečky. Speciální znamená, že stejné pravidlo s tečkou na různých pozicích na prave straně se chápe jako různá otečkovaná pravidla. Zároveň však tato tečka není terminálem ani neterminálem gramatiky. Otečkované pravidlo se také nazývá LR(0) položka.

Ukázka pro pravidlo E → E + T:

E → ♦E + T	E → E + ♦T
E → E ♦+ T	E → E + T♦

Operace uzávěru

Mějme množinu otečkovaných pravidel I z gramatiky G. Definujme operaci CLOSURE(I) jako množinu otečkovaných pravidel zkonstruovaných z I následujícím postupem:

- Přidej do CLOSURE(I) množinu I
- ∀ A→α♦Bβ∈CLOSURE(I), kde B∈N, přidej ∀ B→γ∈P do CLOSURE(I) otečkované pravidlo B→♦γ, pokud tam ještě není. Toto opakuj tak dlouho, dokud přibývají pravidla do CLOSURE(I)

I={S'→♦E}.
CLOSURE(I) = {S'→♦E; E→♦E + T; E→♦T; T→♦T * F; T→♦F; F→♦(E); F→♦id}

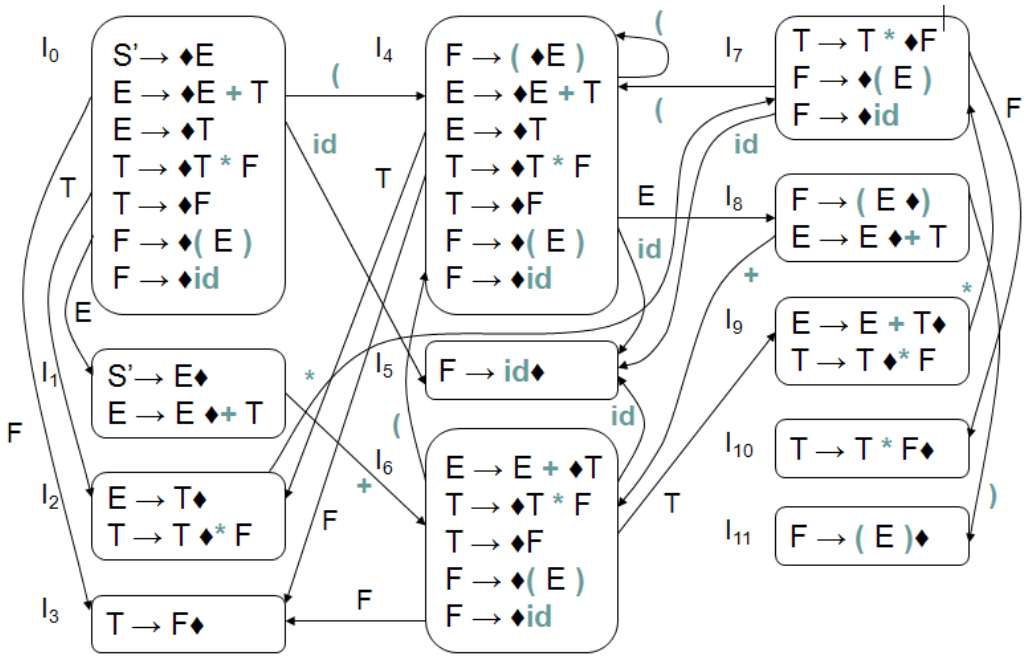
Operace prechodu

GOTO(I, X) pro množinu otečkovaných pravidel I a symbol gramatiky X definujeme jako uzávěr množiny všech pravidel A→αX♦β takových, že A→α♦Xβ∈I.

Konstrukce kanonické kolekce množin LR(0) položek

Mějme rozšířenou gramatiku G'=(T,N',S',P'). Konstrukce kanonické kolekce C množin LR(0) položek:

- Na počátku C={ CLOSURE({S'→♦S}) }
- ∀ I∈C a ∀ X∈T∪N' takové, že GOTO(I, X)∉C ∧ GOTO(I, X)≠∅, přidej GOTO(I, X) do C. Toto opakuj, dokud něco přibývá do C



Platné položky

Otečkované pravidlo $A \rightarrow \beta_1 \blacklozenge \beta_2$ je platnou položkou pro schůdný (viable) prefix $\alpha\beta_1$, pokud ∃ pravá derivace $S' \Rightarrow^{+} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$. Velká nápověda pro parser, zda provádět posun nebo redukci, pokud je na zásobníku $\alpha\beta_1$.

Základní věta LR parsování: Množina platných položek pro schůdný prefix γ je přesně množina položek dosažitelná z počátečního stavu přes cestu γ deterministickým konečným automatem zkonstruovaným z kanonické kolekce s přechody GOTO.

Konstrukce SLR(1) automatu

Mějme rozšířenou gramatiku G'. Tabulky SLR(1) automatu se pak zkonstruují následujícím způsobem:

- Zkonstruuj kanonickou kolekci C LR(0) položek
- Stav *i* je zkonstruován z I_i. Instrukce automatu pro stav *i* se určují podle následujících pravidel:
  - $A \rightarrow \alpha \blacklozenge a\beta \in I_i$, $a \in T \wedge GOTO(I_i, a) = I_j$, pak action[i, a] = shift j
  - $A \rightarrow \alpha \blacklozenge \in I_i$, pak $\forall a \in FOLLOW(A) \wedge A \neq S'$ je action[i, a] = reduce $A \rightarrow \alpha$
  - $S' \rightarrow S \blacklozenge \in I_i$, pak action[i, $]=accept
- Pokud v předchozím kroku nastal konflikt, gramatika není SLR(1) gramatikou a automat není možno zkonstruovat
- Tabulka goto je pro stav *i* a $\forall A \in N'$: pokud $GOTO(I_i,A)=I_j$, pak goto [i,A]=j
- Všechna nedefinovaná políčka se nastaví na error
- Počáteční stav parseru je ten, jehož množina otečkovaných pravidel obsahuje $S' \rightarrow \blacklozenge S$

Při konstrukci SLR(1) ve stavu *i* je nastaveno action[i, a] ∀a∈FOLLOW(A) na redukci A→α, pokud A→α♦∈I_i. V některých situacích, kdy je *i* na vrcholu zásobníku, tak schůdný prefix βα je takový, že βA nemůže být následována terminálem **a** v žádné pravé větné formě. Tedy redukce podle A→α je pro vstup a neplatná.

Řešení: vložit do stavů více informace, abychom se vyhnuli neplatným redukcím.

LR(1) otečkované položky

Tato informace navíc je uchována pro každé otečkované pravidlo ve stavu přidáním terminálu. Takové pravidlo pak má tvar [A→α♦β,a], kde A→αβ∈P, a∈T. Otečkované pravidlo v tomto tvaru nazýváme LR(1) položka. Terminál **a** nazýváme výhled (lookahead).

- Výhled nemá význam pro A→α♦β, kde β≠Λ.
- Redukce A→α se provádí pouze tehdy, pokud [A→α♦, a]∈I_i pro aktuální stav *i* a terminál **a** na vstupu.
- Taková množina terminálů je ⊆FOLLOW(A).

LR(1) položka [A→α♦β,a] je platná pro schůdný prefix γ, pokud ∃ pravá derivace taková, že $S \Rightarrow^{+} \delta Aw \Rightarrow \delta\alpha\beta w$, kde:

- γ=δα
- buď **a** je první symbol w nebo je w=Λ a **a** je $

Upravená operace uzávěru pro LR(1) položky

Mějme množinu LR(1) položek I pro gramatiku G. Definujme operaci CLOSURE1(I) jako množinu LR(1) položek zkonstruovaných z I následujícím postupem:

- Přidej do CLOSURE1(I) množinu I
- ∀ [A→α♦Bβ,a]∈CLOSURE1(I), kde B∈N, přidej ∀ B→γ∈P a ∀b∈FIRST(βa) do CLOSURE1(I) LR(1) položku [B→♦γ,b], pokud tam ještě není. Toto opakuj tak dlouho, dokud přibývají pravidla do CLOSURE1(I)

I={[S'→♦S,$]}

CLOSURE1(I) =

- S'→ ♦S, $ β=Λ,FIRST(β$)=FIRST($)={$}
- S→ ♦CC, $ β=C,FIRST(C$)={c,d}
- C→ ♦cC, c/d
- C→ ♦d, c/d

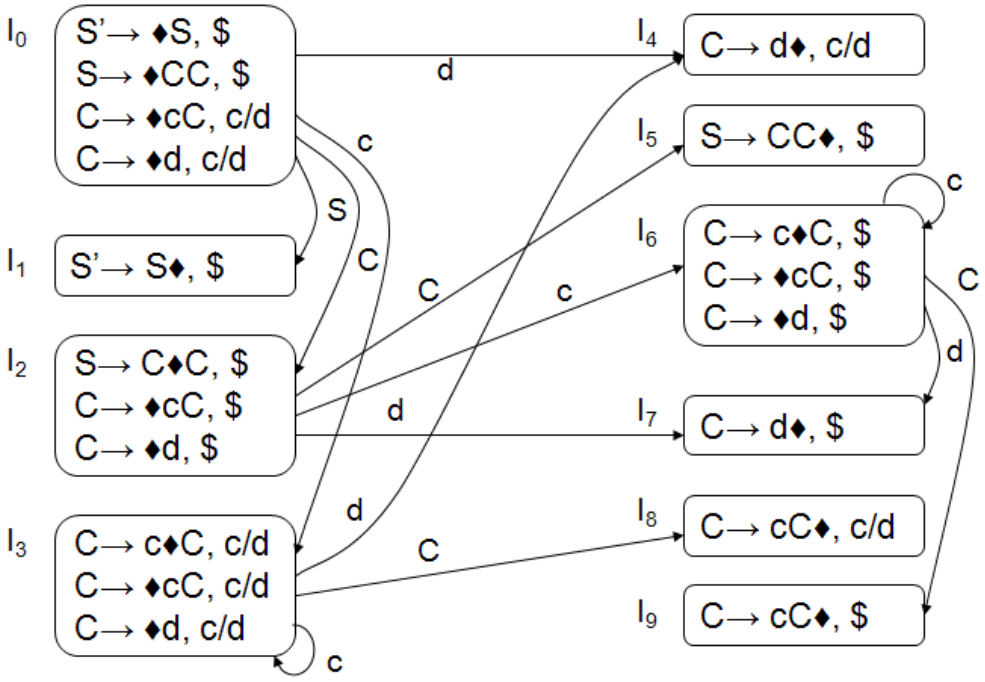
Upravená operace přechodu pro LR(1) položky

Definujme operaci GOTO1(I, X) pro množinu LR(1) položek I a symbol gramatiky X jako CLOSURE1 množiny všech pravidel [A→αX♦β,a] takových, že [A→α♦Xβ,a]∈I.

Konstrukce kanonické kolekce LR(1) položek

Mějme rozšířenou gramatiku G'=(T,N',S',P'). Konstrukce kanonické kolekce C LR(1) položek:

- Na počátku C={ CLOSURE1({[S'→♦S,$]}) }
- ∀ I∈C a ∀ X∈T∪N' takové, že GOTO1(I, X)∉C ∧ GOTO1(I, X)≠∅, přidej GOTO1(I, X) do C. Toto opakuj, dokud něco přibývá do C



Příklad LR(1) gramatiky, která není SLR(1)

S'→S, S→CC, C→cC, C→d

Konstrukce LR(1) parseru

Mějme rozšířenou gramatiku G'. Tabulky LR(1) automatu se pak zkonstruují následujícím způsobem:

- Zkonstruuj kanonickou kolekci C LR(1) položek
- Stav *i* je zkonstruován z I_i. Instrukce automatu pro stav *i* se určují podle následujících pravidel:
  - [A→α♦aβ,b]∈I_i,a∈T ∧ GOTO1(I_i,a)=I_j, pak action[i,a]=shift j
  - [A→α♦,a]∈I_i ∧ A≠S', pak action[i,a]=reduce A→α
  - [S'→S♦,$]∈$I_i$, pak action[i,$]=accept
- Pokud v předchozím kroku nastal konflikt, gramatika není LR(1) gramatikou a automat není možno zkonstruovat
- Tabulka goto je pro stav *i* a ∀A∈N': pokud GOTO1(I_i,A)=I_j, pak goto [i,A]=j
- Všechna nedefinovaná políčka se nastaví na error
- Počáteční stav parseru je ten, jehož množina LR(1) položek obsahuje [S'→♦S,$]

LALR

LALR=LookAhead-LR. Běžně využívána v praxi (Bison). Většina programovacích jazyků se dá vyjádřit jako LALR. Tabulky parseru jsou významně menší než LR(1). SLR a LALR parsery mají stejný počet stavů, LR parsery mají větší počet stavů. Pro programovací jazyky řádově stovky stavů. LR(1) tabulky budou mít řádově tisíce stavů pro stejnou gramatiku.

Jak zmenšit tabulky?

Nápad: sloučit množiny se stejným jádrem do jedné množiny včetně sloučení GOTO1. Jádro: množina otečkovaných pravidel bez výhledu (LR(0) položky). Při tomto slučování nemůže vzniknout shift/reduce konflikt:

- Mějme následující dvě konfliktní pravidla: [A→α♦,a] a [B→β♦aγ,b]
- Jádra množin jsou stejná, takže v množině, kde je [A→α♦,a] musí být i [B→β♦aγ,c] pro nějaké **c**, takže už tam konflikt shift/reduce byl

Může vzniknout reduce/reduce konflikt.

Snadná metoda konstrukce LALR(1) parseru

Mějme rozšířenou gramatiku G'. Tabulky LALR(1) automatu se pak zkonstruují následujícím způsobem:

- Zkonstruuj kanonickou kolekci C LR(1) položek
- Pro každé jádro v kolekci C najdi všechny množiny s tímto jádrem a nahraď je jejich sjednocením
- Nechť C'={ J_0, J_1, …, J_m } je výsledná kolekce LR(1) pravidel
- Tabulka action je zkonstruována pro C' stejně jako pro LR(1) parser
- Pokud nastal konflikt, gramatika není LALR(1) gramatikou
- Nechť množina J∈C' vznikla sjednocením několika položek I_i ($J=I_1 \cup I_2 \cup \ldots I_k$). Pak jádra GOTO1($I_1$,X), …, GOTO1($I_k$,X) jsou stejná, protože i I_1, …, I_k mají stejná jádra. Nechť K je sjednocení všech množin položek, které mají stejné jádro jako goto(I_1,X). Pak GOTO1(J,X)=K

Významná nevýhoda – je třeba zkonstruovat plné LR(1).

Syntaxí rízený preklad a atributové gramatiky

Prozatím jsme se zabývali pouze kontrolou, zda je veta, kterou prekládáme, prvkem prekládaného jazyka. Ted priradíme k syntaktickým konstrukcím dalsí informace – atributy, které se vyhodnocují na základe **sémantických pravidel**. Pro pripojení sémantických pravidel k pravidlum gramatiky existují dve notace: **syntaxí rízené definice a prekladová schemata**. Syntaxí rízené definice jsou specifikací prekladu na vysoké úrovni abstrakce. Obecne jak pri prekladu pomocí syntaxí rízených definic, tak i pri pouzití prekladových schemat rozkládáme vstupní posloupnost symbolu, budujeme derivacní strom a potom **procházíme stromem** tak, abychom **vyhodnotili sémantická pravidla** v uzlech derivačního stromu.

Vyhodnocením sémantických pravidel muze být **generování kódu**, ukládání informací do tabulky symbolu, vydávání zpráv o chybách nebo provádení nejakých jiných cinností. **Výsledkem vyhodnocení sémantických pravidel je preklad posloupnosti vstupních symbolu**.

Speciální prípady syntaxí rízeného prekladu lze implementovat v jednom pruchodu s vyhodnocením sémantických pravidel behem analýzy, bez explicitní konstrukce derivacního stromu nebo grafu ukazujícího závislosti mezi atributy. Vzhledem k tomu, ze jednopruchodová implementace je dulezitá pro efektivitu prekladace, je velká cást této kapitoly venována studiu takových prípadu. Jedna dulezitá podtrída, zvaná **L-atributové defnice**, zahrnuje témer vsechny preklady, které lze provádet bez explicitní konstrukce derivacního stromu.

Syntaxí rízené definice

Každy symbol gramatiky má definovanou množinu atributů. Lze to chápat jako záznam (struktura). Atributy jsou dvojiho druhu: **syntetizovane** a **dedicne**. Atributy mohou reprezentovat cokoliv. Hodnoty atributů jsou definovány sémantickými pravidly přidaných k přepisovacím pravidlům gramatiky. Podle **grafu závislosti** se pak určí pořadí vyhodnocení sémantických pravidel. Vyhodnocení sémantických pravidel pak určí hodnoty atributů.

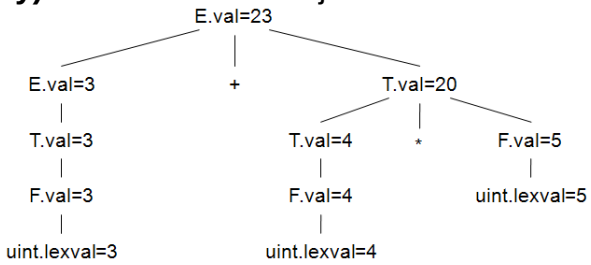
Druhy atributů

Každé přepisovací pravidlo A→α má definovanou **množinu sémantických pravidel** ve formě b = f(c_1,…,c_k), kde f je funkce a c_i jsou atributy symbolů gramatiky daného přepisovacího pravidla. b je **syntetizovaný atribut** neterminálu A. b je **dědičný atribut** symbolu gramatiky z pravé strany daného přepisovacího pravidla.

Atributová gramatika - je syntaxí řízená definice, kde funkce v sémantických pravidlech **nemají vedlejší efekty**:

1.	$E \rightarrow E_R + T$	$E.val = E_R.val + T.val$
2.	$E \rightarrow T$	$E.val = T.val$
3.	$T \rightarrow T_R * F$	$T.val = T_R.val * F.val$
4.	$T \rightarrow F$	$T.val = F.val$
5.	$F \rightarrow (E)$	$F.val = E.val$
6.	$F \rightarrow$ uint	F.val = uint.lexval

Anotovaný (obarvený) derivační strom - je derivační strom s hodnotami atributů:



Syntetizované atributy - Atribut neterminálu z levé strany přepisovacího pravidla je vypočten na základě atributů symbolů z pravé strany. Běžně používány v praxi (Např. výpočet výrazu). Atributová gramatika, která používá pouze syntetizované atributy, se nazývá **S-atributová gramatika** (čistě syntetizovaná atributová gramatika). S-atributová gramatika se snadno použije při analýze zdola-nahoru (výpočet proběhne při redukci a atributy pro symbol jsou na zásobníku parseru).

Dědičné atributy - Hodnota dědičného atributu je vypočtena na základě atributů otce nebo atributů symbolů na stejné úrovni v derivačním stromě. Používají se k vyjádření závislosti nějaké syntaktické konstrukce na kontextu (Např. zda se identifikátor vyskytuje na levé nebo pravé straně přiřazení). Vždy lze přepsat atributovou gramatiku tak, aby byla S-atributová.

Vypocet hodot atributu

Graf závislosti je graf kde uzly jsou vytvořeny pro každý atribut každého uzlu anotovaného derivačního stromu. Pro každé sémantické pravidlo $b = f(c_1,\ldots,c_k)$ sestroj orientovanou hranu z uzlu grafu závislosti pro c_i do uzlu pro b. Najdeme **topologické třídění DAGu**, který vznikl konstrukcí grafu závislosti. Pokud vznikne při konstrukci grafu závislosti kružnice, pak nelze určit pořadí výpočtu. Jakékoliv topologické setřídění je dobré.

L-atributová gramatika

Atributová gramatika je **L-atributová** (zleva-doprava 1-průchodová), pokud ∀ dědičný atribut symbolu X_j z $A \rightarrow X_1 \ldots X_n$ závisí pouze na atributech symbolů $X_1,\ldots,X_{j-1}$ (vlevo od X_j). Tj. na dedicnych atributech A. Lze ji využít pro výpočet atributů přímo při analýze shora-dolů. Každá S-atributová gramatika je L-atributová.

Průchody derivačním stromem

Pokud atributová gramatika není L-atributová, není možné provést výpočet atributů přímo při parsování a je nutné samostatně zkonstruovat derivační strom a jako jedna z částí sémantické analýzy budou i průchody tímto derivačním stromem, které dopočítají atributy, které nešly spočítat během syntaxí řízeného překladu.

Chyby při sémantické analýze odpovídají statickým kontrolám

Neexistující proměnná, špatný počet parametrů volání funkce, špatně použitý operátor na nějaký typ…:

- **Kontrola typů** - použití operátorů na špatný typ (např: násobení ukazatelů).
- **Kontrola toku řízení** - zda je změna toku řízení oprávněná (např. V C příkaz break, pokud nejsem ve switch nebo cyklu).
- **Kontrola jedinečnosti** - některé objekty mohou být definovány právě jednou (např: návěští ve funkci, identifikátory globálních objektů)
- **Kontrola jmen** - některé konstrukty mají mít stejné jméno na svém začátku i na konci (procedura v assembleru).

Symbolické tabulky

Jsou dvojího druhu: tabulky, které stále rostou např. tabulky pro konstanty. Nebo tabulky, které přibývají/ubývají během překladu: např tabulky pro viditelnost identifikátorů v blocích.

Sémantická analýza postupně prochází symboly či skupiny symbolů získané ze syntaktické analýzy a přiřazuje se jim význam. Pokud například skupina symbolů představuje použití konkrétní proměnné, pak analyzátor zjišťuje zda je proměnná už deklarována (pokud je to požadováno, nevyžaduje například programovací jazyk Perl či PHP) a zda je správně použita vzhledem k jejímu datovému typu. Dále například u operací kontroluje zda jsou operandy správného typu, případně provede potřebnou konverzi datového typu. Pro všechny typy operátorů jsou stanoveny předpisy pro jejich operandy. Například pro některé jazyky platí, že pokud sčítáme řetězec a číslo, pak dojde k sémantické chybě. Pokud je to možné pak sémantický analyzátor je schopný implicitního přetypování, když operandy jsou různého datového typu. Implicitní přetypování se provádí na základě stanovených priorit jednotlivých typů. Pokud je tedy jeden operand typu s větší prioritou pak se automaticky druhý přetypuje na typ prvního operandu. U modernějších programovacích jazyků, kde je umožněné přetěžování funkcí, musí překladač správně určit, která funkce je volána a zda je volána správně, tedy zda předáváme správný počet a typ parametrů.

Při sémantické analýze se používá **tabulka symbolů**, která obsahuje veškeré objekty, které jsou při analýze zaznamenány. Jedná se například o proměnné, konstanty, funkce, procedury, atd. Jsou zde uloženy informace o datovém typu objektu, jeho velikosti, názvu, adrese, zda už objekt byl deklarován a zda už byl použit. Tabulka symbolů je používána nejenom ke kontrole objektů v sémantické analýze, ale i ve všech ostatních částech překladače. Například generátor cílového kódu ji prochází při statické či dynamické alokaci paměti. Jednotlivé záznamy jsou do tabulky většinou přidávány lexikálním analyzátorem. V této fázi překladu ale ještě nemáme všechny potřebné informace, a tak jsou záznamy doplňovány dalšími částmi překladače.

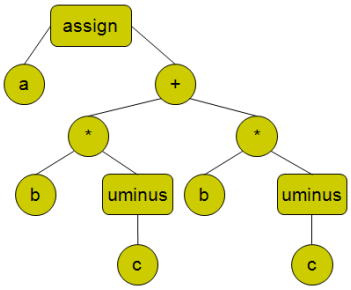
Reprezentace programu mezikódem

V modelu fazi prekladu prevádí prední cást prekladace (front end) zdrojový program do **intermediární reprezentace (mezikod)**, ze které dále zadní cást prekladace (back end) generuje cílový kód. Odděluje tedy front end od back endu. Zjednodusuje se prepracování prekladace pro jiný cílový jazyk (retargeting). Stací vytvořit pouze novou koncovou cást. Tato myslenka taky podporuje různé front endy pro stejný výstupní jazyk – podpora více jazyků pro stejný stroj (.Net). Mezikód lze optimalizovat s vyuzitím metod strojove nezávislých optimalizací.

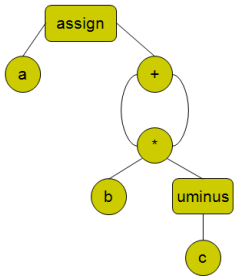
Jako intermediární reprezentace programu se pouzívají nejcasteji **stromy** (mozny i DAG) a **zásobníkový** (postfixovy - linearizovaná reprezentace syntaktického stromu) nebo **tríadresový kód** (také linearizovaná reprezentace syntaktického stromu, vypadá jako instrukce procesoru).

Priklad: a := b*-c+b*-c

Strom



DAG



Zasobnik (postfix)

a b c uminus * b c uminus * + assign

Triadresovy kod

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

Triadresovy kod

Tríadresový kód je **posloupnost príkazu**, které mají obecne tvar: **x := y op z**, kde x, y a z jsou jména, konstanty nebo prekladacem vytvorené docasné objekty; op predstavuje libovolný operátor, napr. nekterý z aritmetických, logických operátoru. Můhžou to být také operátory: nepodmíněný skok, podmíněný skok, mechanismus volání procedur/funkcí, adresové operátory… V operandech nemohou být zádné dalsí výrazy, príkaz obsahuje vzdy jen jediný operátor. Proto musí být slozitejsí výrazy rozlozeny na své nejjednodussí slozky s pouzitím docasných promenných vytvořených prekladacem.

Zde je videt zásadní rozdíl mezi zásobníkovým a tríadresovým kódem. Zásobníkový kód se odkazuje na operandy implicitne, na základe jejich pozice na zásobníku (TOP0, TOP1,…), zatímco tríadresový kód vsechny operandy pojmenovává. Tím se znacne zjednodusují optimalizace tríadresového mezikódu, pri nichz se mohou jednotlivé príkazy navzájem libovolne presouvat.

Implementace triadresoveho kodu

Čtveřice - Záznam se čtyřmi položkami **(*op*, *arg1*, *arg2*, *res*)**. Některé *arg* (typicky *arg2*) nebo i *res* nemusí být pro některé typy tříadresových instrukcí využity (unarni operace, operace skoku…). Operandy jsou odkazy do tabulek symbolů. Výhodnější pro optimalizování mezikódu. Lze s nimi snadno pohybovat, protože jména (operandu) zůstávají stejná.

	op	arg1	arg2	res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Trojice - Chceme se vyhnout generování přechodných jmen. Záznam s třemi položkami **(*op*, *arg1*, *arg2*)**. Operandy jsou odkazy do tabulek symbolů (pro jména a konstanty) nebo odkaz na jinou trojici (napr. cislem trojice ak ich ocislujem – cislo instrukcie). Tato reprezentace je uspornější oproti ctvericim, ale spatná manipulace při optimalizaci – přečíslování (trojic) v celém mezikódu.

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Nepřímé trojice - Jedno pole přímo s trojicemi. Druhé pole s odkazy na trojici. Lze s nimi dobře manipulovat při optimalizaci – změna pouze v poli s odkazy. Zhruba stejně paměťově náročné, jako čtveřice.

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

	stmt
[0]	(0)
[1]	(1)
[2]	(2)
[3]	(3)
[4]	(4)
[5]	(5)

Typy mezikódů

- **Vysokoúrovňový** mezikód - reprezentuje vstupní program. Používá se během fází, řešících konstrukce a pravidla vstupního jazyka. Užívá logické typy a operace vstupního jazyka. Nejčastěji se vyskytuje ve formě anotovaného AST (abstract syntax tree) - derivační strom podle abstraktní gramatiky.
- **Mezikód střední úrovně** - je nejčastější hranice mezi front a back-endem. Na vstupním jazyce nezávislá reprezentace. Užívá fyzické typy a operace na nich. Nejčastěji se vyskytuje ve formě čtveřic.
- **Nízkoúrovňový mezikód** - je ekvivalent strojových instrukcí. Nekompaktní forma, symbolické a relokované operandy. Před alokací registrů je to forma s neomezeným počtem virtuálních registrů.

Informace uložené v mezikódu střední úrovně

- **Seznam globálních proměnných** – velikost, inicializace, příznak konstantnosti, jméno (pro linker).
- **Další globální informace pro generovaný kód** - konstanty (reálné, řetězcové, strukturované), tabulky virtuálních funkcí, RTTI.
- **Seznam procedur**
- **Další informace pro debugger** - jména a konstrukce typů.

Popis procedury

- **Jméno** - pro linker a chybová hlášení.
- **Seznam parametrů** - fyzický typ (+ velikost), umístění podle volací konvence, jméno (pro debugger a chybová hlášení), logický typ (pro debugger a určení aliasů).
- **Seznam lokálních proměnných** - fyzický typ (+ velikost), jméno (pro debugger a chybová hlášení), logický typ (pro debugger a určení aliasů), proměnné ve vnořených blocích se obvykle povyšují na úroveň procedury.
- **Kód procedury**
  - **Plně sekvenční forma** - Posloupnost (pseudo-)instrukcí virtuálního stroje. Tok řízení popsán skokovými instrukcemi.
  - **Částečně sekvenční forma** - Tok řízení popsán grafem, jehož uzly jsou základní bloky. Každý základní blok obsahuje posloupnost (pseudo-)instrukcí. Skokové instrukce pouze na konci BB nebo zaznamenány jinak.
  - **Nesekvenční forma** - Tok řízení popsán grafem, jehož uzly jsou základní bloky. Tok dat uvnitř základního bloku popsán dagem. Různé formy podle stupně analýzy aliasů a rozsahů platnosti.

Kód v C

```
int gcd( int x, int y)
{ int z;
  if ( x > y )
  {
    z = y;
    y = x;
    x = z;
  }
  while ( x > 0 )
  {
    z = y % x;
    y = x;
    x = z;
  }
  return y;
}
```

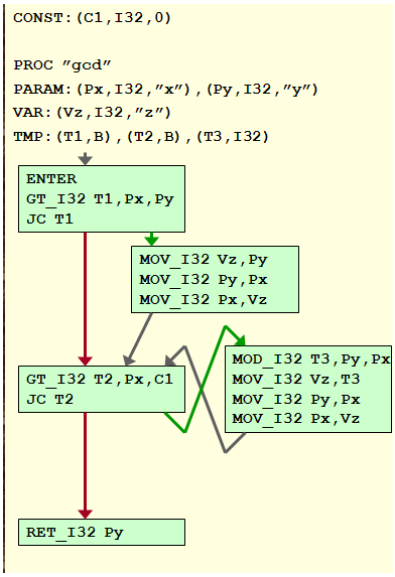
Sekvenční čtveřice

```
CONST:(C1,I32,0)

PROC "gcd"
PARAM:(Px,I32,"x"),(Py,I32,"y")
VAR:(Vz,I32,"z")
TMP:(T1,B),(T2,B),(T3,I32)

ENTER
GT_I32 T1,Px,Py
JF T1,L1
MOV_I32 Vz,Py
MOV_I32 Py,Px
MOV_I32 Px,Vz
L1:
GT_I32 T2,Px,C1
JF T2,L2
MOD_I32 T3,Py,Px
MOV_I32 Vz,T3
MOV_I32 Py,Px
MOV_I32 Px,Vz
JMP L1
L2:
RET_I32 Py
```

Částečně sekv. čtveřice



Preklad výrazu a programových struktur

Otazka zni teda, jak generovat mezikod pro ruzne vyrazy a programove struktury vstupniho jazyka. Přidáme ke každému symbolu gramatiky atributy obsahující informace o mezikódu:

- **Umístění (place)** – jméno objektu, který obsahuje hodnotu symbolu.
- **Kód (code)** – posloupnost tříadresových instrukcí, která „vypočítá“ symbol.
- **Adresa do kódu (label)** – absolutní nebo relativní adresa do tříadresových instrukcí.

Deklarace

Pri zpracování deklarací je základním úkolem prekladace **vytvárení tabulky symbolu**. S tím obvykle souvisí i shromazdování informací o datových typech a velikostech jejich reprezentace a pridelování adres promenným a slozkám záznamu, coz jiz není nezávislé na generovaném cílovém kódu. Pri vytvárení premístitelného kódu je adresa objektu definována vzdy dvema údaji: príslusností do urcité samostatni **adresované skupiny objektu** (napr. globální a lokální promenné nebo procedury, konstanty, externí promenné) a **relativní adresou** vzhledem k zacátku této skupiny.

Skupiny objektu se pak **přesouvají na „dobre“ místo**: Globální na jednom předdefinovaném místě, lokální na začátek funkce.

Prirazovací príkazy

Musi pocitat s moznou **konverzi** mezi typy (real -> int, …). Pro přístup k položkám struktury probiha vyuzitim vypočtených posunutí z deklarace, nebo přímý výpočet přes ukazatel a přičtenou konstantu.

Vícerozměrná pole se chápou jako jednorozměrná (stejně jako paměť) – obvykle rozvinutí **řádka za řádkou**. Pro pristup k položkám pole rozlisujeme:

- **Jednorozměrné pole** A[lb..ub] typu o šířce w
  Adresa prvku na indexu [i]: base + (i-lb) * w
- **Dvojrozměrné (vicerozmerne) pole** A[lb_1..ub_1,lb_2..ub_2]
  Adresa prvku na indexu [i, j]: base + ((i_1-lb_1) * (ub_2-lb_2+1) + i_2-lb_2) * w

Booleovské výrazy

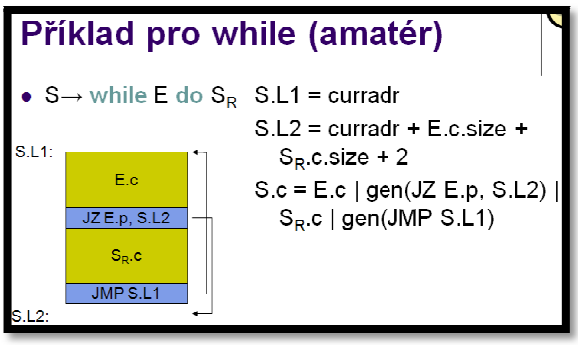
Pro reprezentaci booleovských hodnot a preklad booleovských výrazu se pouzívají dve základní metody. První metoda reprezentuje **logické hodnoty jako císla** a vyhodnocuje booleovské výrazy stejne jako aritmetické. Pro kódování booleovských hodnot se casto pouzívá 0 jako false a 1 nebo nenulová hodnota jako true. Druhou základní metodou je **reprezentace booleovských výrazu tokem rízení** (tj. nasekat do nich instrukce skoku), tj. pozicí dosazenou v programu. Tato metoda je zvládte výhodná pro implementaci rídicích príkazu, neboť umoznuje jejich efektivnejsí vyhodnocování pokud napr. ve výrazu E1 or E2 zjistíme, ze hodnota jednoho operandu je true, nemusíme jiz vyhodnocovat druhý operand (**skracene vyhodnocovani**). Nektere překladače dovedou vybrat vhodnou metodu na základe analýzy kazdého konkrétního výrazu.

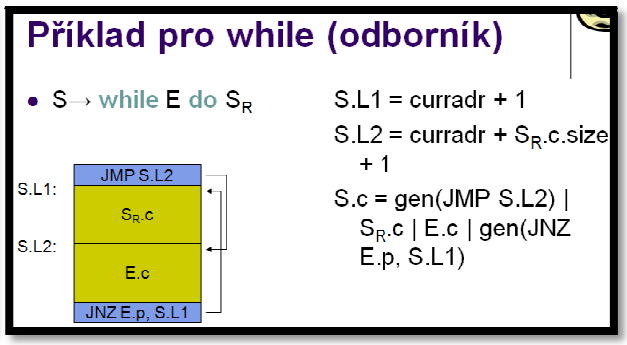
Príkazy pro zmenu toku rízení

If then else

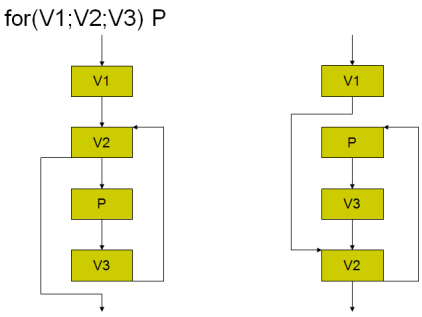
Implementovane pomocou skokov.

While





For



Swtich

Vyhodnotit výraz, podle kterého se větvím. Najít větev podle vypočtené hodnoty. Provést nalezenou větev nebo defaultní větev/nic.

Nalezeni vetve: **Lineární sekvence podmínek** (pouzitelne pro malo vetvi), binární vyhledávání v tabulce **[hodnota, vetev_ptr]**, **binární strom podmínek**, **tabulka ukazatelů indexovaná hodnotou vetve** (vhodne pro velkou hustotu hodnot ve zkoumaném rozsahu).

Backpatching

V predchozích odstavcích jsme si predvedli nekolik ruzných metod generování mezikódu pro rídicí príkazy. Nezabývali jsme se vsak vlastním prirazením adres pro jednotlivá návestí, obcházeli jsme jej pouzitím symbolických jmen. Pokud se preklad provádí jednopruchodove, dostáváme se velmi casto do situace, kdy **neznáme v urcitém bode cílovou adresu návestí**, neboť kód, který bude tímto návestím oznacen, jeste nebyl vygenerován. Tento problém muzeme resit tak, ze necháme adresu návestí prázdnou a udrzujeme seznam adres, ze kterých se na kazdé takové návestí odkazujeme. V okamziku, kdy dosáhneme místa obsahujícího definici návestí, zpetne jeho adresu doplníme do vsech míst uvedených v seznamu. Tato metoda zpetných oprav se nazývá **backpatching**.

Volání procedur/funkcí

Volání podprogramu je obvykle standardizované ve tvaru urcité volací posloupnosti. I kdyz se volací posloupnosti od sebe lisí i pro jednotlivé implementace jednoho programovacího jazyka, jsou obvykle tvoreny následujícími akcemi:

- **Vyhodnocení predavanich argumentu.**
- **Předání hodnot nebo referencí.**
- **Svázání skutečných parametrů s formálními parametry** (**Pozičně** (naskládat ich v spravím poradi na zásobník) nebo **jménem**).
- **Navratova adresa** - místo, na které se vrátí rízení po ukoncení volaného podprogramu se musí ulozit na zname místo.
- **Návratová hodnota** - ulozí se na známé místo.

Mezi úkoly volajícího a volaného podprogramu není zádná presná hranice. Casto se tyto úkoly rozdelují na základe vlastností zdrojového jazyka, cílového pocítace a pozadavku operačního systému.

Rozsahy platnosti promenných, aktivacní záznamy, implementace vnorených procedur, volací konvence

Rozsahy platnosti proměnných

Externí promenné, globalni promenne, lokalni promenne. Proměnná deklarovaná v inicializační části for cyklu se nazývá řídící proměnná cyklu. Rozsah její platnosti je omezen na cyklus. Zivotnost promennych deklarovanych v blocích je omezena na ten blok.

Doba života / rozsah platnosti proměnné (live range) – je množina míst v proceduře, kdy je proměnná zapotřebí. Tedy existuje možnost, že by ještě byla čtena (před zápisem). Zkoumá se obvykle pouze pro skalární lokální proměnné bez aliasu (situace (nebo možnost výskytu situace), kdy k jedné proměnné, její části, či paměťovému místu, vedou dvě různé přistupové cesty). Proměnnou s nesouvislým rozsahem platnosti lze nahradit několika jinými a tím odpadne omezení na shodnou alokaci v jednotlivých souvislých oblastích rozsahu platnosti.

Životnost proměnné

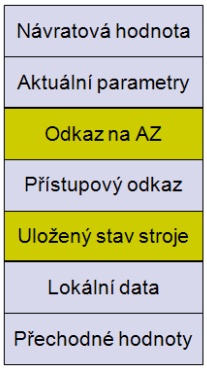
Tříadresová instrukce **x := y op z** ***definuje*** x a ***používá*** y a z. Říkáme, že proměnná je ***živá*** v daném bodě mezikódu, pokud tento bod leží na nějaké cestě začínající bodem, kde byla definována, do nějakého bodu, kde byla použita, přičemž po této cestě nejsou žádné další definice.

Aktivacni zaznamy

Vetsina soucasných procedurálních programovacích jazyku umoznuje vytvárení strukturovaných programu, ve kterých je základním pojmem **podprogram (alias procedura)** jako samostatná programová jednotka, predstavující abstrakci nejaké akce. Podprogramy, které vracejí hodnotu, se nazývají **funkce**.

Kazdé provedení tela podprogramu nazýváme **aktivací podprogramu**. Doba **zivota** aktivace podprogramu p je posloupnost kroku mezi prvním a posledním krokem provádení tela podprogramu, vcetne casu stráveného provádením podprogramu volaných z p, jimi volaných podprogramu atd. Jsou-li a a b aktivace podprogramu, potom jejich doby zivota se bud neprekrývají, nebo jsou do sebe zanorené. To znamená, ze zacne-li b jeste pred ukoncením a, musí rízení opustit b dríve nez a. Podprogram je **rekurzivní**, jestlize jeho nová aktivace muze zacít jeste predtím, nez se ukoncí jeho drívejsí aktivace (tj. jednom okamziku existovat zároven nekolik jejích aktivací).

Pri behu programu má kazdá aktivace podprogramu obvykle k dispozici vlastní oblast pameti pro lokální promenné a dalsí pomocné údaje (obsah registru v okamziku volání, návratová adresa z podprogramu apod.). Tato oblast pameti se nazývá **aktivacní záznam podprogramu**. Aktivacní záznamy mohou mít v prípade, ze zdrojový jazyk neumoznuje rekurzivní volání, pridelenu statickou oblast pameti nebo se mohou uchovávat v zásobníku. Pri volání podprogramu se navrchol rídicího zásobníku ulozí nový aktivacní záznam, který se odstraní pri návratu zpet. Pridelování pameti pro aktivacní záznamy na zásobníku se pouzívá bezne u jazyku, které umoznují rekurzivní volání podprogramu nebo které pouzívají staticky do sebe zanorené podprogramy (vnorené procedury).



- Odkaz na AZ
  - AZ volající funkce
- Přístupový odkaz
  - Odkaz na data sémanticky nadřazených funkcí
- Uložený stav stroje
  - Návratová adresa do kódu
  - Registry

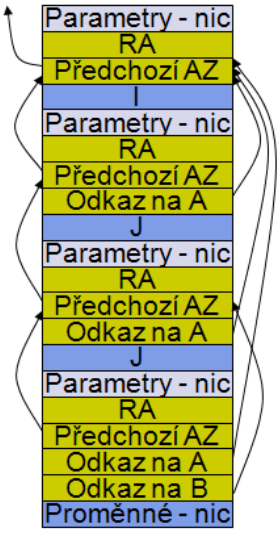
Pri implementaci pridelování pameti na zásobníku bývá jeden registr vyhrazen jako ukazatel na zacátek aktivacního záznamu na vrcholu zásobníku. Vzhledem k tomuto registru se pak pocítají vsechny adresy datových objektu, které jsou umísteny v aktivacním záznamu. Naplnení registru a pridelení nového aktivacního záznamu je soucástí **volací posloupnosti**, obnovení stavu pred voláním se provádí behem **návratové posloupnosti**.

Implementace vnořených procedur

V jazycích jako je Pascal nebo Modula-2 mohou být uvnitr podprogramu deklarovány dalsí podprogramy, které jsou v nich lokální. Kazdý podprogram má prideleno císlo odpovídající jeho (statické) **úrovni zanorení**. Hlavní program má statickou úroven 0, podprogramy v nem deklarované úroven 1 atd. Napríklad vsechny funkce v jazyce C mají statickou úroven 1.

- Pascal
  - Vnořené funkce

```
procedure A;
var I:integer;
   procedure B;
   var J:integer;
       procedure C;
       begin …I+J… end;
   begin
       if I>0 B else C;
   end;
begin B end;
```

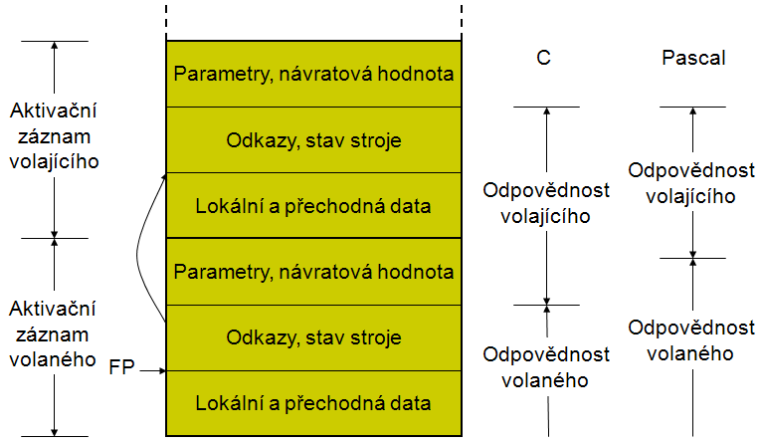


Pro podprogramy, které jsou staticky zanorené do jiných podprogramu, musíme zajistit moznost prístupu k lokálním promenným nadrazených bloku, tj. k jejich aktivacním záznamum. Nejjednodussím resením je rozšíření aktivacního záznamu o ukazatel na aktivacní záznam bezprostredního staticky nadrazeného podprogramu - **prístupový ukazatel**. Odkazuje-li se príkaz v procedure p na statické úrovni n_p na promennou a na statické úrovni n_a, se musí nejprve projít n_p - n_a prístupovými ukazateli, címz získáme adresu aktivacního záznamu obsahujícího promennou a. Tuto adresu pak muzeme jiz prímo pouzít pro zprístupnení promenné a, neboť její

relativní adresa v aktivacním záznamu je známa. Kód pro vytvorení prístupových ukazatelu je soucástí volací posloupnosti podprogramu.

Volaci konvence

Je úmluva o způsobu spolupráce volající a volané procedury. Definována cílovým prostředím nebo autorem překladače. Volací (a návratové) posloupnosti se od sebe v ruzných implementacích lisí. Jejich cinnost bývá rozdelena mezi **volající a volaný program**; obvykle volající program urcí adresu zacátku nového aktivacního záznamu (k tomu potrebuje znát velikost záznamu vlastního), presune do nej predávané argumenty a spustí volaný podprogram zároven s ulozením návratové adresy do urcitého registru nebo na známé místo v pameti. Volaný podprogram nejprve uschová do svého aktivacního záznamu stavovou informaci (obsahy registru, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokracuje zpracováním svého tela. Pri návratu opet volaný podprogram ulozí hodnotu výsledku do registru nebo do pameti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si prevezme návratovou hodnotu a tím je volání podprogramu ukonceno.



Předávaní parametrů

Parametry podprogramu mají obvykle pridelen prostor v aktivacním záznamu. Do tohoto prostoru se pri volání podprogramu umístí skutecné parametry - hodnoty, adresy, prípadne jiné datové struktury zprístupnující predávaný parametr. To, co se konkrétne predává, závisí na typu a pozadovaném zpusobu predávání:

- **Predávání hodnotou** (kopírováním) - hodnota skutecného parametru se zkopíruje do formálního parametru. Do aktivacního záznamu podprogramu se zkopíruje hodnota skutečného parametru a veskeré výpocty uvnitr podprogramu se provádejí s touto kopií.
- **Predávání odkazem** - reprezentují se jako adresa skutecného parametru. Změna takového formálního parametru vede k bezprostrední zmene skutečného parametru. Pri této metode predávání parametru umístí volající do aktivacního záznamu volaného podprogramu pouze adresu predávané l-hodnoty.
- **Predávání jménem** - podle potreby se vyhodnocují pri vsech odkazech. Jejich zpracování je blízké zpracování makrodefinic.

Vliv architektury pocítace na generování kódu a optimalizaci

Architektura procesoru představuje **cílový jazyk** (posloupnost strojových instrukcí s absolutními/relativnimi adresami, assembler). Platí to pro překladače do kódu konkrétního procesoru. **Architektura ovlivňuje celý backend** překladače, zejména generátor kódu a může ovlivnit i tvar a instrukce mezikódu pripadne nasazené druhy optimalizací nad mezikódem. Urcujícími faktory procesu generování jsou specifika výstupního jazyka a tedy specifika a detaily konkrétního počítače (procesoru), na kterém bude prelozený program prováden. Tato specifika muzeme rozclenit do nekolik slozek, které jsou dány architekturou počítače (procesoru) a které se nejvýznamneji podílejí na modelu generátoru cílového programu:

Registry

Je to nejrychlejší „paměť“. Je jich však malý počet (x86 má 7 celočíselných 32-bitových, IA-64 má 128 celočíselných 64-bitových). Jsou z nich různé druhy pro různé typy dat (celočíselné, floatové, adresové, vektorové). A taky se knim různě přistupuje (přímo nebo přez zásobník (FPU)).

Počet registrov procesora teda ovplivňuje generovanie kódu, pretože čím viac registrov, tým je ľahšie optimalizovať výsledný kód pre výkon. Lokálne premenné môžu byť alokované v registroch a nie na zásobníku. Dočasné / priebežné výsledky môžu byť ponechaná v registroch bez zapisovania, resp. čítania do, resp. z pamäte.

Instrukční sada

- **RISC** - Jednoduché instrukce, malý počet (např. bez dělení).
- **CISC** - Složité instrukce, velký repertoár.
- **Ortogonalita** - x86 má neortogonální instrukční sadu.

Inštrukčné sady CISC majú častokrát variabilnú dĺžku inštrukcií a majú aj väčší počet inštrukcií, ktoré môžu byť použité. Každá z nich vyžaduje aj rôzne množstvo času. Inštrukčné sady RISC sú pokusom obmedziť variabilitu. Inštrukcie sú zvyčajne konštantnej dĺžky až na niekoľko výnimiek. Inštrukcie používajú menej kombinácií registrov a operácií s pamäťov. Počet inštrukcií dokončených za časové obdobie, je obvykle celočíselným násobkom hodinového cyklu procesora. Existuje niekoľko môžných spôsobov vykonania určitej úlohy. CISC zvyčajne ponúkajú väčšie možnosti, ako je RISC. Překladače musia teda rozpoznať cenu rôznych inštrukcií a zvoliť ich najlepšiu (najefektívnejšiu) postupnosť.

Pipelining

Zahájení čtení a dekódování další instrukce před dokončením vykonávání předchozí instrukce. Každá fáze a každá instrukce má svoje zpoždění (latency). Problém vzniká při závislosti operandů (RAW).

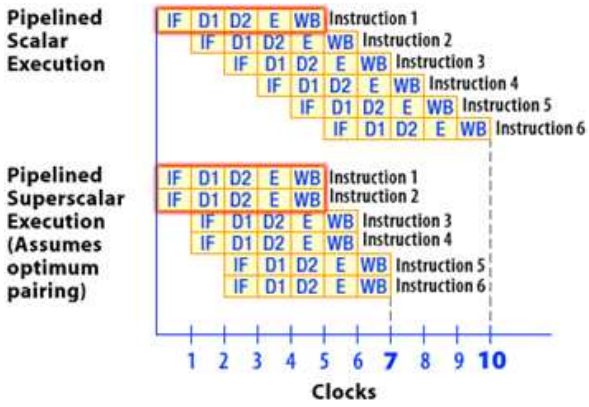


Pipeline je v podstate ALU rozdelené na viac častí. To umožňuje používať jednotlivé časti ALU pre rôzne inštrukcie súčasně, rozdelením inštrukcií na rôzne fázy: dekódovanie inštrukcie, dekódovanie adries, memory fetch, register fetch (načítanie z registrov), execute, uloženie registrov, atď. Kým jedna inštrukcia je vo fáze ukladania do registrov, iná by mohola byť vo fáze register fetch. V pipeline nastane konflikt, keď inštrukcia v jednej etape, závisí od výsledku inej inštrukcie pred ňou v pipeline, ktorá ale nie je ešte dokončená. Pipeline konflikty môžu viesť k jej zdržaniu: CPU potom stráca čas na čakanie vyriešenia konfliktu.¨

Překladač může naplánovat, spřeházet instrukce tak, aby se čekání v pipeline vyskytovano co nejméně.

Superskalarita

Více stejných jednotek schopných paralelního vykonávání instrukcí. Nietoré procesory majú teda niekoľko ALU. To umožňuje vykonávať vícero inštrukcií paralelne. Môžu sa vyskytovať aj obmedzenia ktoré inštrukcie môžu byť vykonávané paralelne v ALU. Majú teda podobné problémy ako pipeliny. Tu musia byť opäť inštrukcie naplánované (zoradené) tak, aby každé ALU bolo vždy naplno vyťažené.



Out-of-Order Execution (vykonávání mimo pořadí)

7. Nactení instrukce, prípadne její rozdrobení na mikroinstrukce
8. Zarazení do vyckávací stanice (instruction pool)
9. Instrukce ceká na všechny svoje operandy
10. Instrukce se vykoná ve své výkonné jednotce (je vybírána z instruction poolu nezávisle na ostatních)
11. Výsledky se uchovají ve fronte (reorder buffer)
12. Až se všechny starší instrukce zapíší do registru, zapíše se výsledek této instrukce (opetovné razení)

Predikce skoku: Hluboké pipeliny mají problém, pokud podmínený skok není proveden (proto je snaha predikovat výsledky skokových instrukcí):

- **Dynamická predicke skoku** - CPU podle historie (vzory nějaké hloubky) odhaduje, jestli skok bude proveden nebo nebude.
- **Statická predikce skoků**
  - **Bez nápovedy** – skok vpred se neprovede, skok vzad se provede.
  - **S nápovedou** – prekladac odhaduje pravdepodobnost skoku.

Spekulativní vykonávaní – je vykonávání kódu, který nemusí být zapotřebí. Je použitné tam, kde je významná disproporce mezi rychlostí CPU a pameti. Typické využití je znacné predsunutí ctecích operací. CPU provádí i odsouvání zápisových operací

Jednočipy (microcontrollers)

Mají malý paměťový prostor. Je tedy nutné vygenerovaný kód optimalizovat na velikost kódu i dat. Mají také oddělené paměťové prostory pro kód a data, ke kterým se různě přistupuje. Mají většinou jednodušší pipeline, žádná superskalarita.

Generování kódu pak pozostává z:

- **Přidělování paměti**
- **Výběr instrukcí**
- **Přidělování registrů**
- **Instrukční scheduling**
- **Generování výsledku** – existuje dohodnutý formát souboru obsahující kód, data, relokace, externí a public symboly, ladící informace.

Přidělování paměti

Kód, statická oblast dat, zásobník (lokální proměnné a parametry funkcí), velikost datových typů, umístění v paměti, zarovnání dat. Co bude v paměti a co v registrech.

Výběr instrukcí

- **1:N** - z jedné instrukce mezikódu se generuje N instrukcí. Je to jednoduché, ale ne příliš dobrý kód.
- **M:N** - z M instrukcí mezikódu N instrukcí kódu. Je to NP-úplný problém výběru. Používají se heuristiky.
- Více cest, jak stejnou věc vygenerovat - CISC, SIMD, VLIW.
- Problémy s neortogonální sadou instrukcí

Přidělování registrů

- Co bude v paměti a co v registrech a v jakých (různé typy registrů).
- Problémy s neortogonální sadou instrukcí.
- Obarvení grafu životnosti proměnných ve funkci.

Instrukční scheduling - přeházení instrukcí, aby zachovávalo sémantiku a nedocházelo k zastavení pipeline:

- Závislosti RAW, WAW, WAR.
- Sestrojení grafu závislosti dat - orientovaný graf, kde hrana říká, která instrukce musí předcházet jinou instrukci, aby data byla k dispozici. Nalezení topologického třídění.
- Obvykle se provádí nad jedním základním blokem.

Metody generování kódu, pridelování registru, optimalizace

Výstupní jazyk generátoru (tato strana asi nie je spravne!!!)

Cílový program vytvárený generátorem cílového programu muze mít obecne tyto formy:

- Posloupnost strojových instrukcí s absolutními adresami - vytvorený program lze okamzite spustit ale není mozné pouzití predem pripravenych knihoven.
- Posloupnost strojových instrukcí s relativními adresami, tj. premístitelný strojový program - výstupem kompilátoru jsou tzv. premístitelné moduly, které je ovsem dále nutno zpracovat sestavovacím programem.
- Posloupnost príkazu jazyka symbolických instrukcí – nasp ASM.

Základní princip klasických metod generovani kodu vychází z predpokladu, ze mnozina jazykových prostredku urcitého programovacího jazyka predstavuje mnozinu nezávislých problému generování. Kazdý takový problém je resen metodou, která je nejvhodnejsí pro urcitou jazykovou konstrukci. Celý generátor je pak implementován jako soubor procedur, z nichz kazdá resí svuj specifický úzký úkol.

Ctverice

Jsou-li napr. vnitrním jazykem prekladace ctverice (n-tice), pak generátor cílového programu interpretuje kazdou ctverici jako volání procedury, operátor jako jméno procedury, operandy jako její parametry. Problém realizace generátoru se tak rozpadá na radu témer triviálních podproblému - implementovat pro kazdý typ ctverice samostatnou proceduru a pridat procedury pro inicializaci, ukoncení, pridelování registru a rízení celého generátoru. Podobne, je-li vnitrním tvarem programu postfixová notace, pak jsou operátory interpretovány opet jako jména procedur. Jejich skutecné parametry jsou vsak zprístupnovány prostrednictvím globálního zásobníku. V prípade, ze je vyvolána procedura odpovídající n-árnímu operátoru, pak pouzívá n vrcholových polozek zásobníku, stejne jako v interpretacním prekladaci. Tato varianta má výhodu v tom, ze výsledná informace muze být ulozena na vrchol zásobníku a pouzita v následujícím kroku (po vyvolání dalsí procedury).

Trojice

Trojice, na rozdíl od ctveric, nespecifikují explicitne výsledek operace. Základem této metody generování je rekurzivní procedura, kterou nazveme KOMP, jejímz úkolem je zpracovat danou trojici, tj. generovat instrukce pro operaci specifikovanou touto trojicí. Výber akcí, které procedura KOMP provede, závisí na operátoru trojice a na umístení jejích operandu. Vzhledem k operandum mohou nastat tri prípady:

- operand je ve stradaci
- operand je reprezentován symbolickou proměnnou
- operand je reprezentován jinou trojicí, presneji ukazatelem na ni

Trojice, která má být zpracována, je urcena parametrem procedury KOMP. Nastáváli prípad (3), pak procedura KOMP volá rekurzivne samu sebe pro specifikovanou trojici.

Architektura back-endu – Back-end přechází postupně těmito fázemi:

- **Vstup** → Sekvenční mezikód střední úrovně
- **Detekce základních bloků** → Částečně sekvenční mezikód (střední úrovně)
- **Analýza aliasů** → Částečně sekvenční mezikód (střední úrovně).
- **Analýza rozsahů platnosti (Live-range analysis)** → Částečně sekvenční mezikód (střední úrovně).
- **Optimalizace** → Částečně sekvenční mezikód (střední úrovně)
- **Výběr instrukcí (Instruction selection)** → Částečně sekvenční mezikód nízké úrovně s virtuálními registry.
- **Optimalizace** → Částečně sekvenční mezikód nízké úrovně s virtuálními registry.
- **Řazení instrukcí (Instruction scheduling)** → Částečně sekvenční mezikód nízké úrovně s virtuálními registry.
- **Alokace registrů** - Částečně sekvenční mezikód nízké úrovně s fyzickými registry.
- **Řazení instrukcí (Instruction scheduling)** → Částečně sekvenční mezikód nízké úrovně s fyzickými registry.
- **Optimalizace** - Částečně sekvenční mezikód nízké úrovně s fyzickými registry
- **Serializace control-flow (Basic-block (re)ordering)** - Sekvenční mezikód nízké úrovně s fyzickými registry
- **Optimalizace** - Sekvenční mezikód nízké úrovně s fyzickými registry
- **Finalizer** - Strojový kód

Detekce základních bloků (v sekvenčním mezikódu)

Základní blok začíná: na začátku procedury, v cíli skoku a za podmíněným skokem.
Základní blok končí: podmíněným skokem, nepodmíněným skokem, návratem z procedury a před cílem skoku.

Určení rozsahů platnosti

Vstup:

- VAR - množina proměnných.
- (BB, CF) – graf control flow. BB – množina základních bloků. $CF \subseteq BB \times BB$ – přechody mezi základními bloky.
- W(b, v) – blok b zapisuje do proměnné v.
- R(b, v) – blok b čte proměnnou v před prvním zápisem do ní.

Výstup: L(b,v) – proměnná v je živá na začátku bloku b.

```
for each b in BB
  L(b,v) = R(b,v);
do {
  for each <b1,b2> in CF
    L(b1,v) |= ~ W(b1,v) & L(b2,v);
} while changed;
```

Algoritmus se provádí vektorově, tj. pro všechny proměnné (v) najednou. Překladač využije SIMD instrukce. Složitost: O(|BB|*|CF|*|VAR|).

Scheduling

Provádí se na jednom základním bloku.

Vstup:

- **Dag** - uzly = instrukce, hrany = závislosti (dependence vzniklé z předávání dat v registrech, dependence a antidependence z přístupů do paměti).
- **Model procesoru** – modeluje se hlavně **latence** – časování závislých dvojic instrukcí a **rezervační tabulky** – schopnosti paralelního zpracování.

Výstup: Přiřazení času každému uzlu dagu čas měřen cykly procesoru. Instrukce trvá několik cyklů – zvolen referenční bod.

Aplikace výstupu schedulingu:

- Běžné procesory (Intel IA-32, včetně x86_64)
  - Seřazení instrukcí podle schedulovaného času.
  - Procesor nemusí dodržet předpokládaný čas.
- Procesory se sekvenčními body (Intel IA-64)
  - V kódu jsou mezi instrukcemi označeny sekvenční body (stops).
  - Procesor má právo přeházet pořadí instrukcí mezi sekvenčními body (ignorují se antidependence i některé dependence).
  - Výstupem scheduleru jsou i sekvenční body.

Pridelovani registru

Efektivní vyuzívání registru v generovaném cílovém programu je obzvláste dulezité, poněvadž instrukce specifikující jako operandy pouze registry jsou rychlejsí a kratsí nez instrukce pracující s operandy v pameti. Efektivní vyuzívání registru vede k minimalizaci presunu údaju mezi pametí a registry instrukcemi typu LOAD, STORE, MOV, coz opet výrazne prispívá k rychlosti generovaného programu. Pri návrhu generátoru se obvykle v souvislosti s problémem vyuzívání registru rozlisují dve cinnosti:

- **pridelování registru** (registr allocation) - rozhodnutí, které objekty programu budou uchovávány trvale nebo prechodne v registrech jako operandy nebo slozky specifikací prístupových cest.
- **prirazování registru** (registr assignement) – Prirazování registru urcuje konkrétní registry, kde budou tyto objekty ulozeny.

Informacní struktura, nad níz se resí pridelování registru, se nazývá **interferencní graf**. Jeho vrcholy tvorí symbolické (abstraktní) registry, odpovídající promenným vnitrního tvaru programu. **Dva vrcholy grafu interferují**, jestlize príslusné promenné jsou soucasne zivé, tj., je-li jedna z nich zivá v oblasti platnosti definice druhé promenné. Interferující vrcholy jsou spojeny hranou. Je zrejmé, ze promenným odpovídajícím takovým vrcholum nemuze být pridelen jediný registr (nemohou ho postupne sdílet), a proto pri barvení grafu musí temto vrcholum príslušet ruzné barvy (v dusledku jejich incidence). Je-li mozné celý interferencní graf obarvit n barvami, kde n je pocet skutecných registru, které jsou k dispozici, pak obarvení grafu dává výsledné pridelení registru. Promenné odpovídající stejne obarveným vrcholum budou pridelovány jedinému (stejnemu) registru.

Algoritmus barvení grafu je zalozen na velmi jednoduchém poznatku: **graf G lze obarvit n barvami tehdy, kdyz lze n barvami obarvit redukovaný graf G', který byl získán z grafu G vypustením vrcholu stupne mensího nez n**. Algoritmus postupne odebírá z interferencního grafu vsechny vrcholy mající stupen mensí nez n. Tento proces, který v kazdém kroku snizuje stupne zbývajících vrcholu grafu, casto vede az k redukci na prázdný graf, coz indikuje, ze výchozí interferencní graf je obarvitelný n barvami. V tomto prípade získáme hledané obarvení zpetným postupem, který vede k danému interferencnímu grafu; pridáváním vrcholù v opacném poradí, nez v jakém byly tyto vrcholy odebírány. S pridáním kazdého vrcholu je nalezena jeho barva, tak aby byla splnena podmínka obarvení (tato barva vzdy existuje, ponevadz stupen každého vrcholu je v okamziku pridání mensí nez n). Uvedený algoritmus má lineární výpoctovou složitost. Pripomenme, ze klasická úloha nalezení chromatického císla grafu (tj. barevnost grafu) je NP-úplný problém.

Implementace

Vstup

- **Částečně sekvenční kód** - možnost přesně definovat rozsahy platnosti.
- **Instrukce pracují s virtuálními registry** - docasne promenne…
- **Optimistický kód** - predpoklada, ze všechny jednoduché proměnné bez aliasu budou (mohou byt) v registrech.

Vystup = Přiřazení virtuálních registrů fyzickým.

Algoritmus

- **Vytvoření matice kolizí** - Kolize = neprázný průnik oblastí platnosti, matice C = VAR × VAR → Bool (true ak doslo ku kolizii).
- **Odhad ceny za spill-kód** - P : VAR → Integer (Odhad počtu použití proměnné v průměrném provedení procedury).
- **Vlastní algoritmus alokace** - Barvení grafu (VAR,C) n barvami.
- **Doplnění spill-kódu**

Barvení grafu (VAR,C) n barvami

Algoritmus spociva v barvení grafu (VAR,C) n barvami - n je počet fyzických registrů. Je to NP-úplná úloha, dobré heuristické řešení. Má-li vrchol méně než n sousedů, lze jej vždy obarvit. Algoritmus ma dve faze:

- **Postupné odebírání vrcholů** - Ve vhodném pořadí tak, aby později měly šanci na obarvení, Odebrané vrcholy se ukládají na zásobník.
- **Zpětná rekonstrukce grafu** - Vrcholy se odebírají ze zásobníku, Pokud je lze obarvit, obarví se a přidají do grafu, Neobarvitelné nebudou přiděleny do registru.

▪ Vstup: graf (VAR,C), ceny P, počet n

```
(VAR2,C2) := (VAR,C)
while not empty(VAR2) do
  if not exists v:VAR2 st deg(v,C2) < n then
    select v st P(v) is minimal; // or deg(v,C2) is maximal
  end if;
  (VAR2,C2) := (VAR2-{v},C2-edges(v))
  stack.push(v);
end while;
while not empty(stack) do
  stack.pop(v)
  M = {u ∈ VAR2; <v,u> ∈ C}
  if exists i ∈ {1..n} st not i ∈ color(M) then
    color(v) := i
    (VAR2,C2) := (VAR2+{v},C2+)
  else
    spill(v) := true
  end if
end while
```

▪ Výstup: přidělení registrů *color*, nepřidělené registry *spill*

Spill-code - kód „navíc“, který musi být přidán pro manipulaci s proměnnými, které se nevešly do registrů. S neobarvenymi proměnnými je třeba s nimi nalozit nasledovne:

- **Alokovat neobarvené uzly na zásobníku**
- **Opatřit každou instrukci spill-kódem**
  - **Prefix:** Přesun neobarvených vstupních operandů do registrů
  - **Suffix:** Přesun neobarvených výstupních operandů do paměti
- Problém: kde vzít registry pro tyto operandy? - Alokovat pomocné registry v rámci normální alokace.

Optimalizace

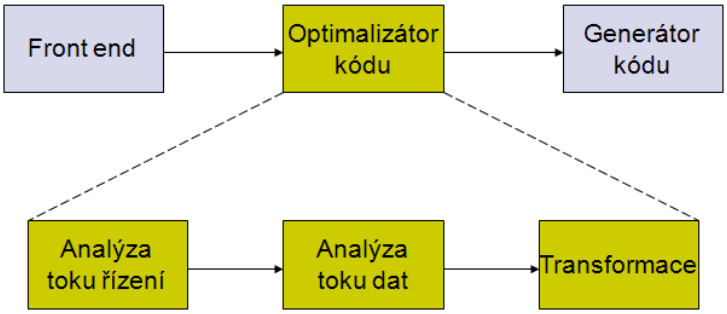
Je transformace programu. Výstup a chování programu nesmí být transformací změněny. Obvykle transformace zvyšující rychlost nebo znizovani velikosti kodu. Transformace musí být hodna vynaložené námaze (složitá/pomalá optimalizace s malým zrychlením není moc užitečná).

Druhy optimalizaci (transformaci)

- **Intraprocedurální** - uvnitř jedné procedury.
  - **V rámci jednoho základního bloku (lokalni)** - nejsou žádné skoky.
  - **V rámci jedné procedury (globální)** - pohyb kódu mezi bloky, vznikají, zanikají základní bloky.
- **Interprocedurální** - v rámci celého programu. Zrychlení volání mezi funkcemi, provádí se ve fázi linkování zpožděným překladem.

Organizace optimalizátoru

Sklada se z **analýzy toku řízení** (probiha konstrukce grafu toku řízení) a z **analýzy toku dat** (pobiha výpočet životnosti proměnných).



Analyza toku rizeni (konstrukce grafu toku řízení)

Graf toku řízení - Je grafová reprezentace tříadresového kódu. Uzly jsou výpočty. Orientované hrany představují tok řízení. Je vhodný pro optimalizace, ale i pro reprezentaci mezikodu.

Základní blok - Souvislá sekvence tříadresového kódu, do které tok řízení vstoupí na začátku a vystoupí na úplném konci a nikde jinde. Algoritmus na rozsekání sekvence tříadresového kódu na základní bloky:

- **Určení množiny vůdců** - začátek sekvence je vůdce. Každá instrukce mezikódu, která je cílem skoku, je taky vůdce. Každá instrukce mezikódu, která následuje těsně po instrukci skoku, je taky vůdce.
- **Urceni základních bloku** - Pro každého vůdce se základní blok sestává z vůdce a všech následujících tříadresových kódu až po dalšího vůdce (bez něj) nebo po konec sekvence.

Konstrukce grafu toku řízení

Uzly jsou základní bloky. Jeden ze základních bloků je **počáteční blok** - jehož vůdce je první instrukcí mezikódu v sekvenci. Mezi dvěma základními bloky B_1 a B_2 je orientovaná hrana z B_1 do B_2, pokud platí jedna z podmínek: poslední příkaz B_1 je skok na vůdce B_2, nebo B_2 následuje ihned za B_1 a B_1 nekončí nepodmíněným skokem.

Životnost proměnné

Tříadresová instrukce **x := y op z *definuje*** x a ***používá*** y a z. Říkáme, že proměnná je ***živá*** v daném bodě mezikódu, pokud tento bod leží na nějaké cestě začínající bodem, kde byla definována, do nějakého bodu, kde byla použita (čtena?), přičemž po této cestě nejsou žádné (její?) další definice (přiřazení?).

Analýza toku dat (výpočet životnosti proměnných)

- **„Obarvení“ uzlů grafu toku řízení** (základních bloků) proměnnými, které jsou v tomto uzlu živé - Nemusí být živé v celém základním bloku. Uvnitř základního bloku se dopočítá přesná životnost.
- **Výpočet pořadí vyhodnocení** - topologické setřídění DAGu, který popisuje definici proměnné jako cíl hran uzlů použitých pro výpočet.
- **Modelování ukazatelů** (Pointer aliasing) - Pokud jsou ve zdrojovém kódu použity ukazatele, je obtížné ho optimalizovat, protože při zápisu do paměti pomocí ukazatele může dojít ke změně libovolné proměnné uložené na adrese zápisu. Analýza ukazatelů se proto snaží určit, které ukazatele a proměnné operují se stejnou pamětí, čehož pak lze využít při další optimalizaci.
- **Modelování volání funkcí** - Interprocedurální optimalizace.

Lokální optimalizace (v rámci jednoho základního bloku)

- **Eliminace společného podvýrazu** - (a+b)*(a+b) - a+b lze pocitat jen jednou.
- **Propagování kopií**
- **Eliminace mrtvého kódu** - x = y+z, a x již není nikde dál použitá.
- **Výpočet konstantních výrazů** - Konstantní výrazy jako (3+5) lze vypočítat již během překladu, nahradit ve zdrojovém kódu jejich výslednou hodnotou.
- **Algebraické transformace** (x = x + 0, x = x * 1 -> lze uplne odstranit)

Globální optimalizace

Více základních bloků, možný pohyb kódu mezi bloky, příp. i změna grafu toku řízení.

- **Eliminace společného podvýrazu**
- **Propagování kopií**
- **Eliminace mrtvého kódu**
- **Optimalizace cyklů** - přesun invariantního kódu, redukce síly operace, odstranění indukční proměnné. Vytvoríme nový základní blok, nazývaný **prolog cyklu**, do nehoz presuneme invariantní operace.

Optimalizace řízená profilem – pozostává ze tří fáz:

- **Instrumentace** - Speciálně přeložený kód s voláním „sběrných“ funkcí na konci a začátku základních bloků.
- **Profilování** - Spouštění instrumentovaného programu s „typickými“ vstupy. Sběr statistických dat o průchodu kódu „sběrnými“ funkcemi.
- **Optimalizace** - Nasbíraná statistická data ovlivní optimalizaci. Využití při optimalizaci v mezikódu i kódu. Změření váhy cest v grafu toku řízení. Přidělování registrů.

Priklady

Lokální eliminace společného podvýrazu

- a = b + c
- b = a – d
- c = b + c
- d = a – d

- a = b + c
- b = a – d
- c = b + c
- d = b

- Je tady schovaný problém
  - Pointer aliasing
  - ap = &a
  - b = a + c
  - *ap = d
  - e = a + c

Optimalizace cyklů – 1

- Přesun invariantního kódu
  - Výpočet, který se v cyklu nemění se předpočítá před cyklem

```
while(i<limit-5) S
```

```
t = limit - 5;
while(i<t) S
```

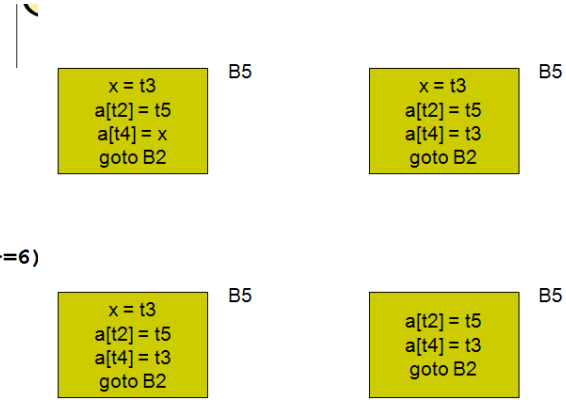
Optimalizace cyklů – 2

- Redukce síly operace
  - Typicky operace násobení převedená na sčítání
- Odstranění indukční proměnné
  - Stačí pouze jediná indukční proměnná cyklu
  - Obvykle se podaří zrušit při redukci

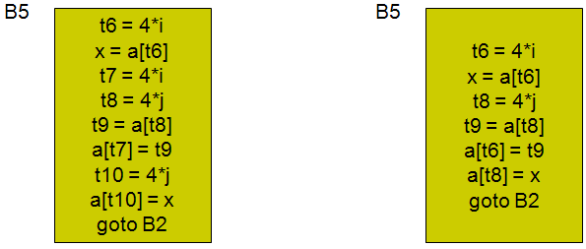
```
for(i=3;i<8;i+=3)
{
  j = i*2;
  a[j] = j;
}
```

```
for(t=6;t<16;t+=6)
  a[t] = t;
```

Příklad – copy propagation a dead-code elimination



Eliminace společného podvyrazu



Podpora kompilátoru pro synchronizacní primitiva, vlákna

Proces, vlakno, vzajemne vylouceni.

- **Proces** – ma adresový prostor, prostředky, práva a signály.
- **Vlakno** – ma programový čítač, registry CPU a zásobník.

Vlákno je běžící program, aktivita uvnitř procesu, akt vykonávání programu. Proces je pasivní obálka, poskytující běžícím vláknu uvnitř zdroje, kód a paměť. Vlákna jsou vykonávané instrukce uvnitř procesu, vláken v jednom procesu může být více, sdílejí paměť a kód, ale mají každé svůj vlastní zásobník.

Vzájemné vyloučení (mutual exclusion) - kritickou operaci (operaci v kriticke sekci) provádí nejvýše jedno vlakno.

Kritická sekce - část programu, kde se provádí kritická operace.

Metody dosažení vzájemného vyloučení

- **Aktivní čekání (busy waiting)** - spotřebovává čas procesoru, vhodnější pro předpokládané krátké doby čekání**,** nespotřebovává prostředky OS**,** rychlejší.
- **Pasivní čekání/blokování** - vlakno je ve stavu blokován, vhodné pro delší doby čekání, spotřebovává prostředky OS, pomalejší.

Synchronizacni primitiva

Semafory

Jsou implementovány OS. Cítač a fronta uspaných procesů. Atomické operace DOWN/UP. Dvě sémantiky chování podle možných hodnot čítače (čítač >= 0 a čítač v rozsahu celých čísel (i záporný)). Binární semafor (pouze hodnoty 0 a 1).

První semantika

- **DOWN** - pokud je čítač > 0, sníží čítač o 1 a pokračuje dál. Pokud je čítač = 0, operace DOWN se zablokuje a proces je přidán do fronty čekající na tomto semaforu
- **UP** - pokud je fronta neprázdná, vybere libovolný proces a ten probudí za DOWN, jinak zvětší čítač o 1.

Druha semnatika

- **DOWN** - vždy se sníží hodnota čítače o 1. Pokud je čítač >= 0, pokračuje dál jinak se zablokuje.
- **UP** - vždy se zvětší hodnota čítače o 1. Pokud je hodnota <= 0 a fronta je neprázdná, vybere libovolný proces a ten odblokuje.

Monitor

Je synchronizační primitivum, které se používá pro řízení přístupu ke sdíleným prostředkům. Jeho zvláštností je, že jde o speciální konstrukci programovacího jazyka (**musí ho tedy implementovat překladač**), typicky implementovanou pomocí jiného synchronizačního primitiva (napr. **semaforem**). Výhodou monitoru oproti jiným primitivům je jeho vysokoúrovňovost – snadněji se používá a je bezpečnější. Při jeho použití je méně pravděpodobné, že programátor udělá chybu. Monitor se skládá z dat (atributy triedy), ke kterým je potřeba řídit přístup, a množiny funkcí (metod), které nad těmito daty operují.

Vzajemne vylouceni pomoci monitoru

Monitor se **podobá třídě** z OOP. Odlišností je to, že **překladač doplní monitor o zámek**, díky němuž se dosáhne *vzájemného vyloučení* – v jednu chvíli může být uvnitř monitoru jen jeden proces.

Když chce proces vstoupit do monitoru (tj. zavolat jeho funkci), musí nejdříve získat zámek. Pokud zámek v tu chvíli drží někdo jiný, tak se proces zablokuje a čeká, dokud se zámek neuvolní (tj. dokud jiný proces neopustí monitor nebo nezačne čekat na **podmíněnou proměnnou**).

Celý proces zamykání je **pro programátora transparentní**. V programu se funkce monitoru volají stejně jako ostatní funkce. Kód, který provádí zamykání a odemykání, **vygeneruje překladač**.

Monitor většinou splňuje **dodatečné podmínky** a pokud jsou tyto podmínky splněny, platí, že žádný proces nenajde data v nekonzistentním stavu, což je přesně důvod pro zavedení synchronizačního primitiva:

- Data monitoru jsou přístupná jen z jeho funkcí.
- Funkce monitoru nepoužívají data mimo monitor.
- Každá funkce zajistí, že před uvolněním zámku jsou data v konzistentním stavu.

Podminene promenne

Občas je potřeba, aby proces, který je právě v monitoru, počkal na nějakou událost. Monitor poskytuje tuto funkcionalitu pomocí tzv. *podmíněných proměnných*.

Když funkce monitoru potřebuje počkat na splnění podmínky, vyvolá **operaci *wait*** na podmíněné proměnné, která je s touto podmínkou svázána. Tato operace proces zablokuje, zámek držený tímto procesem **je uvolněn** a proces je odstraněn ze seznamu běžících procesů a čeká, dokud není podmínka splněna. Jiné procesy zatím **mohou vstoupit** do monitoru (zámek byl uvolněn). Pokud je jiným procesem podmínka splněna, může funkce monitoru „signalizovat“, tj. probudit čekající proces pomocí **operace *notify***.

„Zajímavých událostí“ může být více, každá může být spojená s vlastní podmíněnou proměnnou. Operace *notify* budí jen ty procesy, které provedly *wait* na **stejné proměnné**.

V dřívějších implementacích signalizování způsobilo, že čekající proces se okamžitě rozběhl a získal zámek (signalizující proces se zablokoval, dokud se vzbuzený proces zámku zase nevzdal), čímž bylo zaručeno, že podmínka je stále ještě splněna. Implementace tohoto chování je komplikovaná a má velkou režii. Je také nekompatibilní s plánovači, které mohou proces kdykoliv přeplánovat. Z těchto důvodů existuje několik různých sémantik podmíněných proměnných, co se týče signalizování. Ve většině moderních implementací signalizace neblokuje signalizující proces, pouze způsobí, že proces čekající na podmíněnou proměnou bude čekat na získání zámku (prakticky **je proces přeřazen z jedné fronty do druhé**).

Tento přístup má dva vedlejší efekty. Signalizující proces nemusí před signalizací uvést monitor do konzistentního stavu, protože stále drží zámek. Na druhou stranu nelze zaručit, že důvod signalizace stále platí ve chvíli, kdy se probuzený proces rozběhne. Mezitím totiž mohl jiný proces podmínku opět zneplatnit. Čekání na podmínku tedy musí být nezbytně implementováno pomocí ***dvojí kontroly*** – podmínka se **testuje před i po volání *wait*** a pokud není splněna, opět se volá *wait*.

Většina implementací také poskytuje **operaci *notifyAll***, která probudí všechny procesy čekající na danou podmíněnou proměnnou.

Podpora v programovacích jazycích

V některých objektově-orientovaných programovacích jazycích jsou monitory hlavním synchronizačním primitivem kvůli své vysokoúrovňovosti: objekty mohou být přímo svázány s monitory, každý objekt automaticky vlastní svůj unikátní monitor. **Java** (synchronized), **.NET** (lock).

Volatile

Je rezervované slovo programovacího jazyka C. Pokud je nějaká proměnná označena slovem volatile, nebude kompilátor žádným způsobem optimalizovat její užití.

```
volatile int i;
i = 0;
while (i == 0) ; /* smyčka bude ukončena nějakou vnější událostí, která změní hodnotu i */
```

Deklarace volatile (nestálý) znamená, že k obsahu proměnné přistupuje ještě nějaký jiný proces než ten, který je řízen aktuálním zdrojovým kódem. Může to být například souběžné běžící vlákno jiného procesu nebo hardwarové přerušení. Že je proměnná označena jako volatile, ještě neznamená, že operace s ní jsou atomické.

Objektove orientované jazyky a principy jejich implementace

Definice: Na **objektove-orientované programování** se dá nahlédnout jako na kolekci spolupracujících objektu – v protikladu k tradicnímu pohledu, kdy se za program považuje sled instrukcí pro pocítac. V OOP je každý objekt schopný prijímat zprávy, zpracovávat data a posílat zprávy jiným objektum. Na každý objekt se tak dá nahlížet jako na nezávislý „malý stroj“ s vlastní rolí a zodpovedností. Zjednodušene receno jde o dotažení **konceptu data + algoritmy = program**. Data tvorí s kódem, který je spravuje, jeden celek.

Trida – Predstavuje abstrakci prvku modelované reality (jak data, tak související funkčnost) jsou v jazyku seskupeny do entit, nazývaných tridy. Jsou zobecněním pojmu struktura. Obsahuje deklarace datových položek, funkce (metody), virtuální funkce a statické funkce a definice výčtových konstant a typů (včetně vnořených tříd).

Objekt – Běhová reprezentace jednoho exempláře třídy. Reprezentace objektu v paměti obsahuje **datové položky a skryté pomocné položky** umožňující funkci virtuálních metod, výjimek, RTTI, virtuální dědičnosti.

Dědičnost – Třída (struktura) může mít jednoho (nebo více) předků. Relace předek-potomek je **orientovaný acyklický graf**. Je možné napsat kód, který pracuje s daty (a dalšími prvky) společného předka několika tříd bez závislosti na tom, jaký je typ celého objektu (tento kód lze přeložit bez znalosti potomků).

Reusabilita - potomci třídy mají automaticky její datové položky a další vlastnosti a schopnosti. *Inherit to be reused, not to reuse.*

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

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

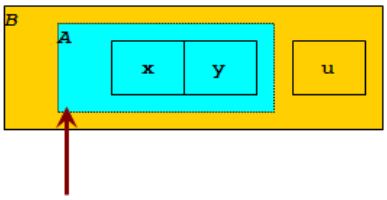
Implementace objektů

Z hlediska jazyka není velký rozdíl mezi složenými datovými typy a trídami. Deklarace trídy obsahuje, stejne jako u složeného datového typu, datové položky. Navíc ale obsahuje i deklarace funkcí (metod), které s nimi pracují. Nekteré funkce mohou mít speciální vlastnosti – statické, virtuální, konstruktory, destruktory. Navíc vetšina jazyku pridává možnost oznacení kterýchkoliv položek jako verejné nebo privátní. Trídy mohou nekdy (C++, Java) obsahovat i vnorené datové typy (výcty, …) a dokonce vnorené trídy.

Za behu je jedna instance trídy – objekt reprezentována v pameti pomocí: datových položek (stejne jako složený datový typ), skrytých pomocné položky umožnujících funkci virtuálních metod, výjimek, RTTI a dedicnosti (identifikace typu / jeho velikosti apod.)

Implementace dědičnosti v C++

Je-li třída B (přímým či nepřímým) potomkem třídy A, pak paměťová reprezentace objektu typu B obsahuje část se shodným tvarem jako reprezentace samostatného objektu typu A.



Z každého ukazatele na typ B je možno odvodit ukazatel na část typu A a tato konverze je implicitní, tj. není třeba ji explicitně uvádět ve zdrojovém kódu (konverze je jednoduchym prictenim posunuti).

Z ukazatele na typ A je možno odvodit ukazatel na typ B když konkrétní objekt, na který ukazuje ukazatel na typ A, je typu B. Tuto konverzi je nutno explicitne uvadet pretypovanim a zodpovědnost za ověření skutečného typu objektu má programátor. Konverze může vyžadovat odečtení posunutí.

Implementace virtualni dědičnosti v C++

Riesi **diamond inheritance problem** (nasobna dedicnost z rovnakeho typu). Každý objekt s virtualnym dedenim ziska iba ukazatel na predka (neni nakopirovany primo v nem) a při nasobne dědičnosti budou oba objekty ukazovat na toho sameho společného predka.

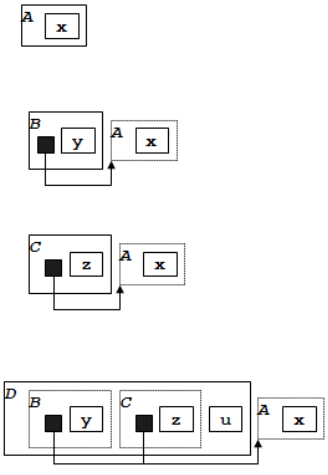
```
class A
{ int x;
};

class B : virtual public A
{ int y;
};

class C : virtual public A
{ int z;
}

class D : public B, public C
{ int u;
}

D * dp;
A * ap = dp;  // OK
// dp = (D*)ap; chyba: nelze najít
```



Implementace virtualnich metod v C++

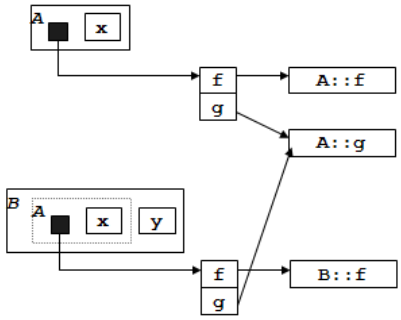
Každý objekt obsahuje ukazatel na **tabulku virtuálních funkcí** (tabulka ukazatelů na funkce). Při volání metody se volá tělo funkce z této tabulky. Při pristupu přes

ukazatel (adresu) se uplatnuje tzv. **pozdni vazba (late binding)** – volanie virtualnej metody potomka. Konstruktor objektu vyplnuje skryte položky třídy, zajišťujících funkci virtuálních metod. Toto vyplňování se děje až po konstrukci součástí třídy.

```
class A {
public:
  int x;
  virtual int f() { return 1; }
  virtual int g() { return 2; }
};

class B : public A {
public:
  int y;
  virtual int f() { return 9; }
};

A* paa = new A;
A* pab = new B;
B* pbb = new B;
paa->f(); paa->g();   // 1 2
pab->f(); pab->g();   // 9 2
pbb->f(); pbb->g();   // 9 2
```



Prekladace vs. interpretry, skriptovací jazyky

Interpretované jazyky

Zdrojový jazyk se nepřekládá do kódu skutečného procesoru, ale do kódu nějakého **abstraktního stroje**. **Interpret** je přeložený do kódu skutečného stroje a simuluje zvolený abstraktní stroj. Pokud je podporovana dynamicka pamet, pak vyhradne s **garbage collectorom** (program obvykle nepotřebuje tolik paměti, aby GC vůbec musel zasahovat). Ukazatele jsou tedy pod kontrolou a prinasi to teda snadnější programování. Duvodu pro pouzivani interpretovaných jazyku a jejich abstraktnich stroju je několik:

- **Není prostor pro překladač**
  - 8-bitové počítače a Basic.
- **Přenositelnost**
  - Stejný abstraktní stroj může běžet na různých OS i různých architekturách CPU.
  - AS/400, Java.
- **Bezpečnost**
  - Větší kontrola nad vykonávanými instrukcemi.

Tento pristup však prinasi i několik problemu:

- **Problém s rychlostí**
  - Dá se řešit pomocí JIT (Just-In-Time compilation) - Pokud interpret narazí na kód abstraktního stroje, který ještě není přeložen, okamžitě ho přeloží na kód cílového stroje a uloží si ho vedle do své cache.
- **Problémy s přenositelností**
  - Nevhodné změny v chování abstraktního stroje mohou přivodit problémy s přenositelností (Java).
- **Jak zvolit abstraktní stroj**
  - Aby pokryl chování všech zdrojových jazyků (.NET).

Skriptovaci jazyky

Skript je väčšinou počas behu interpretovaný interpretrom. Skripty spúšťa interpreter, teda aplikácia, ktorá tento skript vykonáva priamo v jeho pôvodnej podobe - podobe zdrojového kódu. Interpreter prechádza zdrojový kód a vykonáva postupne príkaz za príkazom bez potreby kompilácie. Pôvodne navrhnutý na uľahčenie (zautomatizovanie) operácií v počítači. Sú pomalšie, ale nemajú tak veľké formálne požiadavky (nie je potreba inicializovať premennú, jej dátový typ sa môže počas behu meniť, ukazovatele sú zbytočné). Preto ak sa vyskytuje chyba v skripte, odhali sa az za behu (**dynamic type checking**) a oznami sa vinimkou. Skriptovacie jazyky sa dnes najčastejšie využívajú v prostredí webových aplikácií. Perl, PHP, Python.

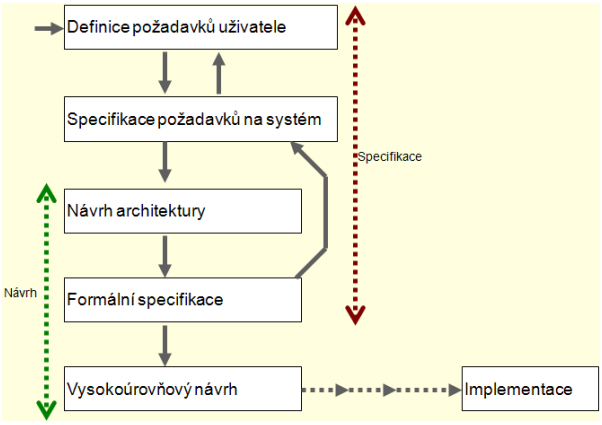
Vyhody: Skript je možné kedykoľvek spustiť bez nutnosti ho vždy znovu skompilovať.

Analýza a návrh softwarových systému

Uvod

- **Specifikace** - přesný popis požadavků na produkt resp. software (funkčnost, efektivita, způsob implementace). Zapsána v přirozeném nebo umělém jazyce.
- **Jazyk (notace) specifikace** – může být **formální**: formální syntaxe i sémantika. Nebo **semi-formální**: formální syntaxe.
- **Specifikační metoda** - posloupnost kroků vedoucích k vytvoření specifikace (jazyk + návod k použití).

Vývoj software – z pohledu pohled teoretika softwarového inženýrství:



Specifikace – je "**smlouva**":

- **Mezi zákazníkem a autorem specifikace**. Zákazník musí specifikaci porozumět. Vyžaduje to tedy alespoň pasivní znalost specifikačního jazyka nebo přepis do přirozeného jazyka, nebo simulovatelná ("executable") specifikace.
- **Implementátorem a mezi autorem specifikace**.
    - Implementátor musí porozumět specifikaci (formální popis + komentáře v přirozeném jazyce).
    - Musí být možné ověřit, zda implementace splňuje specifikaci. Matematický důkaz (správnosti specifikace) nebo generátor testovacích dat (nad kterými se výsledek ověří).
    - Specifikace musí umožňovat různé implementace - neurčuje algoritmus výpočtu, ale požadavky na výsledek. Nemusí být deterministická.

Formální metody - používají výrazně méně, než se před lety předpokládalo.Důvody:

- **Způsob vývoje software se změnil** - Jiné metody vývoje se vyvíjely rychleji.
- **Změnilo se obchodní prostředí** - důraz je u vývoje kladen na time-to-market.
- **Změnil se charakter software** - jsou aplikovatelné pouze na část software.
- **Software se zvětšil** - formální metody nezvládají rozsáhlejší systémy.

Algebraické specifikace, formální popis datových struktur

Algebraicke metody specifikace

Pouzivaji se pro specifikaci rozhraní. Popisovaný objekt je černá skříňka s definovaným vnějším chováním. Jazyky algebraickych metod jsou založené na vícesortových algebrách.

Priklad (zasobnik)

---------- SIGNATURA ----------

Sorts:

V (vrchol, prvok), S (stack - zasobnik)

Operations:

push: S x V → S
pop: S → S
top: S → V
empty: → S

---------- AXIOMY ----------

Equations (axiomy):

pop(push(x,y)) == x
top(push(x,y)) == y

Priklad (mnozina INTů)

```
sorts: SET, INT, BOOLEAN          | vsechny vyuzivane typy
operations:                       | popisuji syntaxi daneho ADT
  - empty:            --> SET     | generator
  - insert: SET x INT --> SET     | generator
  x delete: SET x INT --> SET
  x member: SET x INT --> BOOLEAN
axioms:                           | popisuji semantiku specifikovanych operaci
  member(empty(),j) = false       | axiomy pro vsechny co nejsou generator se vsema moznyma ..
  member(insert(S,j),k) =         | generatorama na vstupu
    if (j=k) then true else member(S,k)
  delete(empty(),j) = empty()
  delete(insert(S,j),k) =
    if (j=k) then delete(S,j)
             else insert(delete(S,k),j)
```

member(p, insert(q, s)) = member(p, s) or p=q
member(p, empty) = false

Modelove orientované metody: Z, VDM

Modelove orientovane metody

Pouzivaji se pro behaviorální specifikace. Vyvíjený software se má chovat stejně jako model. Jazyky modelove orientovaných metod kombinují teorii množin a programovací jazyky.

Z

Formální specifikace je matematický popis (informačního) systému, který přesně popisuje, **co** má systém dělat, **ale neříká jak**. Jazyk Z používá "matematické" ***datové typy*** popsané pomocí ***predikátové logiky*** (nezávislé na počítačové reprezentaci). Dekomponuje specifikaci do malých částí zvaných ***schémata***, ty popisují statické i dynamické aspekty systému. Schéma může popisovat i transformaci jednoho pohledu na systém na jiný, který přidává více detailů (při korektní implementaci původní abstrakce). Postupným zjemňováním (***refinement***) specifikace lze dospět až ke konkrétnímu programu.

Statické aspekty

- stavy
- invarianty

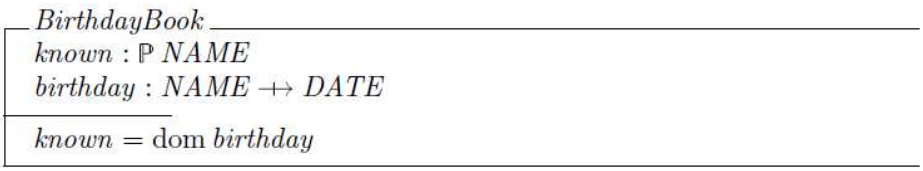
Dynamické aspekty

- možné operace
- vztahy mezi vstupy a výstupy
- změny stavů

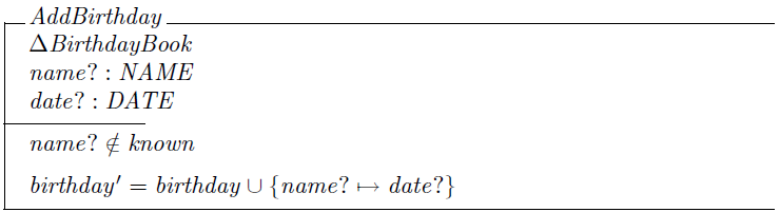
Schéma - každé schéma má svůj název a obsahuje: **deklarace proměnných** a **vztahy mězi proměnnými** (odděleno vodorovnou čárou).

Priklad: BirthdayBook

Základní typy pro účely příkladu: [NAME, DATE]. *Known* je mnozina jmen, a *birthday* je funkce prirazujici k jmenu datum. Schéma popisující "prostor stavů" (***state space***):



Nasledujici schéma popisuje operaci *pridani jmena* do BirthdayBooku. **Δ** znaci, ze operace bude menit BirthdayBook, **?** znaci ze parameter je vstupnim parametrem pro operaci, **‘ (apostrof)** oznacuje novou hodnotu:



- operace zachovává invarianty předchozího schématu.
- podobně jako **?** označuje vstupní proměnné, **!** označuje výstupní.
- podobně jako **Δ** značí operaci modifikující stav, **Ξ (Xí)** značí operaci zachovávající stav.

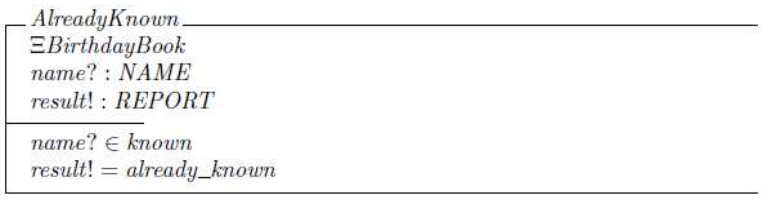


Skládání schémat - Schémata lze skládat pomocí **operátoru =^** (rovnítko se stříškou (ale nad nim)) a **logických operátorů** (∧ a ∨).

Pro ucely prikladu zavedeme novy enum *REPORT* ::= *ok* | *already_known* | *not_known.* A speciální schema *Success* popisující uspech:



Schema *AlreadyKnown* popisující neuspech (v pripade ak tam také jmen uz je):



Složené schéma (robustni pridavani jmen s datumy – ošetřuje pripadny neuspech):

RAddBirthday =^ (AddBirthday ∧ Success) ∨ AlreadyKnown;

- složené schéma RAddBrithday vrací ok v případě úspěchu AddBirthday a already_known jinak
- *RAddBrithday* by šlo nadefinovat přímo, ale jeho definice vztahů by byly nepřehledně složité

Další syntax (např.)

- ∀ z : N • x ≤ z (znamená ze x je menší než všechna přirozená čísla (neboli x je nula))
- *birthday'* = *birthday* ⊕ {*name?* → *date?*} (předefinovává funkci *birthday* v bodě *name?* na novou hodnotu (za opertárotrem ⊕ může být i víceprvková množina))

Z – výrazy

(výraz, výraz,..., výraz)
{výraz, výraz,..., výraz}
{ deklarace | predikát ● výraz }
P výraz
výraz × výraz
(λ deklarace | predikát ● výraz)
(μ deklarace | predikát ● výraz)
(**let** identifikátor == výraz ● výraz)
výraz(výraz,...,výraz)
výraz operátor výraz
výraz . identifikátor
Θ identifikátor-schematu dekorace
if predikát **then** výraz **else** výraz

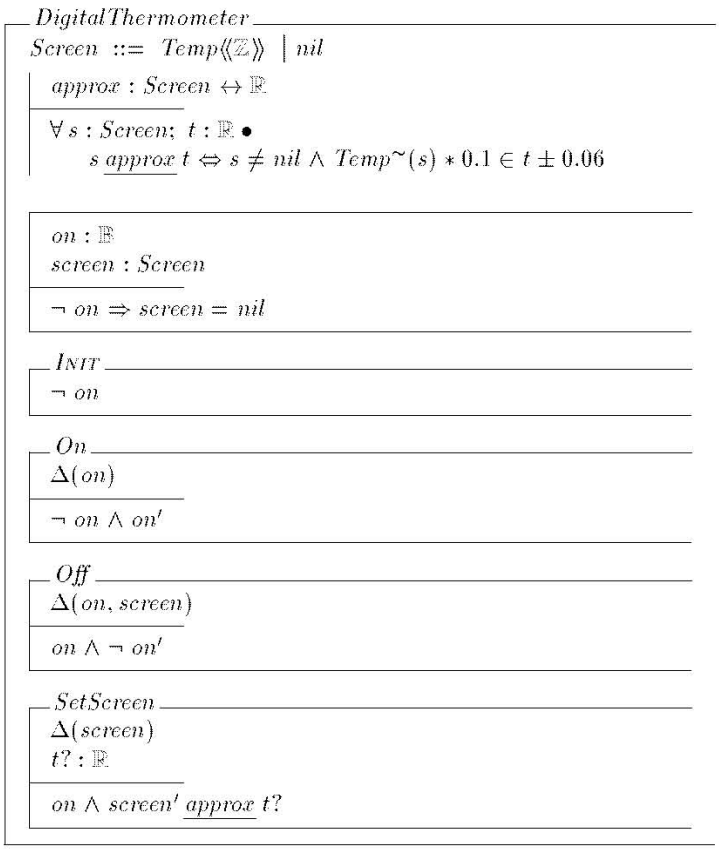
Z – predikáty

true, **false**
výraz = výraz
výraz ∈ výraz
logické spojky
(∀ deklarace | predikát ● predikát)
(∃ deklarace | predikát ● predikát)
($\exists_1$ deklarace | predikát ● predikát)
(**let** identifikátor == výraz ● predikát)
schema-reference

Z - kalkulus schémat

identifikátor-schematu
¬schema-výraz
schema-výraz ∧ schema-výraz
schema-výraz ∨ schema-výraz
schema-výraz ⇒ schema-výraz
schema-výraz ⇔ schema-výraz
(∀ deklarace | predikát ● schema-výraz)
(∃ deklarace | predikát ● schema-výraz)
($\exists_1$ deklarace | predikát ● schema-výraz)
schema-výraz \ (identifikátor,...,identifikátor)
schema-výraz ⌈ schema-výraz

Object-Z


DigitalThermometer
Screen ::= Temp⟨⟨ℤ⟩⟩ | nil
approx : Screen ↔ ℝ
∀ s : Screen; t : ℝ •
s approx t ⇔ s ≠ nil ∧ Temp~(s) * 0.1 ∈ t ± 0.06
on : 𝔹
screen : Screen
¬ on ⇒ screen = nil
INIT
¬ on
On
Δ(on)
¬ on ∧ on'
Off
Δ(on, screen)
on ∧ ¬ on'
SetScreen
Δ(screen)
t? : ℝ
on ∧ screen' approx t?

VDM (Vienna Development Method)

Je skupina technik založená na specifikačním jazyce (VDM-SL). Má variatnu VDM++, která podporuje objektově orientované a "concurrent" systémy. Umožňuje modelování na vysoké úrovni abstrakce a převod na implementaci pomocí zjemění specifikace (refinement). Má spustitelnou podmnožinu, kterou je možné testovat. Vydáno jako ISO standard 1996 (ten definuje ASCII syntax pro věci, co jsou jinak popsány "matematickou" syntaxí).

Basic Types		
bool	Boolean datatype	false, true
nat	natural numbers (including zero)	0, 1, 2, 3, ...
nat1	natural numbers (excluding zero)	1, 2, 3, 4, ...
int	integers	..., -3, -2, -1, 0, 1, 2, 3, ...
rat	rational numbers	a/b, where a and b are integers, b is not 0
real	real numbers	...
char	characters	A, B, C, ...
token	structureless tokens	...
<A>	the quote type containing the value <A>	...

Basic Type Constructors	
T1 | T2 | ... | Tn	Union of types T1,...,Tn
T1*T2*...*Tn	Cartesian product of types T1,...,Tn
T :: f1:T1 ... fn:Tn	Composite (Record) type

Main Operators on Sets (s, s1, s2 are sets)	
{a, b, c}	Set enumeration: the set of elements a, b and c
{ x | x:T & P(x)}	Set comprehension: the set of x from type T such that P(x)
{i, ..., j}	The set of integers in the range i to j
e in set s	e is an element of set s
e not in set s	e is not an element of set s
s1 union s2	Union of sets s1 and s2
s1 inter s2	Intersection of sets s1 and s2
s1 \ s2	Set difference of sets s1 and s2
dunion s	Distributed union of set of sets s
s1 psubset s2	s1 is a (proper) subset of s2
s1 subset s2	s1 is a (weak) subset of s2
card s	The cardinality of set s

Main Operators on Sequences (s, s1,s2 are sequences)	
[a, b, c]	Sequence enumeration: the sequence of elements a, b and c
[f(x) | x:T & P(x)]	Sequence comprehension: sequence of expressions f(x) for each x of (numeric) type T such that P(x) holds (x values taken in numeric order)
hd s	The head (first element) of s
tl s	The tail (remaining sequence after head is removed) of s
len s	The length of s
elems s	The set of elements of s
s(i)	The i^{th} element of s
s1^s2	the sequence formed by concatenating sequences s1 and s2

Main Operators on Mappings	
{a | -> r, b | -> s}	Mapping enumeration: a maps to r, b maps to s
{x | -> f(x) | x:T & P(x)}	Mapping comprehension: x maps to f(x) for all x for type T such that P(x)
dom m	The domain of m
rng m	The range of m
m(x)	m applied to x
m1 munion m2	Union of mappings m1 and m2 (m1, m2 must be consistent where they overlap)
m1 ++ m2	m1 overwritten by m2

The *max* function

This is an example of an implicit function definition. The **postcondition** characterizes the result rather than defining an algorithm for obtaining it. The function returns the element from a set of positive integers:

```
max(s : set of nat) r : nat
pre true
post r in set s and
    forall r' in set s & r' <= r
```

Analýza algoritmu: Hoareova metoda, dynamická logika, temporální logika

Validace - "Are we building the right product?". Nebo jinými slovy - jde o kontrolu, zda-li daný produkt odpovídá reálným požadavkům.

Verifikace - "Are we building the product right?" Neboli - jde o kontrolu, zda-li daný produkt odpovídá výchozí specifikaci.

Hoareova logika

Cílem je, aby se dala formálně dokazovat korektnost programů pomocí rigorózních prostředků matematické logiky. Základem je Hoare triple, popisující jak kousek kódu změnil stav výpočtu - $\{P\}C\{Q\}$; kde **P a Q jsou assertions** a **C je příkaz**. P nazýváme **precondition**, Q **postcondition**. Obojí jsou formule predikátové logiky.

Hoaerova logika obsahuje **axiomy** a **odvozovací pravidla** pro všechny konstrukty jednoduchého imperativního programovacího jazyka. Standardní Hoaerova logika poskytuje pouze **partial correctness**, neboli říká, že pokud před provedením C platí P, pak po jeho provedení platí Q, **nebo** C neskončí. Existuje ale rozšíření poskytující **total correctness** tj. je mozne dokazat, ze program aj skonci.

Axiomy

Empty statement axiom schema

$$\frac{}{\{P\}\ \mathbf{pass}\ \{P\}}$$

Assignment axiom schema

$$\frac{}{\{P[x/E]\}\ x := E\ \{P\}}$$

Pravidla

Rule of composition

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Conditional rule

$$\frac{\{B \wedge P\}\ S\ \{Q\}\ ,\ \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{endif}\ \{Q\}}$$

While rule

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{done}\ \{\neg B \wedge P\}}$$

Consequence rule:

$$\frac{P' \rightarrow P \ , \ \{P\} \ S \ \{Q\} \ , \ Q \rightarrow Q'}{\{P'\} \ S \ \{Q'\}}$$

While rule for total correctness (pre ucely dokazu ukoncenia programu):

$$\frac{\{P \wedge B \wedge t = z\} \ S \ \{P \wedge t < z\} \ , \ P \rightarrow t \geq 0}{\{P\} \ \mathbf{while} \ B \ \mathbf{do} \ S \ \mathbf{done} \ \{\neg B \wedge P\}}$$

Dynamická logika

Navrhl Vaughan Pratt v roce 1974. Je **rozšířením modální logiky** zaměřené na dedukci o počítačových programech (ovšem později se začla využívat i jinde). Modální logika je charakteristická svými dvěma **modálními operátory**:

$\Box p$ - říká že **p platí vždy**

$\Diamond p$ - říká že **p může platit**

Platí pro ně $\Box p \leftrightarrow \neg\Diamond\neg p$ a $\Diamond p \leftrightarrow \neg\Box\neg p$.

Dynamická logika přidává ke dvou termům logik prvního řádu (tvrzení a data) ještě třetí typ termu – **akci.** Dynamická logika toto rozšiřuje přidáním dvou modálních operátorů pro akce:

$[a]$, kde $[a]p$ znamená, že **po provedení akce a musí p platit**

$\langle a \rangle$, kde $\langle a \rangle p$ znamená, že **po provedení akce a může p platit**

Opět samozřejmě platí $\langle a \rangle p \leftrightarrow \neg[a]\neg p$ a $[a]p \leftrightarrow \neg\langle a \rangle \neg p$.

Dynamická logika umožňuje **skládání akcí** a pro jejich zápis se dá využít notace **regulárních výrazů**:

$a|b$ **provede se a nebo b**

$a;b$ **provede se nejprve a, pak b**

$a*$ **a se provede 0 nebo vícekrát**

Dále jsou k dispozici **konstantní akce**:

0 **nic neudělá a neskončí (aka BLOCK)**

1 **nic neudělá a skončí (aka NOP)**

Axiomy

A1. $[0]p$

A2. $[1]p \leftrightarrow p$

A3. $[a|b]p \leftrightarrow [a]p \wedge [b]p$

A4. $[a;b]p \leftrightarrow [a][b]p$

A5. $[a*]p \leftrightarrow p \wedge [a][a*]p$

A6. $p \wedge [a*](p \rightarrow [a]p) \rightarrow [a*]p$ (s akcí n:=n+1 odpovídá matematické indukci)

Speciální axiomy

A7. axiom schema $[x := e]\Phi(x) \leftrightarrow \Phi(e)$, odpovídá přiřazení z Hoareovy logiky. Tedy Φ(*e*) odpovídá fci ve které jsou všechny výskyty x nahrazeny výrazem e.

A8. $[p?]q \leftrightarrow p \rightarrow q$, kde p? je speciální akce definována ke každému tvrzení taková, že p? odpovídá NOP pokud je p true, jinak je to BLOCK. If p then a else b se dá tedy zapsat jako $(p?;a)|(\neg p?;b)$.

Stejně jako Hoareove logike se v ní nedají elegantně vyjádřit konkurentní chování (zatímco sekvenční se zapisuje hezky). To ovšem jde v temporalni logike.

Temporální logika

Přináší prvek času, pravdivost tvrzení se tedy může v závislosti na čase měnit. Například "mám hlad" je pravdivé jenom někdy, díky temporální logice můžeme být proto přesnější a říct "budu mít hlad dokud se nenajím". Přínos pro formální verifikaci je zřejmý - snadno můžeme zapsat tvrzení typu "*kdykoli* příjde požadavek, zařízení *může být* zpřístupněno, ale *v žádném případě* nebude současně zpřístupněno dvěma žadatelům".

Je několik variací - **linear temporal logic** (AKA LTL, existuje jen jedna časová linie) a **branching logic** (více možných časových linií = nedeterminismus, zřejmě má stejnou sílu jako lineární).

Temporální logika zavadi dva druhy operátorů: **logical operators** (klasika: $\neg, \vee, \wedge, \rightarrow$) a **modal operators** (operátory, které říkají například že fce musí někdy v budoucnu platit, nebo že musí platit odteď nafurt).

Modální operátory v Linearni temporalni logike

Textově	**Symbolicky**	**Význam**
Unární operátory:		
N φ	$\circ\phi$	**N**ext: φ musí platit v příštím stavu. (někdy se používá **X**)
G φ	$\Box\phi$	**G**lobally: φ musí platit od teď pořád.
F φ	$\Diamond\phi$	**F**inally: φ musí platit někdy v budoucnu.
Binární operátory:		
ψ **U** φ	$\psi\mathcal{U}\phi$	**U**ntil: φ platí v teď nebo v budoucnu a ψ musí platit až do té doby. Pak už ψ platit nemusí.
ψ **R** φ	$\psi\mathcal{R}\phi$	**R**elease: ψ uvolňuje φ jestliže φ platí do první pozice ve které ψ platí (nebo navždy, pokud taková pozice neexistuje).

Dalo by se zredukovat na dva z těchto operátorů, protože vždy platí:

- **F** φ = **true U** φ
- **G** φ = **false R** φ = ¬**F** ¬φ
- ψ **R** φ = ¬(¬ψ **U** ¬φ)

Petriho síte

Jsou matematickou reprezentaci diskrétních distribuovaných systémů. Vynalezl je Carl Adam Petri ve své disertační práci práci v roce 1962. **Orientované bipartitní grafy** ze dvěma druhy uzlů - **místa** a **přechody**.

Petriho sítě obsahují **místa**, **přechody** a **hrany**. Hrany jsou pouze mezi místy a přechody, nikoliv mezi dvěma místy nebo dvěma přechody. Místa, ze kterých vedou hrany do přechodu jsou nazývána *vstupní místa* tohoto přechodu; místa, do kterých vedou hrany z přechodu jsou nazývány *výstupní místa* tohoto přechodu. Místa mohou obsahovat libovolný počet **teček** (někdy též *značek* nebo *tokenů*; anglicky: *tokens*). Rozložení značek mezi místy v síti je nazýváno **označkování** (anglicky: *marking*). Přechody mohou tečky ze vstupních míst takzvaně **odpalovat** (anglicky: *firing*) do míst výstupních. Přechod je **uschopněn** (anglicky: *enabled*) a může odpálit, pokud je v každém ze vstupních míst alespoň tečka. Když přechod odpálí, odebere tečky z jeho vstupních míst, provede nějaké výpočetní úlohy, a vloží zvolený počet teček do každého výstupního místa. Tento proces činí automaticky v každém jednotlivém kroku.

Výpočet Petriho sítě je **nedeterministický**. Což znamená následující:

1. více přechodů může být uschopněno současně a libovolný může odpálit,
2. žádný přechod není nutno odpálit – odpalují se dle libosti mezi časy 0 až nekonečno nebo vůbec nikdy. Je tedy možné, že se neodpálí vůbec nic.
3. Protože je odpalování nedeterministické, Petriho sítě jsou vhodné pro modelování souběžného chování distribuovaných systémů.

Formálně - **Petriho síť** je pětice (P, T, F, M_0, W), kde:

- P je **seznam míst**
- T je **seznam přechodů**
- F je **seznam hran** (žádná hrana nesmí spojovat dvě místa nebo dva přechody, tedy $F \subseteq (P \times T) \cup (T \times P)$)
- $M_0 : P \to \mathbb{N}$ je **iniciální označkování**, v každém místě $p \in P$, existuje $n_p \in \mathbb{N}$ tokenů.
- $W : F \to \mathbb{N}^+$ je **množina váh hran**, které každé hraně $f \in F$ přiřazují $n \in \mathbb{N}^+$ které označuje kolik tokenů je při přechodu danou hranou navazujícím přechodem zkonzumováno, případně kolik tokenů výchozí přechod generuje.

Stav sítě - je vektor, kde v každé položce je počet tokenů aktuálně obsažený v místě s příslušným indexem.

Vlastnosti Petriho siti

- **State-transition list** - seznam postupně navazujících přechodů mezi možnými stavy sítě. **State-transition matrix**.
- **Reachability** - otázka, zda-li je daný stav dosažitelný z výchozího. Řeší se pomocí kreslení grafu dosažitelnosti (kde každý vrchol je možný stav a hrana je přechod mezi stavy). Graf se musi budovat do šířky, protože může být nekonečný a tak bychom se při cestě do hloubky snadno mohli stratit …
- **Liveness** - vlastnost přechodů. Pokud je každý přechod $L_k\ live$, pak totéž můžeme říct o síti jako celku. Přechod je:
  - $L_0\ live$ (AKA ***dead***), právě když nemůže být **fired** (neexistuje firing sequence která by ho obsahovala) .
  - $L_1\ live$, právě když může být fired.
  - $L_2\ live$, právě když pro každé celé kladné k může být fired k-krát.
  - $L_3\ live$, právě když existuje firing sequence kde je přechod fired nekonečně
  - $L_4\ live$ (AKA *live*), právě když je přechod v každém dosažitelném stavu $L_1\ live$.

Boundedness - Petriho síť je **k-omezená**, pakliže v každém dosažitelném stavu má v každém místě vždy nejvýše k tokenů. Petriho síť je **ohraničená**, pakliže má její graf dosažitelnosti konečný počet vrcholů.

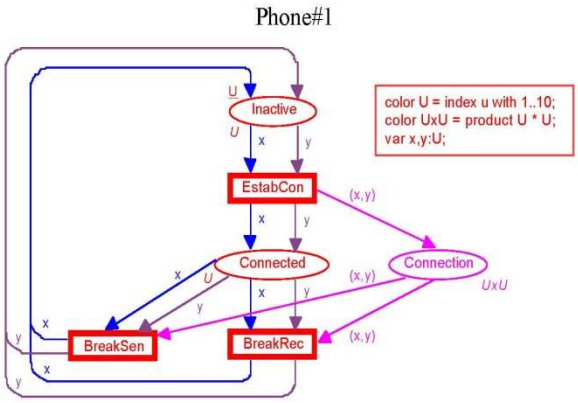
Možná rozšíření

- **Barevné Petriho sítě** - tokeny je možno rozlišovat.
- **Prioritizované petriho sítě** - přechody mají priority.

Hierarchie Petriho sítí

- **State machine** - každý přechod má 1 in a 1 out hranu => může vzniknout konflikt.
- **Marked graph** - každé místo má 1 in a 1 out hranu => může vzniknout konkurence.
- **Free choice** - hrana je buď jediná která vystupuje z místa, nebo jediná která vstupuje do přechodu => může být konflikt i konkurence, ale ne najednou.
- **Extended free choice** - taková, která se dá transformovat na FC.
- **Asymetric choice** - konkurence i konflikt povoleny, ale ne asymetricky.
- **Petri net** - všecko povoleno).

Hierarchical Coloured Petri Net



Vyjadrovací prostredky a metody (datové modelování, procesní modelování - funkcní a dynamické) strukturované analýzy a návrhu informacních systému

Fáze tvorby SW produktu

- **Úvodní studie** (informační plánování, feasibility study) - podklady pro rozhodnutí, zda vůbec má projekt smysl.
- **Analýza** - vytvoření dostatečně přesné specifikace produktu.
- **Návrh (design)** - dekompozice systému na komponenty, které lze naprogramovat.
- **Implementace** - realizace komponent definovaných v návrhu a jejich sestavení do výsledného produktu.

Úvodní studie

Měla by odpovědět na otázku **proč**? Musí proto vymezit hranici projektu. Musí odpovědet na otázky: “je projekt uskutečnitelný?” (feasibility study), “kdo a co bude k řešení zapotřebí?” a “vyplatí se projekt řešit?”

- **Vstupy úvodní studie** - Požadavky na systém - zadání projektu, deklarace záměru, vize projektu, odborný článek, tj. všechny dokumenty, které mají k řešenému problému nějaký vztah.
- **Výstupy úvodní studie** jsou:
    - **Definice systému**
        - Katalog požadavků
        - Definice hranice systému (diagram kontextu, model jednání)
        - Datový (pojmový) slovník
    - **Projektová dokumentace** - řesitelský tým (funkce, zodpovědnosti). Návrh řešení: HW, SW, komponenty. Seznam úloh a harmonogram řešení. Rozpočet: - cena HW, cena licencí na SW, cena vývoje SW a HW (COCOMO).

Deklarace záměru

Je krátký výstižný text se stručnými informacemi o projektu - jaké služby poskytuje, pro koho je určen a jaká předpokládá omezení. Měla by posloužit pro odpověď na otázku “co ano, a co ne?”. Je obvykle základem budoucího prospektu pro vytvořený produkt.

Odborný článek

Všechny informace, které lze o projektu sehnat (články, interview, předpisy, …). Označení „odborný článek“ má vystihovat představu, že se jedná o texty v přirozeném jazyce, které sepsal odborník na řešenou problematiku. Informatik ji bude analyzovat a vytvoří popis přesnější.

Katalog požadavků

Zadávací dokumentace obsahuje deklaraci záměru a odborný článek. Odborný článek je nejprve předzpracován tak, aby tvořil katalog požadavků. Katalog požadavků je strukturovaná verze odborného článku ve formě stromu požadavků. Požadavky jsou očíslovány a přes čísla se na ně lze odvolávat v další dokumentaci. Některé metodiky navrhují vybrat z textů podstatná jména (kandidáti na data) a slovesné fráze (kandidáti na služby - případy použití). Vybrané termíny uložit do významového (datového) slovníku.

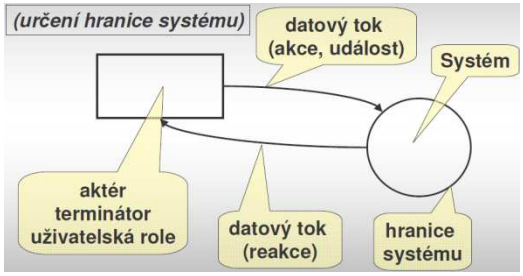
Formalizace informací

Formální vyjádření je nutné pro jednoznačnou interpretaci. Často se doporučuje z odborného článku vyrobit katalog požadavků a z něj extrahovat hned na začátku:

- **Aktéry** (uživatelské role a spolupracující systémy) - **Aktér** je cokoliv, co se nachází mimo **navrhovaný** systém**, co ale může vyvolat událost**, na kterou systém musí reagovat (vstupní událost - **akce**), nebo je to cokoliv, co může informaci od systému přijmout (výstupní událost – **reakce**). **Aktéři tedy představují uživatelské role a spolupracující systémy.**
- **Události** (případy použití) - Událost je cosi, co vzniklo mimo systém, ale systém na to musí reagovat. S událostí souvisí data, která procházejí přes hranici systému. Událost může vyvolat bezprostřední reakci systému (posílají se data nějakému aktérovi) a/nebo změnu stavu systému.
- **Potřebná a požadovaná data**

Kontextový diagram

Kontextový diagram slouží pro evidenci aktérů a datových toků.

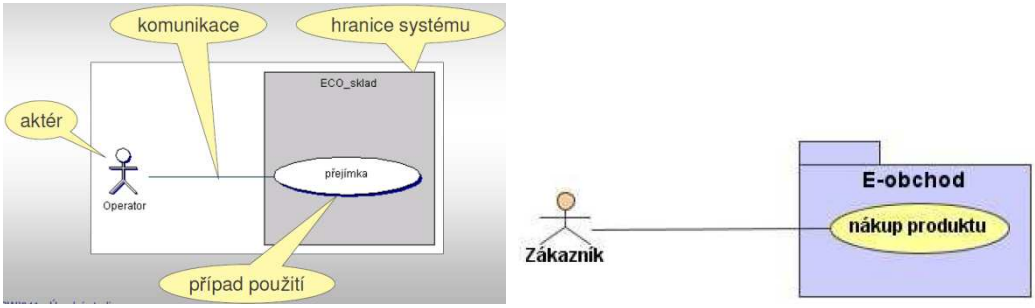


Model jednání (Use Case Model)

Model jednání (use case model) slouží pro evidenci aktérů a služeb systému.

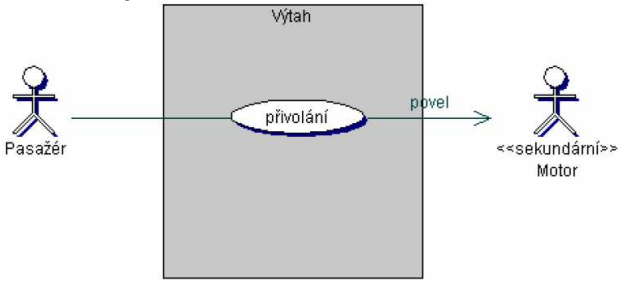
Prvky:

- **Aktér (actor)** - uživatelská role nebo spolupracující systém.
- **Hranice systému (systém boundary)** - vymezení hranice systému.
- **Případ použití (use case)** – dokumentace události, na kterou musí systém reagovat.
- **Komunikace** - vazba mezi aktérem a případem použití (aktér komunikuje se systémem na daném případu).

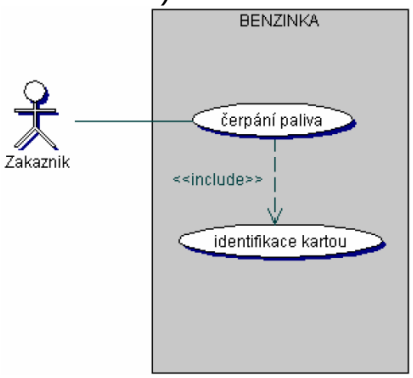


Doplňky k modelu jednání

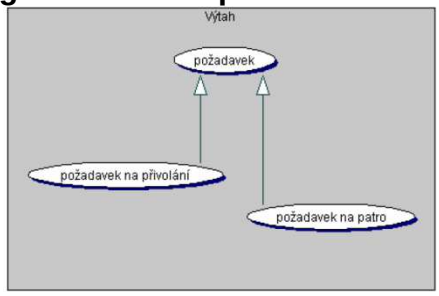
- **Orientovaná komunikace** - případ, kdy chceme vyznačit směr komunikace.
- **Sekundární aktér** - uživatelská role nebo spolupracující systém nutná pro činnost systému.



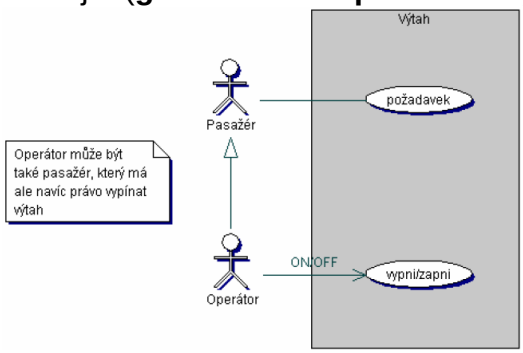
- **Vztahy mezi případy použití** – pokud chceme explicitně vyjádřit fakt, že takový vztah existuje.
  - **<<include>>** - pokud jeden případ zahrnuje případ jiný (např. autentizace).



  - **<<extend>>** - pokud nějaký případ rozšiřuje chování (je zde možnost volby)
  - **generalizace/specializace**



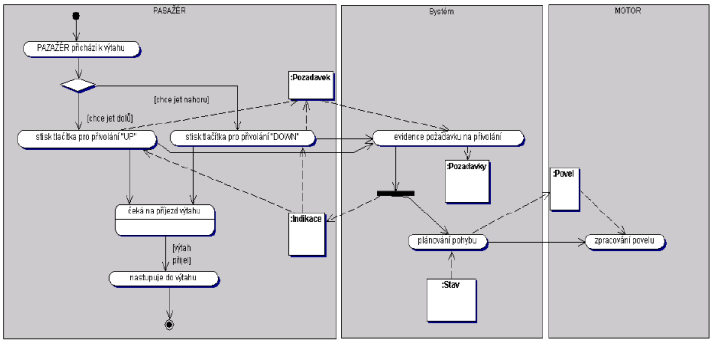
- **Vztahy mezi aktéry** - pokud chceme explicitně vyjádřit fakt, že takový vztah existuje. (**generalizace/specializace**)



Diagramy aktivit

V úvodní studii se používají pro dokumentaci případů použití (modelování procesů, modelování „workflow“, atd.). Nahrazují do určité míry v UML neexistující diagramy datových toků. Slouží jako prostředek pro domluvu mezi zadavatelem a řešitelem. Prvky:

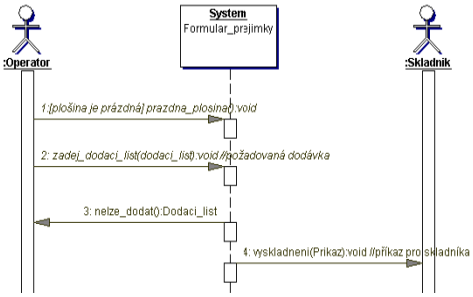
- **Aktivity** – činnosti, které modelujeme.
- **Přechody** – po ukončení činnosti se přejde k činnosti jiné.
- **Objekty** – s činností může souviset vytváření nebo konzumace objektů.
- **Začátek**, **Konec**
- **Synchronizační značky** (rozvětvení a synchronizace)
- **Plavecké dráhy** – okruhy zodpovědností.



Scénáře událostí (Sequence diagrams)

Slouží pro zachycení sledu událostí. Prvky:

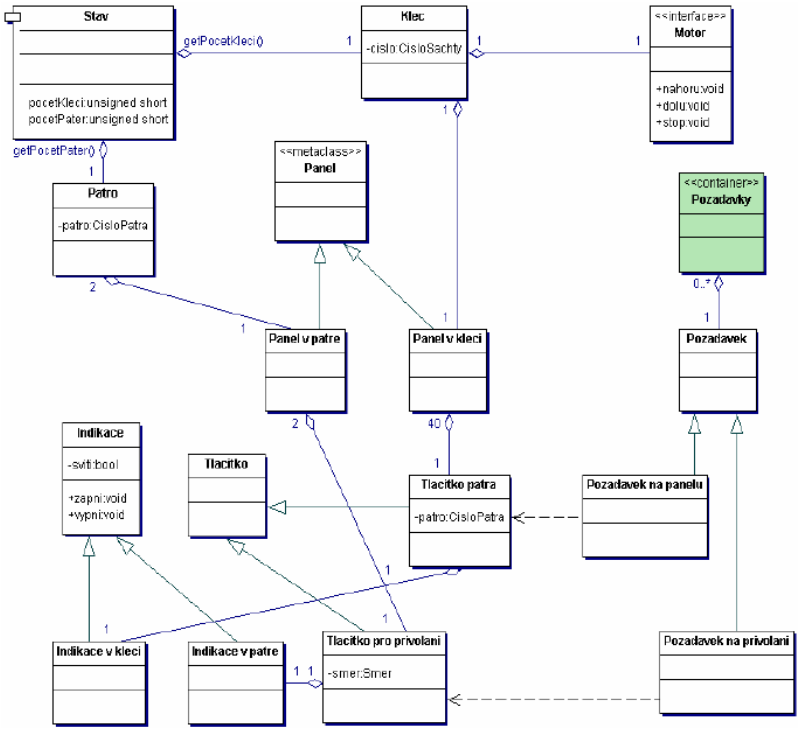
- **Objekty** - znázorněné obvykle jako sloupce.
- **Interakce mezi objekty (stimuly)** - orientované šipky mezi objekty.
- **Události** - události, které vyvolaly interakci.
- **Reakce** - odezvy na události (výstupy).
- **Časová osa** - pro vyznačení sledu událostí.



Datový model (konceptuální)

Používá se k zachycení analýzy dat. Komponenty:

- **Typy objektů** (entity) - entita = rozlišitelný identifikovatelný objekt.
- **Vztahy** (relationships) - množiny instancí reprezentujících vztahy mezi (2 a více) objekty.
- Indikace **přidružených** objektů - pro vztahy o nichž si potřebujeme něco pamatovat.
- Indikace vztahů **nadtyp-podtyp**, **celek-část** (genspec, whole-part) - vyjádření vztahu společný - speciální (dědičnost).



Analýza

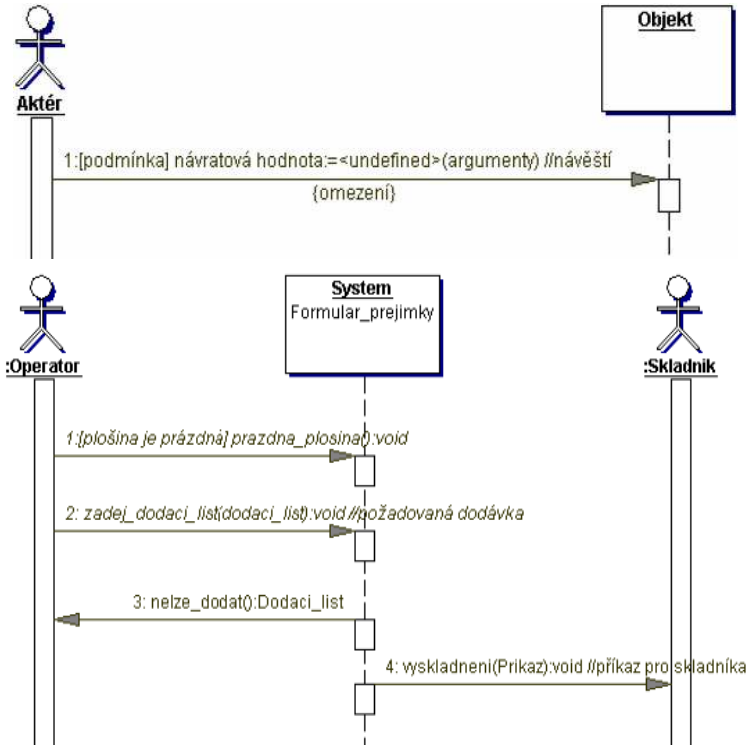
Výstup analýzy = konceptuální model:

- **Datový model** - popisuje entity, atributy, vztahy, integritní omezení.
- **Funkční model** - popisuje služby, které systém poskytuje pro záznam, údržbu a využití dat.
- **Dynamický model** - popisuje možné stavy data jejich změny.

Datové modelování – pozostává z jednotlivých fáz:

- Sběr požadavků
- Analýza dat a vytvoření **konceptuálního datového modelu** (ER-model, model tříd, PIM)
- Návrh reprezentace dat – **logický datový model** (např. relační model, objektový model, PSM)
- Implementace datového modelu – skutečné vyjádření datových charakteristik pro konkrétní prostředí (**fyzický model**)

Funkční model



Popis akce (operace, funkce)

- **Operation**: název
- **Description**: textový popis
- **Reads**: jaká data jen čte
- **Changes**: jaká data mění nebo vytváří
- **Sends**: jaké reakce vyvolává (jaké zprávy posílá)
- **Assumes**: co předpokládá
- **Results**: co zajišťuje (zaručuje)

Příklad: Popis pro "prázdná plošina"

- Operation: prázdná plošina
- Description: informuje systém, že nakládací plošina je prázdná
- Reads:
- Changes: plošina
- Sends:
- Assumes:
- Results: vyprázdní v modelu nakládací **plošinu,** uvolní identifikátory barel, které jsou na **plošině**

Odvození funkcí z datového modelu

Vytvoříme matici CRUD (Create, Read, Update, Delete) a zkoumáme, zda pro každý typ dat existuje odpovídající funkce. Řádky odpovídají typům objektů. Sloupce odpovídají funkcím. V průsečíku je zapsáno, zda funkce C,R,U a/nebo D odpovídající data. V každém řádku by mělo být vše. Některá funkce musí objekt vytvářet, jiná využívat, či rušit.

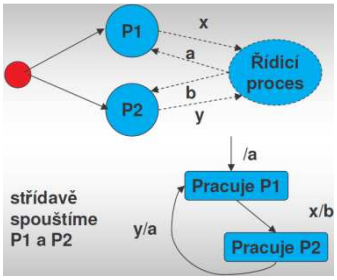
Dynamický model

Slouží k popisu dynamiky systému. Stavový diagram definuje možné **stavy**, možné **přechody** mezi stavy, **události**, které přechody iniciují, **podmínky** přechody a **akce**, které s přechody souvisí. Stavový diagram lze použít pro popis dynamiky objektu (pokud má rozpoznatelné stavy), pro popis metody (pokud známe algoritmus), či pro popis protokolu (včetně protokolu o styku uživatele se systémem).



Popis řídicích procesů pomocí stavových diagramů

Vstupy řídicího procesu lze modelovat pomocí událostí stavového diagramu. Výstupy řídicího procesu lze modelovat pomocí akcí stavového diagramu. Pak lze řídicí procesy modelovat stavovými diagramy.



Životní cyklus systému

Vyjádření souhrné dynamiky systému, která je zachycena ve scénářích. Definuje povolené návaznosti akcí a reakcí. Představuje hrubou "uživatelskou příručku" pro systém. Definice systému jako "konečného automatu". Životní cyklus se dá zapsat i jako regulární výraz:

```
<Životní cyklus> = Lifecycle <jméno objektu> : <regulární výraz>
<regulární výraz> =
    <akce>
    | #<reakce>
    | <regulární výraz>. <regulární výraz>          sekvence
    | [ <regulární výraz> ]                         volitelně
    | <regulární výraz>*                            iterace
    | (<regulární výraz> | <regulární výraz>)       selekce
    | (<regulární výraz> || <regulární výraz>)      paralelně
<akce> = jméno události
<reakce> = jméno reakce
```

Lifecycle BAREL **:** zarad_barel . #ID barelu . #příkaz prouskladnění . dodávka . #příkaz pro vyskladnění

Kontroly analytických modelů

1. **Kontrola datového modelu**
   - existuje entita pro každý typ objektu?
   - nejsou zde nadbytečné entity (entity tvořené pouze identifikací, entity s jedinou instancí, apod.)?
   - jsou zde zaneseny všechny vztahy (včetně generalizací a agregací)?
   - nejsou zde odvoditelné vztahy?
   - je model v normální formě?
   - jsou zanesena všechna integritní omezení?

2. **Kontrola funkčního modelu**
   - existuje funkce/metoda pro každou událost?
   - každá funkce/metoda musí být popsána dekompozicí, nebo mít minispecifikaci (vstupy a výstupy musí odpovídat)
   - nejsou zde nadbytečné funkce/metody?

3. **Kontrola dynamického modelu**
   - existuje model pro každou entitu, která může mít různé stavy?
   - existuje model pro každý řídicí proces?
   - existuje popis životního cyklu systému?

Návrh

Z analytického konceptuálního modelu, kde je popsáno **CO**, musíme navrhnout, **JAK** se to udělá. Kroky návrhu:

- návrh architektury systému
- návrh uživatelského vzhledu
- návrh komponent
- návrh komunikace mezi komponentami
- návrh způsobu integrace komponent a testování celku

Základní technologická rozhodnutí

- architektura systému
- datové zdroje, datové paměti, přístupové mechanismy k nim
- distribuce programových modulů, komunikační mechanismy
- typy a formy výstupů
- uživatelské rozhraní
- vývojové prostředí

Výstupní dokumenty návrhu

Architektura systému (HW, SW), Popis implementace dat (logický datový model), Popis komponent (modulů), Projektová dokumentace návrhu.

Návrh (datový model)

Z analytického konceptuálního datového modelu musíme navrhnout model logický. Konceptuální datový model specifikuje:

- Typy dat (které entity, třídy, …)
- Vztahy mezi nimi
- Další logická (integritní) omezení (integrity constraints)

Návrh reprezentace dat

Pro každou datovou paměť (úložiště) musíme navrhnout způsob reprezentace - může to být systém ovládání souborů, systém řízení báze dat (relační, objektové, objektově-relační), speciální datový stroj (SW, SW+HW). Následuje převod konceptuálního modelu do logického. Součástí převodu je i návrh zajištění integrity dat. Návrh zajištění konzistence dat, zálohování, archivace apod.

Návrh reprezentace vrh dat pomocí relačního databázového systému

- **Vstup**: konceptuální datový model (diagram tříd + popis integritních omezení).
- **Výstup**: logický relační datový model (SQL- 1999), včetně návrhu realizace integritních omezení.

Postup návrhu

- **Úprava (normalizace) konceptuálního modelu**
  - Vyloučení multihodnotových a násobných atributů
  - Vyloučení funkčních závislostí (odstranění redundance dat) – převod modelu do 3-NF.
  - Náhrada nebinárních vztahů binárními
  - Náhrada vztahů typu M:N přidruženými třídami
- **Návrh reprezentace typů (entit)**
  - Pro každou jednoduchou entitu (typ) navrhneme tabulku, jméno tabulky bude množné číslo jména typu.
  - Návrh jmen sloupců pro reprezentaci atributů a odpovídajících domén.
  - Doplníme informace o volitelnosti formátu sloupců.
  - Z nejčastěji používané unikátní identifikace vytvoříme primární klíč, nebo zavedeme nový identifikační sloupec (OID).
- **Návrh reprezentace vztahů**
  - Pro N-konce vztahů přidáme k tabulce jednoznačné identifikace z tabulky na 1-konci (volitelné vztahy indikují nepovinnost. Současně přidáme odpovídající cizí klíče.
  - Pro každý vztah typu nadtyp/podtyp navrhneme reprezentaci (společná tabulka s rozlišovací položkou, samostatné tabulky).
  - Pro každý vztah typu celek/část navrhneme reprezentaci (společná tabulka s rozlišovací položkou, samostatné tabulky).
- **Návrh reprezentace integritních omezení**
  - použijeme deklarativní relační integritní omezení, nebo navrhneme „triggery“.

Návrh (funkční model)

Funkční model musíme rozložit na části zvládnutelné technikami programování (v malém). Takové části se obvykle nazývají moduly, komponenty, …

Z analytického funkčního modelu, kde je popsáno **CO** (jaké služby), musíme navrhnout, **JAK** se to udělá (který modul bude službu realizovat a jakými dalšími moduly bude muset spolupracovat).

Techniky návrhu modulů:

- **Objektový návrh** (např. Coleman) – z analytických modelů vytvoříme popisy tříd.
- **Návrh na základě datových toků** (např. Page- Jones) – z analytických diagramů datových toků vytvoříme diagramy struktury.
- **Návrh na základě datových struktur** (např. Jackson) – z analytických popisů datových struktur vytvoříme popisy programů.
- **Návrh fukční dekompozicí** – z analytických popisů funkcí vytvoříme přímou funkční dekompozicí dostatečně podrobné popisy funkcí.

Návrh dle Fusion

Vstup: Datový slovník, Objektový model (ER-model), Operační model (kontext, model jednání, scénáře, popisy operací), Model životního cyklu.

Výstup: Datový slovník, Architektura systému (HW, SW), Popis komponent jako popis tříd, Model životního cyklu.

Postup dle Fusion

Datový slovník → objektový model → funkční model - popisy operací → model životního cyklu → model interakce - scénáře, diag.kolaborace → graf viditelnosti → popis tříd → graf dědičnosti → program.

Postup návrhu ve Fusion

- Pro každou operaci funkčního modelu nakresli diagram interakce (včetně metod vzniklých při návrhu).
- Pro každou třídu datového modelu nakresli graf viditelnosti podle diagramů interakce. Graf viditelnosti popisuje nutné vazby mezi třídami, které souvisí s potřebou viditelnosti (mám-li volat službu objektu jiné třídy, musím jej vidět).
- Pro každou třídu vytvoř popis třídy.
- Doplň návrh grafy dědičnosti.

Diagramy datových toků (Data Flow Diagrams (DFD))

Hierarchická sada diagramů, jejíž vrchol tvoří diagram kontextu a jejíž listy představují elementární funkce, popsané již jiným způsobem, než dekompozicí (pomocí tzv. minispecifikací – popisů operací, metod).

Úvod do databázových systémů

Definice: **Databáze** je logicky usporádaná (integrovaná) kolekce navzájem souvisejících dat. Je sebevysvetlující, protože data jsou uchovávána spolecne s popisy, známými jako metadata (také schéma databáze). Data jsou ukládána tak, aby na nich bylo možné provádet strojové dotazy – získat pro nejaké parametry vyhovující podmnožinu záznamu. Nekdy se slovem „databázeÿ myslí obecne celý databázový systém.

Definice: **Systém rízení báze dat (SRBD, anglicky database management system, DBMS)** je obecný softwarový systém, který rídí sdílený prístup k databázi, a poskytuje mechanismy, pomáhající zajistit bezpecnost a integritu uložených dat. Spravuje databázi a zajištuje provádení dotazu.

Definice: Databázovým systémem rozumíme trojici, sestávající z: **databáze**, **systému rízení báze dat** a chudáka **admina**.

Smysl databází

Hlavním smyslem databáze je schranovat datové záznamy a informace za úcelem: sdílení dat více uživateli, zajištení unifikovaného rozhraní a jazyku definice dat a manipulace s daty, znovuvyužitelnosti dat, bezespornosti dat a snížení objemu dat (odstranení redundance).

Databázové modely

Definice: Typicky pro každou databázi existuje strukturální popis druhu dat v ní udržovaných, ten nazýváme **schéma**. Schéma popisuje objekty reprezentované v databázi a vztahy mezi nimi. Je nekolik možných zpusobu organizace schémat (modelování databázové struktury), známých jako **modely**. V modelu jde nejen o zpusob strukturování dat, definuje se také sada operací nad daty proveditelná. Relacní model napríklad definuje operace jako „select“ nebo „join“. I když tyto operace se nemusejí prímo vyskytovat v dotazovacím jazyce, tvorí základ, na kterém je jazyk postaven. Nejduležitejší modely v této sekci popíšeme.

Poznámka

Vetšina databázových systému je založena na jednom konkrétním modelu, ale cím dál castejší je podpora více prístupu. Pro každý logický model existuje více fyzických prístupu implementace a vetšina systému dovolí uživateli nejakou úroven jejich kontroly a úprav, protože toto má velký vliv na výkon systému. Príkladem necht jsou indexy, provozované nad relacním modelem.

„Plochý“ model

Toto sice nevyhovuje úplne definici modelu, presto se jako triviální prípad uvádí. Predstavuje jedinou dvoudimensionální tabulku, kde data v jednom sloupci jsou považována za popis stejné vlastnosti (takže mají podobné hodnoty) a data v jednom rádku se uvažují jako popis jediného objektu.

Relacní model

Relacní model je založen na predikátové logice a teorii množin. Vetšina fyzicky implementovaných databázových systému ve skutecnosti používá jen aproximaci matematicky definovaného relacního modelu. Jeho základem jsou relace (dvojdimensionální tabulky), atributy (jejich pojmenované sloupce) a domény (množiny hodnot, které se ve sloupcích mužou objevit). Hlavní datovou strukturou je tabulka, kde se nachází informace o nejaké konkrétní tríde entit. Každá entita té trídy je potom reprezentována rádkem v tabulce – n-ticí atributu. Všechny relace (tj. tabulky) musí splnovat základní pravidla – poradí sloupcu nesmí hrát roli, v tabulce se nesmí vyskytovat identické rádky a každý rádek musí obsahovat jen jednu hodnotu pro každý svuj atribut. Relacní databáze obsahuje více tabulek, mezi kterými lze popisovat vztahy (všech ruzných kardinalit, tj. 1 : 1, 1 : n apod.). Vztahy vznikají i implicitne napr. uložením stejné hodnoty jednoho atributu do dvou rádku v tabulce. K tabulkám lze pridat informaci o tom, která podmnožina atributu funguje jako klíc, tj. unikátne identifikuje každý rádek, nekterý z klícu muže být oznacen jako primární. Nekteré klíce mužou mít nejaký vztah k vnejšímu svetu, jiné jsou jen pro vnitrní potreby schématu databáze (generovaná ID).

Hierarchický model

V hierarchickém modelu jsou data organizována do stromové struktury – každý uzel má odkaz na nadrízený (k popisu hierarchie) a setrídené pole záznamu na stejné úrovni. Tyto struktury byly používány ve starých mainframeových databázích, nyní je mužeme videt napr ve strukture XML dokumentu. Dovolují vztahy 1 : N mezi dvema druhy dat, což je velice efektivní k popisu ruzných reálných vztahu (obsahy, razení odstavcu textu, trídené informace). Nevýhodou je ale nutnost znát celou cestu k záznamu ve strukture a neschopnost systému reprezentovat redundance v datech (strom nemá cykly).

Sítový model

Sítový model organizuje data pomocí dvou hlavních prvku, záznamu a množin. Záznamy obsahují pole dat, množiny definují vztahy 1 : N mezi záznamy (jeden vlastník, mnoho prvku). Záznam muže být vlastníkem i prvkem v nekolika ruzných množinách. Jde vlastne o variantu hierarchického modelu, protože sítový model je také založen na konceptu více struktur nižší úrovne závislých na strukturách úrovne vyšší. Už ale umožnuje reprezentovat i redundantní data. Operace nad tímto modelem probíhají „navigacnímÿ stylem: program si uchovává svoji současnou pozici mezi záznamy a postupuje podle závislostí, ve kterých se daný záznam náchází. Záznamy mohou být i vyhledávány podle klíce. Fyzicky jsou vetšinou množiny – vztahy – reprezentovány prímo ukazateli na umístení dat na disku, což zajištuje

vysoký výkon pri vyhledávání, ale zvyšuje náklady na reorganizace. Smysl sítové navigace mezi objekty se používá i v objektových modelech.

Objektový model

Objektový model je aplikací prístupu známých z objektove-orientovaného programování. Je založen na sbližování programové aplikace a databáze, hlavne ve smyslu použití datových typu (objektu) definovaných na jednom míste; ty zpřístupňuje k použití v nejakém bežném programovacím jazyce. Odstraní se tak nutnost zbytečných konverzí dat. Prináší do databází také veci jako zapouzdrení nebo polymorfismus. Problémem objektových modelu je neexistence standardu (nebo spíš produktu, které by je implementovaly). Kombinací objektového a relacního prístupu vznikají objektove-relacní databáze – relacní databáze, dovolující uživateli definovat vlastní datové typy a operace na nich. Obsahují pak hybrid mezi procedurálním a dotazovacím programovacím jazykem.

Konceptuální modelování, databázové modelování, implementace

Konceptuální datové modelování

Datová analýza (ne funkční analýza) zpravidla následuje po analýze informačního systému (ta řeší funkcionalitu systému). Probiha modelování schématu databáze modelování „datové reality“ (jaká budeme mít v IS data). Napr. ER modelem.

Logické schéma

Logické schéma je datový model organizace nejakého specifického celku pomocí jednoho z databázových modelu – podle databázových modelu popsaných v predchozí sekci, tj. napr. pomocí relacních tabulek, objektových tríd nebo XML. Svojí úrovní abstrakce se nachází mezi konceptuálním a fyzickým schématem.

Fyzické schéma

Fyzické datové modely jsou modely, ktere používají databazové stroje smerem k nižším vrstvám (operacního) systému. V zásade jde o ruzné zpusoby fyzického uložení dat (tedy schémata organizace souboru) – sekvencní soubory, B-stromy apod.

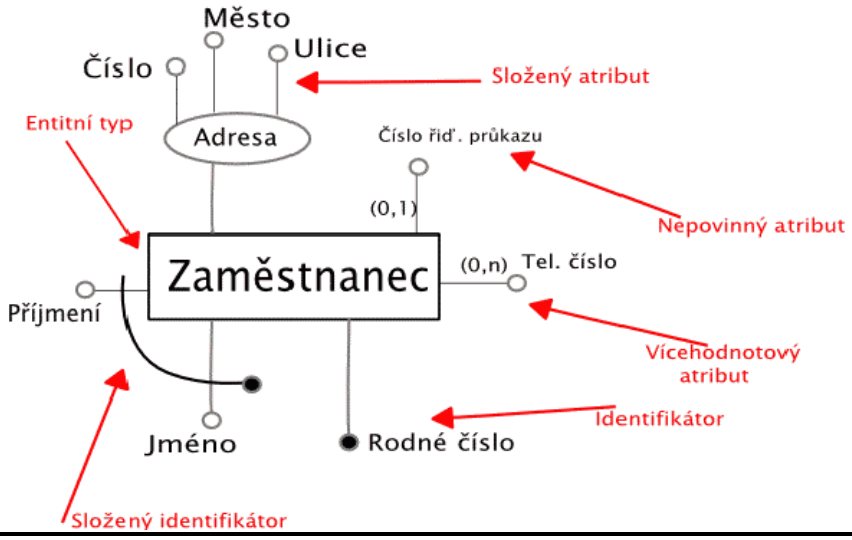
ER model

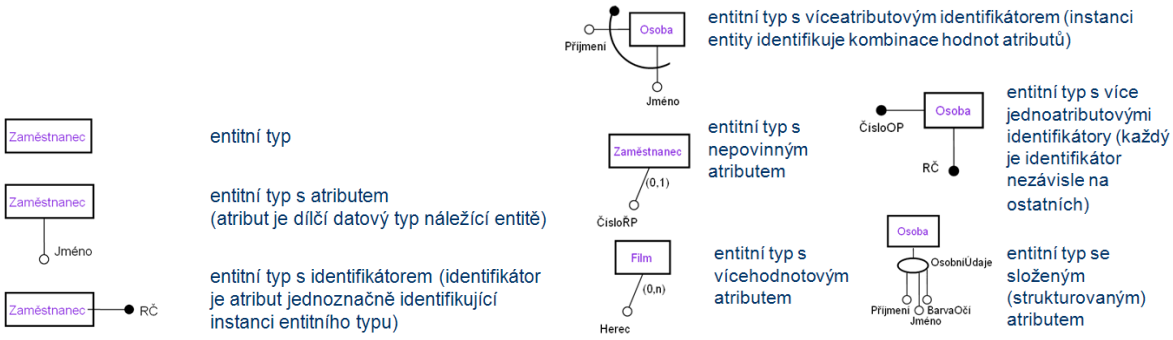
Je de-facto standardem pro datové modelování. Vhodny pro „plochá“ formátovaná (strukturovaná) data (objektové, relační a objektově-relační databáze). Nevhodné pro multimediální data, XML, text. E-R (**entity-relationship**) modelování pouziva dva typy „objektů“ – **entity** (mají atributy a identifikátory) a **vztahy** (mezi entitami. Mají kardinality 1:1,1:N, M:N). ER model databáze definuje její **konceptuální schéma**. ER modelování v DB je obdobou UML v OOP. Hotovy ER model se pak prevadi schématu relacni databáze – do schématu logického modelu.

Entitní typ

Entitní typ (v diagramu se znací hranatým rámeckem) reprezentuje nejakou trídu entit (napr. „Zamestnanec“). Každý entitní typ má nejaké **atributy** (napr. „jméno“), z nichž nekteré mohou být **identifikátory**, tj. takové, které jednoznacne urcují instanci entity. Pokud nemá žádné identifikátory explicitne oznacené, jsou jimi všechny atributy dohromaty (tzv. složený identifikátor). Identifikátory mohou být i víceatributové.

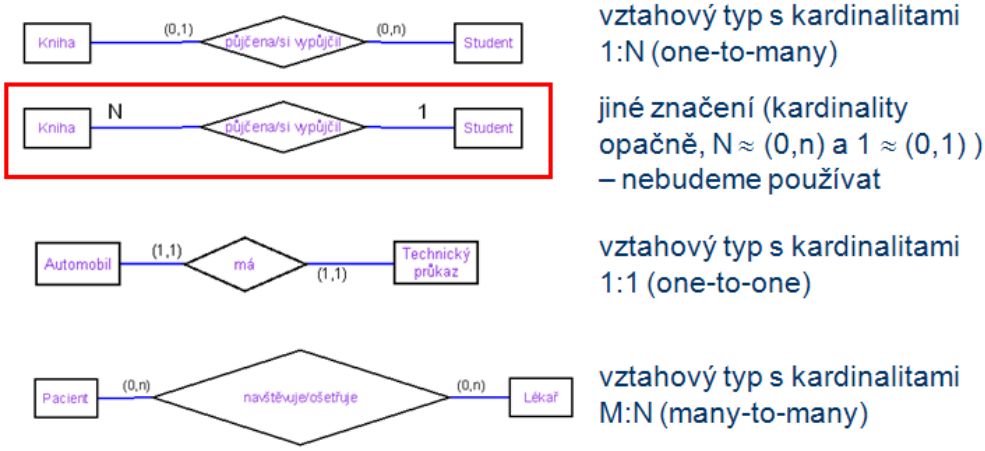
Atributy entitních typu mohou být **jednoduché** nebo **složené**, **povinné** ci **nepovinné**, prípadne **jednohodnotové** a **vícehodnotové**. Jejich zobrazení ukazuje následující obrázek:





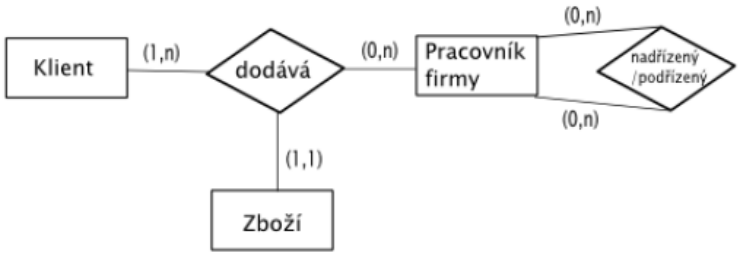
Vztahový typ

Vztahový typ (v diagramu znacený kosoctvercem) popisuje vztahy mezi jednotlivými entitami – s temi entitami, se kterými je v nejakém vztahu, je spojen carou. Vztah muže mít danou i **kardinalitu** (kolik entit z každé strany do vztahu vstupuje), která muže být typu **1 : 1**, **1 : n**, **m : n** a je znacená vedle cáry spojující vztahový typ s entitou. Entity ve vztahu mohou mít navíc **povinné** ci **nepovinné clenství** (vstupovat do nej vždy nebo jen nekdy).

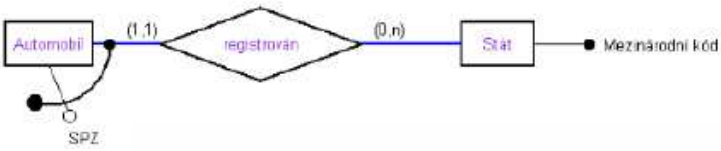


Vztahy mohou být bud binární nebo obecne **n-ární**, ale více než ternární vztahy se vetšinou neobjevují. Vztahy mohou být i rekurzivní, tj. do vztahu vstupují entity stejného typu. Instance vztahového typu je jednoznacne urcena identifikátory

instancí entit ve vztahu. Nekteré entitní typy mohou být spoluidentifikovány (nebo prímo identifikovány) vztahem – pak se nazývají **slabé entitní typy**.



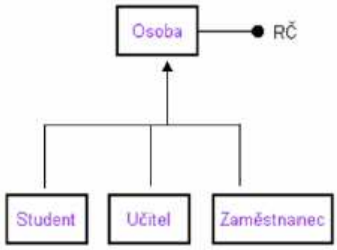
Předchozí obrázek ukazuje ternární vztah s ruznými kardinalitami – klientovi nekdo dodává zboží jednou až n-krát, pracovník dodává nula až n-krát zboží (tj. jde o nepovinné clenství ve vztahu, mužou existovat pracovníci, kterí nic nedodávají) a zboží je vždy nekomu dodáváno práve jednou. Na zamestnancích je zároven ukázán rekurzivní binární vztah.



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

ISA hierarchie

ISA hierarchie je rozšírení ER diagramu o „dědičnost“ entit – tj. rozdelení entitních typu na subtypy (a pridání dalších vztahu nebo atributu pro subtypy). V ISA hierarchii se povoluje pouze jednonásobná dedicnost, navíc potomci nejakého entitního typu musí být jednoznacne identifikováni predkem (tj. všechny entity v hierarchii sdílí identifikátor).



Úpravy ER diagramu

V ER diagramu je možné provádet víceméne „ekvivalentní“ úpravy (výsledný diagram reprezentuje stejný koncept databáze), napr. pro odstranení vztahu s kardinalitou m : n (prevod na dva vztahy kardinalitami 1 : n a **prunikový entitní typ**, který je vztahy urcený, takže je slabý). Dalším duvodem úprav muže být zbavení se ISA hierarchie. To se dá provést více zpusoby, pricemž žádný z nich nefunguje úplne obecne:

- agregace atributu a vztahu potomka do predka a úprava kardinalit (prevod na nepovinné atributy a nepovinné clenství ve vztahu)
- odstranení predka a duplikace všech jeho atributu a vztahu v potomcích
- nahrazení ISA vztahu klasickým vztahem (z potomku vzniknou slabé entitní typy) Jiná úprava je odstranení vícehodnotového atributu – prevede se na vztah s kardinalitou 1 : n a slabý entitní typ.

Relační model

Relační model neformálně

Relační model je model uchovávání dat (resp. vztahů mezi daty) v **tabulkách**. Datová **entita** E je reprezentována **řádkem** tabulky. **Atribut** A entity je reprezentován **sloupcem** tabulky. Buňka tabulky na pozici (x,y) uchovává hodnotu atributu A(x) v kontextu entity E(y).

Schéma tabulky je popis struktury tabulky (všeho kromě dat v ní). $S(A_1:T_1, A_2:T_2, ...)$;kde S je název tabulky, A_i jsou atributy a T_i jejich typy.

Schéma relační databáze je množina schémat tabulek (+ další věci, např. integritní omezení, atd.).

Základní integritní omezení

- neexistují 2 stejné řádky (unikátní identifikace entit)
- rozsah hodnot atributu je dán typem (jednoduché typované atributy)
- tabulka není „děravá“, tj. všechny hodnoty jsou nenulové
- každý řádek je identifikovatelný (odlišitelný od ostatních) jedním nebo více pevně určenými atributy v rámci tabulky (taková skupina atributů se nazývá **nadklíč** tabulky. nadklíč s minimálním počtem atributů se nazývá klíč tabulky (může existovat víc klíčů)).
- speciální „mezitabulkové“ integritní omezení – **cizí klíč** (= skupina atributů, která existuje i v jiné (tzv. referenční) tabulce a tam tvoří klíč - důsledky: není to klíč v tabulce referující, mohou existovat duplicitní hodnoty).

Relační model formálně

Odděluje data od implementace (přes pojem relace) – při manipulacích s daty se nezajímáme o konkrétní přístupové mechanismy. Existuji dva silné prostředky pro dotazování relačních dat: **relační kalkul** a **relační algebra**. Nad relacnim modelem lze vhodně navrhovat relační schémata pomocí **normalizací relačních schémat** (algoritmicky - **dekompozice** a **syntéza**).

Definice relacniho modelu

Algebraický pojem relace $R \subseteq D_1 \times D_2 \times ... \times D_n$ je „databázově“ rozšířen - množiny D_i jsou tzv. **atributové domény – tj. typy** a **schéma relace**, které je určeno výčtem atributů (případně jejich typů):

$R(A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$, také $R(A)$, kde $A = \{A_1, A_2, ... , A_n\}$

Tabulky vs. relace

- **Relace** odpovídá konkrétním **datům** v tabulce.
- **Prvek relace** odpovídá konkrétnímu **řádku** (**záznamu**) v tabulce.
- **Schéma relace** odpovídá **schématu tabulky**.
- **Domény** relace odpovídají **typům atributů** tabulky.

Rozsireni relačního schema o funkcni zavislosti ($R(A, F)$; $F = \cup_i\{f_i\}$)

Funkční závislost (FZ) $X \rightarrow Y$ nad schématem $R(A)$ je parciální zobrazení $f_i : X_i \rightarrow Y_i$, kde $X_i, Y_i \subseteq A$ (kde i = 1..počet závislostí pro R(A)). *N*-tice z X_i **funkčně určuje** *m*-tici z Y_i (neboli hodnoty atributů X **společně** určují hodnoty atributů Y). *M*-tice z Y_i funkčně závisí na *n*-tici z X_i.

Pokud $X \rightarrow Y$ a $Y \rightarrow X$, potom X i Y jsou **funkčně ekvivalentní** a lze psát $X \leftrightarrow Y$. Pokud $X \rightarrow a$, kde $a \in A$, potom $X \rightarrow a$ je **elementární funkční závislost.**

Klíč $K \subseteq A$ je definován pomocí funkčních závislostí jako:

- **$K \rightarrow A$ …** K je alespon **nadklic.** (Atribut obsazen v klici = **klicovy atribut**).
- Neexistuje $K' \subset K$ takové, že $K' \rightarrow A$ … K je **minimální klic.**

E-R schémata a jejich transformace do relacního modelu

Převod ER diagramu do schématu relačního databáze

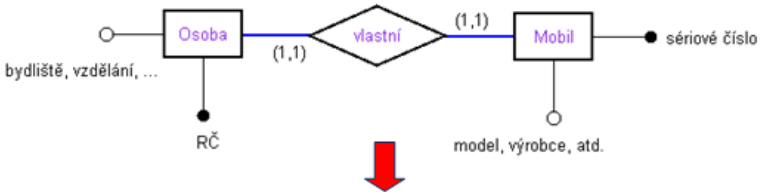
Chceme implementovat ER koncept v relační databázi. Převod je realizován ze schéma konceptuálního (sémantického) modelu na schéma logického modelu (schéma relační databáze). Snažíme se při tom zachovat integritních omezení (pokud možno 1:1).

Převod obecně

- **Silný entitní typ** → tabulka. Klíčem je identifikátor entity.
- **Slabý entitní typ** → tabulka. Klíčem je smíšený (externí) identifikátor.
- **Vztahový typ** → tabulka. Počet tabulek a jejich sdílení vztahy a entitami je řídí kardinalitami vztahu, do něhož tyto entity vstupují.

Kardinality (1,1) : (1,1)

- jediná tabulka – klíčem může být identifikátor libovolné z entit (nebo oba)

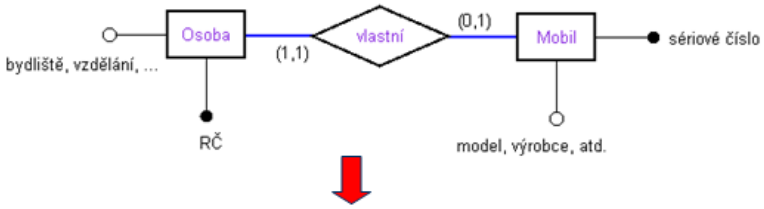


Relační schéma:

OsobaMobil(**RČ**, bydliště, vzdělání, ..., **sériové číslo**, model, výrobce, ...)

Kardinality (1,1) : (0,1)

- dvě tabulky T1 a T2
  - T1 existuje nezávisle na T2
  - T2 obsahuje identifikátor T1, který je zde cizím klíčem do T1
  - reprezentace vztahu je „ukrytá“ v tabulce T2

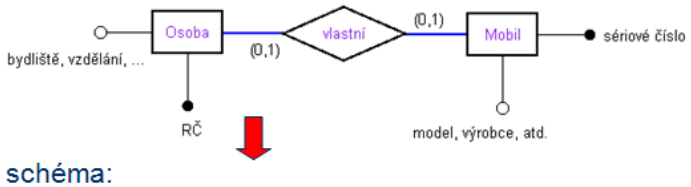


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ..., **sériové číslo**)
Mobil(**sériové číslo**, model, výrobce, ...)

Kardinality (0,1) : (0,1)

- tři tabulky – dvě entitní a jedna vztahová
  - vztahová tabulka má dva klíče odpovídající identifikátorům obou entitních typů
  - klíče jsou zároveň cizí klíče do entitních tabulek

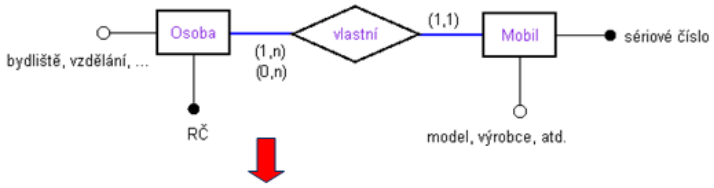


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Kardinality (0/1,n) : (1,1)

- podobně jako u (1,1) : (0,1) – dvě tabulky T1 a T2
  - T1 existuje nezávisle na T2
  - T2 obsahuje identifikátor T1, který je zde cizím klíčem do T1
  - reprezentace vztahu je „ukrytá“ v tabulce T2

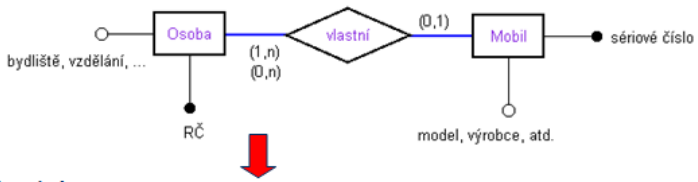


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ..., **RČ**)

Kardinality (0/1,n) : (0,1)

- podobně jako (0,1) : (0,1) – tři tabulky – dvě entitní a jedna vztahová
  - vztahová tabulka má jeden klíč odpovídající identifikátoru jednoho z entitních typů, identifikátor druhého entitního typu je pouze cizím klíčem
  - klíč je zároveň cizí klíč do jedné z entitních tabulek

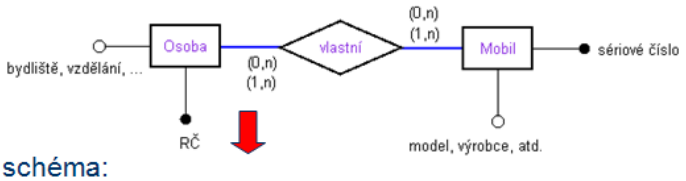


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Kardinality (0/1,n) : (0/1,n)

- nejobecnější situace – tři tabulky
  - dvě entitní a jedna vztahová
  - klíč ve vztahové tabulce je složený z identifikátorů obou entitních typů



Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Návrh relacních schémat v 3NF

Problémy při návrhu relačního schématu

Redundance – ukládání stejné informace víckrát na různých místech (zvyšuje prostorové nároky). Mohou se objevovat **aktualizacni anomálie** (při modifikaci jedne kopie redundantních dat je třeba zmenit i ostatne kopie).

Normalizace, anomálie

Normalizace databází je technika návrhu relacních databázových tabulek, pri které se minimalizují duplicity informací - a zamezuje se tak nekonzistentnosti dat. Stupne normalizace se „popisují“ pomocí normálních forem - cím vyšší forma, tým vyšší striktnost. Problémy rešené normalizací:

- **update anomaly** – pr.: tabulka (clovek, adresa, skill); kdyby se nevykonal update správne, muže tabulka zustat v nekonzistentním stavu (napr. by se mohly zmenit jen nekteré adresy jednoho cloveka). Obecně pokud se změní jedna kopie redundantních dat, je třeba změnit i ostatní kopie, jinak se databáze stane nekonzistentní.
- **insertion anomaly** – za jistých okolností by se nekterá fakta nedala zaznamenat, napr. v tabulce (fakulta, datum založení, kurz) mužeme zaznamenat jen data pro fakulty, které mají kurzy. Obecně při vložení dat příslušejících jedné entitě je potřeba zároveň vložit data i o jiné entitě.
- **deletion anomaly** – za jistých okolností by se mohlo stát, že vymazání některých faktu by zpusobilo vymazání dat reprezentujících jiná fakta.
  V predchozí tabulce bude fakulta vymazána úplne, když se vzdá všech kurzu.
  Obecně při vymazání dat příslušejících jedné entitě je potřeba vymazat data patřící jiné entitě.

Ideálne by relacní databáze mela být navržena tak, aby vylucovala možnost takových anomalií. Normalizace obvykle zahrnuje dekomponování nenormalizované tabulky na dve nebo více tabulek takových, že po jejich spojení (join) dostaneme všechny puvodní informace. Normalizace DB schématu využívá funkční závislosti.

Armstrongova pravidla

Mějme R(A,F). Nechť X, Y, Z $\subseteq$ A a množinu funkčních závislostí F:

1) jestliže Y $\subseteq$ X, potom X $\rightarrow$ Y	(**triviální FZ**, axiom)
2) jestliže X $\rightarrow$ Y a Y $\rightarrow$ Z, potom X $\rightarrow$ Z	(**tranzitivita**, pravidlo)
3) jestliže X $\rightarrow$ Y a X $\rightarrow$ Z, pak X $\rightarrow$ YZ	(**kompozice**, pravidlo)
4) jestliže X $\rightarrow$ YZ, pak X $\rightarrow$ Y a X $\rightarrow$ Z	(**dekompozice**, pravidlo)

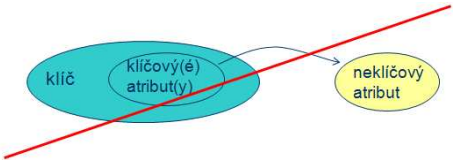
První normální forma (1NF)

Každý atribut schématu relace je elementárního typu a je nestrukturovaný. 1NF je tedy základní podmínka „plochosti databáze“ – tabulka je opravdu dvourozměrné pole, ne např. skrytý strom nebo graf. To zahrnuje i neexistenci více sloupcu tabulky se stejným druhem obsahu:

$$\left.\begin{array}{r}(\text{manager, podřízený1, podřízený2, podřízený3})\\(\text{manager, podřízení-vice_hodnot_v_jednom_sloupci})\end{array}\right\} \rightarrow (\text{manager, podřízený})$$

Druhá normální forma (2NF)

Neexistují částečné závislosti neklíčových atributů na klíči, tj. platí
$\forall x \in$ NK $\exists$KK : KK $\rightarrow$ x
(kde NK je množina neklíčových atributů a KK je podmnožina nějakého klíče)



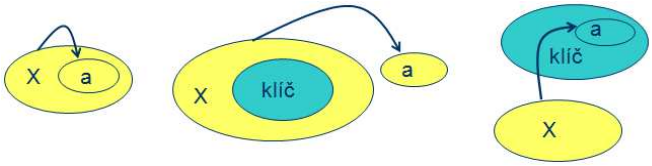
Existuje klíc a všechna neklícová pole jsou funkcí celého klíce (a tedy ne jen jeho částí).

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

Třetí normální forma (3NF)

Žádný neklíčový atribut není tranzitivně závislý na žádném klíči. Přesněji, v daném schématu R(A, F) platí alespoň jedna z podmínek pro každou závislost X $\rightarrow$ *a* (kde X $\subseteq$ A, *a* $\in$ A):

- závislost je triviální
- X je nadklíč
- *a* je částí klíče

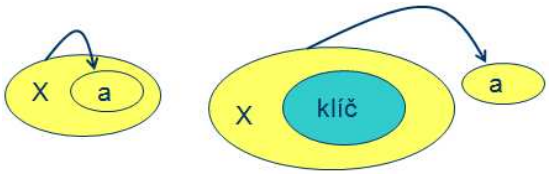


Boyce-Coddova normální forma (BCFN)

Každý atribut je závislý na klíči. Přesněji, v daném schématu R(A, F) platí alespoň jedna z podmínek pro každou závislost X $\rightarrow$ *a* (kde X $\subseteq$ A, *a* $\in$ A):

- závislost je triviální
- X je nadklíč

Stejně jako 3NF bez poslední možnosti (*a* je součást klíče).



Algoritmy návrhu schémat relací

Schémata relací by mela být navrhována tak, aby odpovídala predem pripravenému konceptuálnímu modelu (napr. pomocí ER diagramu) a zároven pokud možno splnovala co nejprísnejší požadavky na normální formy. Pro modelování relacní databáze existují dva prístupy:

- Získání množiny relacních schémat (rucne nebo prevodem z napr. ER diagramu) a provádení normalizace pro každou tabulku zvlášt. Hrozí riziko prílišného rozdrobení databáze na velký pocet malých tabulek (a nadbytecný i vzhledem k požadované normální forme).
- Návrh tzv. **univerzálního schématu databáze** – jedna velká tabulka pro celou databázi (vc. platných funkcních závislostí) a normalizace provádená globálne. Entity jednotlivých relací jsou „vypozorovány“ jako efekt funkcních závislostí, což není príliš pruhledné a jednoduše proveditelné, ale minimalizuje to šanci na rozdrobení databáze

Oba prístupy lze také zkombinovat – prevést ER model databáze do schémat a nekterá (nebo až všechna) potom pred normalizací sloucit.

Základní definice

Uzávěr F^+ množiny FZ F (funkční uzávěr) je množina všech FZ odvoditelných Armstrongovými pravidly z FZ v F.

Pokrytí množiny FZ F je libovolná množina FZ G taková,
že $F^+ = G^+$.

Kanonické pokrytí = pokrytí tvorené elementárními FZ.

Závislost f je **redundantní** v F, jestliže platí $(F - \{f\})^+ = F^+$, tj. f lze odvodit.

Neredundantní pokrytí F je pokrytí F po odstranění všech redundantních závislostí.

Uzávěr množiny atributů X^+ vzhledem k F je množina všech atributů funkčně závislých na X. (Důsledek – pokud $X^+ = A$, potom X je nadklíč).

Pokud obsahuje F závislost $X \rightarrow Y$ a v X existuje atribut *a* takový, že $(X - a)^+$, nazýváme *a* **atributem redundantním v $X \rightarrow Y$**.

Redukovaná FZ je taková, která na levé straně neobsahuje žádné redundantní atributy (jinak je **částečná**).

Neredundantní kanonické pokrytí, které se skládá pouze z redukovaných FZ se nazývá **minimální pokrytí**.

Základní algoritmy pro analýzu schémat

Algoritmus atributového uzávěru

Atributový uzávěr množiny atributů X vůči množině závislostí F. Princip: postupně odvozujeme všechny atributy „F-určené“ atributy z X. Polynomiální složitost (O(m*n), kde n je počet atributů a m počet závislostí).

$F = \{a \rightarrow b,\ bc \rightarrow d,\ bd \rightarrow a\}$

$\{b,c\}^+ = ?$

1. ClosureX := {b,c} (inicializace)
2. ClosureX := ClosureX ∪ {d} = {b,c,d} ($bc \rightarrow d$)
3. ClosureX := ClosureX ∪ {a} = {a,b,c,d} ($bd \rightarrow a$)

$\{b,c\}^+ = \{a,b,c,d\}$

Algoritmus příslušnosti

Často potřebujeme zjistit příslušnost nějaké závislosti $X \rightarrow Y$ do F^+, tj. vyřešit problém $\{X \rightarrow Y\} \in F^+$. Počítat celý F^+ je nepraktické, lze použít algoritmus atributového uzávěru: pokud $Y \in$ *AttributeClosure*(F, X) potom je zavislost v uzávěru.

Testování redundancí

Algoritmus příslušnosti lze jednoduše použít k testu redundance závislosti $X \rightarrow Y$ v F a k testu redundance atributu v X (vzhledem k F a $X \rightarrow Y$).

```
algorithm IsDependencyRedundant(set of dependencies F, dependency X → Y ∈ F)
    return IsDependencyInClosure(F – {X → Y}, X → Y);

algorithm IsAttributeRedundant(set of deps. F, dep. X → Y ∈ F, attribute a ∈ X)
    return IsDependencyInClosure(F, X – {a} → Y);
```

Algoritmus vracející k FZ redukovanou levou stranu

Bude užitečný jako podprocedura v dalších algoritmech.

```
algorithm GetReducedAttributes(set of deps. F, dep. X → Y ∈ F)
    X' := X;
    for each a ∈ X do
        if IsAttributeRedundant(F, X' → Y, a) then X' := X' – {a};
    endfor
return X';
```

Minimální pokrytí

Použijeme postupně na všechny FZ testy redundance a ty odstraníme.

```
algorithm GetMinimumCover(set of dependencies F): returns minimal cover G
    decompose each dependency in F into elementary ones
    for each X → Y in F do
        F :=        (F –
                    {X → Y}) ∪
                    {GetReducedAttributes(F, X → Y) → Y};
    endfor
    for each X → Y in F do
        if IsDependencyRedundant(F, X → Y) then F := F – {X → Y};
    endfor
return F;
```

Nalezení (prvního) klíče

Algoritmus testu redundance atributu lze přímo použít při hledání klíčů. Postupně se odstraňují redundantní atributy z A → A. Klíčů samozřejmě může být víc, algoritmus najde jen jeden (který – to záleží na pořadí procházení množiny atributů uvnitř algoritmu GetReducedAttributes().

```
algorithm GetFirstKey(set of deps. F, set of attributes A) : returns a key K;
    return GetReducedAttributes(F, A → A);
```

Nalezení všech klíčů

K již nalezenému klíči hledáme ekvivalentní množiny atributů, tj. jiné klíče. NP-úplný problém (teoreticky exponenciální počet klíčů/závislostí).

```
algorithm GetAllKeys(set of deps. F, set of attributes A) : returns set of all keys Keys;
    let all dependencies in F be non-trivial, i.e. replace every X → Y by X → (Y – X)
    K := GetFirstKey(F, A);
    Keys := {K};
    Done := false;
    while Done = false do
        Done := true;
        for each X → Y ∈ F do
            if (Y ∩ K ≠ ∅ and ¬∃K' ∈ Keys : K' ⊆ (K ∪ X) – Y) then
                K := GetReducedAttributes(F, ((K ∪ X) – Y) → A);
                Keys := Keys ∪ {K};
                Done := false;
        endfor
    endwhile
return Keys;
```

Testování normálních forem

NP-úplný problém kvůli nutnosti znát všechny klíče. Stačí otestovat jen FZ z F, nemusím testovat celý F^+.

Normalizace

Jediným zpusobem, jak u nejakého obecného relacního schématu dosáhnout normální formy (obecne se požaduje vetšinou 3NF nebo BCNF), je rozdelení na několik podschémat. Dá se to provést rucne nebo algoritmicky a existuje více prístupu podle požadavku na normální formu:

- **Bezztrátovost -** dekompozice relace R(A, F) do R1(A1, F1) a R2(A2, F2) je bezeztrátová, když $A1 \cap A2 \rightarrow A1$ nebo $A1 \cap A2 \rightarrow A2$, tedy opetovným spojením do puvodní relace nevzniknou další rádky.
- **Pokrytí závislostí -** dekompozice R(A, F) do R1(A1, F1) a R2(A2, F2) zachovává pokrytí závislostí, když $F^+ = F_1^+ \cup F_2^+$ – nesmí se ztratit závislost ani v rámci dílcího schématu, ani jdoucí napríc schématy.

Algoritmus „Syntéza“

Algoritmus pro dekompozici do 3NF, zachovávající pokrytí závislostí. Základní verze nezachovává bezztrátovost. Lze ji zajistit pridáním dalšího schématu do dekompozice, které obsahuje univerzální klíč (tj. nejaký klíč puvodního univerzálního schématu). Schéma v dekompozici, které je podmnožinou jiného mužu vypustit. Mužu se pokusit sloucit schémata s funkcne ekvivalentními klíči, ale tato operace muže obecne porušit 3NF!! (nebo BCNF pokud jí bylo dosaženo)

algorithm ***Synthesis***(set of elem. deps. F, set of attributes A) : **returns set** {$R_i(F_i, A_i)$}
    create minimal cover from F into G
    compose FDs having equal left side into a single FD
    every composed FD forms a scheme R_i (A_i, F_i) of decomposition
return $\cup_{i=1..n}\{R_i (A_i, F_i)\}$

Priklad

Contracts(A, F)
A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId, q = Quantity, v = Value}
F = {c → sjdpqv, sd → p, p → d, jp → c, j → s}

Minimální pokrytí:
V závislostech z F nejsou redundantní atributy. Byly vyrazeny redundantní FZ c → s a c → p.
G = {c → j, c → d, c → q, c → v, sd → p, p → d, jp → c, j → s}

Kompozice podle levých stran:
G' = {c → jdqv, sd → p, p → d, jp → c, j → s}

Výsledek:
R_1({cqjdv}, {c → jdqv}), R_2({sdp}, {sd → p}), R_3({~~pd~~}, {~~p → d~~}), R_4({jpc}, {jp → c}), R_5({js}, {j → s}))
(podmnožina v R_3)

Ekvivalentní klíče: {c, jp, jd}
R_1({cqjpdv}, {c → jdqv, jp → c}), R_2({sdp}, {sd → p, p → d}), R_5({js}, {j → s}))
sloucení R_1 a R_4
(nyní ale p → d porušuje BCNF)

Algoritmus „Dekompozice“

Dekompozice je algoritmus, který relacní schéma prevede do Boyce-Coddovy normální formy. Zarucuje zachování bezeztrátovosti, ale už ne pokrytí závislostí (bez ohledu na algoritmus toto u BCNF nekdy není možné). Jeho beh vypadá následovne:

1. Vyber nejaké schéma, které není v BCNF.
2. Vezmi pro nej neklícovou závislost X -> Y (tak že X není klíc) a dekomponuj podle ní – vyhod ze schématu Y a dej XY do zvláštní tabulky.
3. Opakuj od kroku 1, dokud existuje schéma, které není v BCNF.

Modely životního cyklu softwarových systému

Životní cyklus programového díla

- Nápad
- Neformální specifikace (odborný článek, úvodní studie)
- Formální specifikace (analýza)
- Dekompozice (návrh)
- Řešení komponent (modulů)
- Implementace komponent
- Testování komponent
- Integrace komponent do celku
- Testování celku (akceptační test)
- Instalace
- Provoz a údržba

Fáze tvorby SW produktu

- **Úvodní studie** (informační plánování, feasibility study) - podklady pro rozhodnutí, zda vůbec má projekt smysl
- **Analýza** - vytvoření dostatečně přesné specifikace produktu
- **Návrh (design)** - dekompozice systému na komponenty, které lze naprogramovat
- **Implementace** - realizace komponent definovaných v návrhu a jejich sestavení do výsledného produktu

Každý produkt by měl projít všemi fázemi životního cyklu. Ze statistik (pro velké systémy – řádově stovky tisíc řádků kódu) vyplývá, že úsilí věnované těmto fázím by mělo být rozděleno zhruba v poměru: **analýza 40%**, **návrh 40%**,**implementace 20%**. Pokud nevěnujeme počátečním fázím dostatečné úsilí, projeví se to zvýšenými nároky při implementaci či údržbě systému.

Modely životního cyklu

Pro standardizaci postupů jsou zavedeny typové modely životních cyklů, např.:

- **Model vodopád (Waterfall)**
- **Model průzkumník**
- **Spirálový model**
- **Přírůstkový model**

Model životního cyklu určuje základní schéma postupu. Životní cyklus by měl vždy začínat dostatečně přesnou specifikací a návrhem. Není nutno realizovat celý systém najednou – naopak přírůstky poskytují uživateli dobrý pocit postupu prací.

Waterfall model

Postupně se prochází přesně definoványmi fázemi:

- **analýza** a definice požadavků
- **design** systému a softwaru
- **implementace** a unit testing
- **integrace** a testování systému
- **provoz** a údržba

Nevýhody - neflexibilní (teoreticky by se nemělo zpětně zasahovat do ukončených fází). Na zacatku se neda udelat 100% dobra analyza pozadavku!

Výhody - v každé fázi je dobře definováno co se bude dělat a z čeho vycházet. Snazší pro management (uz na začátku se dá rozumně plánovat, ví se co se bude dít). Dá se očekávat robustní návrh systému. Model je vhodny pro změnové rizeni (male zmeny alebo buxfixy).

Evolutionary development

Idea je spociva v udělani jednoduche verzi systému, která bude následně zpřesňována spolu s požadavky uživatele. 2 typy:

- **Exploratory development** - snahou je spolu s uživatelem postupně prozkoumávat požadavky a postupovat směrem k cílovému systému (vždy se udělá to čemu všichni dobře rozumí).
- **Throw-away prototyping** - narozdíl od předchozího se tady zaměřujeme spíš na špatně specifikované části systému a ty se snažíme pomocí prototypů lépe definovat.

Nevýhody - špatně se kontroluje průběh procesu. Casto vznikne špatně strukturovaný systém (časté opravy a zásahy do předchozích částí, aby mohly být ty nově definované dodělány). Casto jsou potřeba speciální nástroje.

Výhody - vyplatí se zejména u menších projektů (míň než 100k lines of code).

Formal systems development

Požadavky a specifikace jsou detailně popsány v některém formálním jazyku. Další kroky jsou nahrazeny postupnými transformacemi až do podoby spustitelného kódu (postupně jsou tedy přidávány detaily které zpřesňjí specifikaci a zároveň neporušují její predikáty).

Výhody - jednotlivé malé kroky jsou snadno vystopovatelné. "Snadno" se dá dokázat správnost programu.

Nevýhody - potřeba speciálních skillů (rozhodnot o následující transformaci je složité).

Re-use oriented development

- Analýza komponent
- Úprava požadavků podle definovaných komponent
- Design systému s využitím komponent
- Vývoj ostatních potřebných částí a jejich integrace

Výhody - redukce kódu který se bude psát (=snižování rizika chyby).

Nevýhody - výsledek nemusí 100% odpovídat výchozím požadavkům. V případě využití COTS (Commercial Off-The-Shelf) komponent vzniká závislost na dalším vendorovi .

Incremental development

Myšlenka je umožnit uživateli učinit některá potřebná rozhodnutí až v době kdy bude mít jasno:

- Definice požadovaných služeb systému, přidělení priorit.
- Definice počtu jednotlivých inkrementů, rozdělení jednotlivých služeb k inkrementům které se budou dělat.
- V každém inkrementu se pak detailně dospecifikují služby které se budou dělat a proběhne jejich vývoj, výsledek je nasazen k uživateli.

V každém inkrementu se může na danou část použít jiná metodika vývoje. V dalších fázích jsou pak nově vzniklé inkrementy integrovány do stávajícího systému. Z této metody vychází extreme programming (kde jsou dané myšlenky dotažené do extrému ;).

Výhody - zákazník nemusí čekat až do konce, než bude moct aspoň něco používat. Zákazník může uplatnit u pozdějších inkrementů experience, kterou nabyde používáním těch předchozích. Menší riziko celkového selhání. Nejdůležitější části systému jsou nejlépe otestované (byly vytvořené v ranných inkrementecha a s každým přírůstkem testovány znovu a znovu).

Nevýhody - inkrementy mají být malé a může být složité na ně namapovat požadavky customera. Některé komponenty budou využívány mnoha částmi systému a zpočátku nemusí být přesně vidět jejich všechny požadované vlastnosti.

Spiral development

Místo popisu procesu jako sekvence ativit s backtrackingem zpět je využito spirály. Každá jedna otočka spirály se skládá ze čtyř sektorů:

- Stanovení cílů aktuální smyčky - identifikace omezení a rizik, stanovení plánů.
- Ohodnocení a redukce rizik (definovaných dříve).
- Vývoj a validace - výběr metody v závislostí na rizicích (pokud je riziko třeba špatné UI, budeme prototypovat).
- Plánování - revize otočky spirály a plány co dál.

Na rozdíl od ostatních se v tomto modelu explicitně pracuje s riziky.

Plánování a rízení projektu, alokace zdroju, použití metrik, rízení kvality, stupne zralosti softwarových týmu (CMM)

Plánování projektu

Co je to projekt?

*„Projekt je dobře definovaná **posloupnost** č**inností**, která má určen začátek a konec, je zaměřena na dosažení určitého cíle a je uskutečňována pomocí **zdroj**ů – lidí, věcí, nástrojů a prostředků“. „Projekt je specifická **nerutinní** akce, která proto vyžaduje plánování.“ „Čím **složitější** je projekt, tím více vyžaduje plánování.“*

Co je cílem plánování?

Plány nejsou vytvářeny proto, aby se dodržovaly (i když se na to dají také použít). Plánování je nutné proto, abychom si rozmysleli, co je třeba udělat. Plánování je zaměřeno na odstranění nejistoty. Plánování se využívá pro sestavení odhadů harmonogramu a rozpočtu. Plánování je nástrojem pro řízení projektů. **Plánování projektů je řešení problému: „Jak dlouho a za kolik“ Metodou „divide et impera“.**

Typy plánování

- **Strategické plánování** - dlouhodobé – řádově desítky let a více.
- **Taktické plánování** - střednědobé – řádově měsíce až roky.
- **Operační plánování** - krátkodobé – řádově hodiny až měsíce.

Strategické plánování definuje potřebné projekty. Plán jejich realizace vzniká taktickým plánováním, kde je projekt rozdělen na úlohy. Řešení úloh přiřazenými zdroji je věcí operativního plánování. Plánování projektů je zaměřeno na realizaci – podrobný popis dílčích činností na projektu, jejich vzájemné vztahy a rozvržení času, zdrojů a prostředků na tyto dílčí činnosti. Příklad nástroje pro vytváření plánů projektů a sledování jejich realizace je MS Project.

Elementy plánu projektu

- **úlohy** (tasks) - vyžadují čas a zdroje.
- **milníky** (milestones) - kontrolní okamžiky. Ukončení nějaké podfáze projektu, vznikají při něm výstupy pro management.
- **zdroje** (resources) - lidé, vybavení, místa.
- **vztahy** (dependencies, assignments) - co dříve, co později, kdo-co udělá.
- **Deliverable** - výstup který se předává zákazníkovi (specifikace, design, ..). Deliverable je milestone, ale milestone nemusí být deliverable (u milestone totiž může jít o nějaké interní dokumenty).

Proces plánování projektu

- **Vytvoření** a uspořádání plánu projektu – rozdělení plánové činnosti na jednodušší elementy
- **Rozvržení úloh** – analýza vztahů mezi elementy
- **Doplnění zdrojů** (lidí, zařízení, prostředků) do projektu – odhad potřeb
- **Přiřazení nákladů** úlohám a zdrojům – porovnání se skutečnými možnostmi
- **Úpravy a ladění** plánu (až vznikne směrný plán)
- **Monitorování** a sledování plánu
- **Uzávěrka** projektu (úschova plánu pro budoucnost)

Rozvrhování úloh

První rozvrh je vždycky moc optimistický, je proto třeba ho průběžně aktualizovat. Jde hlavně o rozpad jednotlivých prací na aktivity a jejich rozvrhnutí tak, aby ty co mohou běželi paralelně a naopak navazovaly ty které jsou závislé. Důležitý není jenom čas, je potřeba zajistit taky dostatek prostředků (ať už jde o lidi, nebo třeba místo na disku).

Existují různé typy vazeb mezi úlohami

FS (Finish to Start) – další úloha může začít, až když předchozí úloha skončí, **SS (Start to Start)** – další úloha začne současně s předchozí úlohou, **FF (Finish to Finish)** – další úloha skončí současně s předchozí úlohou, **SF (Start to Finish)** – další úloha musí skončit, aby mohla předchozí úloha začít.

Nástroje

- **Activity graph** - graf návaznosti aktivit (task = čtvereček, milestone = kolečko, závislost = hrana).
- **Kritická cesta** - nejdelší cesta ze startu do cíle, její natáhnutí znamená natáhnutí projektu. Kritická úloha je úloha ležící na kritické cestě. Změna doby trvání kritické úlohy ovlivňuje dobu trvání projektu.
- **Gannt chart (aka activity bar chart)** - horizontální osa ukazuje čas, sloupečky pak jednotlivé tasky popisují stejnou informaci jako acivity graph a dají se v nich líp znázornit rezervy tasků.

Náklady na projekt

Cena projektu = Σ cen úloh. Cena úlohy = fixní náklady + cena za použití zdrojů. Cena za použití zdroje = odměna za práci v normální pracovní době + odměna za práci přesčas + fixní náklady na použití zdroje. Práce = jednotky * délka. Práce je dána součinem počtu jednotek zdroje, které na úloze pracují a délky úlohy.

Další život plánu projektu

Úpravy plánu tak dlouho, až dojde k jeho schválení. Nastaví se směrný plán. Manažer projektu dostane projekt k realizaci. Zjistí a najme si zdroje. Aktualizuje plán dle skutečnosti. Uzavře projekt a vyhodnotí. Plán se uloží do archivu firmy.

Řízení projektu

Project management - je iterativní proces, který končí až končí vlastní projekt. Vše by se dalo popsat následujícím pseudokódem:

```
Establish the project constraints
Make initial assesments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
  Draw up project schedule
  Initiate activities according to schedule
  Wait ( for a while )
  Review project progress
  Revise estimates of project parameters
  Update the project schedule
  Renegotiate project constraints and deliverables
  if ( problems arise ) then
    Initiate technical review and possible revisions
  end if
end loop
```

Řízení rizik

Riziko = pravděpodobnost že nastane některá nežádoucí událost.

Základní rozdělení rizik

- **Project risks** - rizika ohrožující plán či zdroje projektu
- **Product risks** - rizika ohrožující výkon nebo kvality vyvíjeného produktu
- **Business risks** - rizika ohrožující organizaci nebo vlastní vznik softwaru

Risk management - je opět iterativní proces, který se musí dít v průběhu projektu, nejen na začátku. Základní kroky risk managementu:

- **Risk identification** - identifikace možných rizik (jako: technology risks, people risks, organisational risks, tools risks, req. risks, estimation risks)
- **Risk analysis** - zhodnocena pravděpodobnost a dopad jednotlivých rizik. Nejde o přesná čísla, ale spíš rozsah psti (jako: very low .. < 10%, low .. 10 - 25%, moderate .. 25 - 50%, high .. 50 - 75%, very high .. > 75%). U dopadu podobně (jako: catastrophic, serious, tolerable nebo insignificant). Výsledkem je prioritizovaná (a průběžně aktualizovaná) **tabulka rizik**.
- **Risk planning** - výběr strategie řízení každého z rizik. **Minimalizace pravdepodobnosti**. **Minimalizace dopadu**. **Přijmutí rizika** (když nastane, budu se s tím umět vyrovnat). **Přesun rizika** na někoho jiného (pojištění).
- **Risk monitoring**

Alokace zdroju

Když máme připraven rozpad práce na tasky, je potřeba přiřadit zdroje nutné k vykonání daného tasku. Tzn. třeba i lidi (i když označovat je za zdroje není moc hezké ;) Ve velkých společnostech může být problém s pracovníky kteří jsou specialisti - prodloužení jeho práce na jiném projektu může zasáhnout i do našich plánů. Ke znázornění se hodí výše popsané activity graphs a gant charts.

Použití metrik

Cílem je získat pro **nějaký atribut** sw produktu **numerickou hodnotu**. Porovnávaním takových hodnot se dá sledovat např. vývoj kvality. Problém je, že neexistují žádné obecně uznávané standardy a tedy ani rozšířené tooly.

Příklady: počet řádek kódu, počet reportovaných chyb ve výsledném produktu, počet člověko-dní potřebných k vývoji komponenty. **CO**nstructive **CO**st **MO**del - slouží k odhadu počtu člověko-měsíců, které bude vývoj sw. produktu trvat.

- **Control metrics** - metriky asociované s procesem. Např. jak dlouho trvá odstranění nalezeného defektu.
- **Predictor metrics** - metriky asociované s produktem. Např. počet atributů a operací třídy, nebo průměrná délka identifikátorů.

Extrení vlastnosti software se nedají přímo měřit. Jde třeba o komplexnost či pochopitelnost. Proto je snaha najít vztah mezi vnitřními (velikost software) a vnějšími metrikami. K tomu je třeba: přesné měření interního atributu, existence vztahu mezi tím co umíme změřit a ext. atributem, pochopení daného vztahu a jeho vyjádření v podobě vzorce nebo modelu.

Proces měření – pozostává z několika fáz:

- Výběr měření která se budou provádět - je třeba vědět co chci změřit.
- Výběr komponent k posouzení.
- Měření charakteristik komponent.
- Identifikace měření vybočujících z řady.
- Analýza vybočujících komponent.

Klíčové ovšem je naměřená data zaznamenávat a mít k dispozici při dalších měřeních.

Metriky produktů

Dají se rozdělit na:

- **Dynamicke metriky** - měřené na běžícím programu.
- **Statické metriky** - měření nějaké reprezentace systému (program, dokumentace).

Příklady: **Fan-in** - počet volání fce X (hodně znamená že je fce úzce svázaná se zbytkem). **Fan-out** - počet fcí které jsou z fce X volané. **Lenght of code** - větší kód = komplexnější = náchylnější k chybám. **Cyclomatic complexity** - měří komplexnost programu. **Lenght of identifiers** - delší = víc samovysvětlující = vyšší čitelnost. **Depth of conditional nesting** - větší hloubka = horší srozumitelnost. **Fog index** - čím delší slova a věty v dokumentu, tím je méně srozumitelný (třeba pro dokumentace).

Řízení kvality

- **Quality assurance** - zřízení frameworku organizačních procedur, které vedou k vysoké kvalitě.
- **Quality planning** - výběr patřičných procedur z daného frameworku a jejich adaptace pro specifický projekt.
- **Quality control** - definice a ustanovení procesů, které zaručují, že jsou procedury a standardy dodržovány vývojovými teamy.

O quality management by se měl starat nezávislý team. Quality management by měl být oddělen od project managementu. Neměl by být tedy závislý na nějakých schedulech a budgetech. Ke kvalitě existuje sada standardů ISO 9000 (ISO 9000-3 se vztahuje přímo k sw. vývoji). ISO 9000 -> jeho instancí je Organisation quality process -> jeho instancí je Project quality management.

Dva typy QA standardů:

- **Product standards**
- **Process standards**

Casto jsou standardy považovány za zbytečnou byrokracii (a každý se snaží najít důvod proč nejsou v tom kterém projektu potřeba). Jak lidi přesvědčit?

- Zapojit je do vývoje standardů - musí rozumět motivaci
- Průběžně standardy aktualizovat podle aktuálních potřeb
- Poskytnout tooly pro podporu

Standardy dokumentace jsou důležité, protože dokumentace je jediná hmotná součást vyvíjeného software. Jde o:

- document process standards - proces v němž dokument vzniká
- document standards - definují strukturu dokumentů
- document interchange standards - zaručují kompatibilitu dokumentů

Quality planning - musí definovat co přesně vysoká kvalita znamená (což je u sw někdy složité - viz požadavek na dobrou udržovatelnost). Problém je, že výroba sw. je spíš kreativní proces, takže hodně záleží na tom kdo to dělá. Quality plan by měl být as short as possible (jinak ho nikdo nebude číst). Je hodně parametrů které je možné optimalizovat (jako: safety, robustness, modularity, portability, usability, efficiency, lernability).

Quality control - je potřeba kontrolovat dodržování stanovených standardů. Deje se pomoci:

- **quality reviews** - QA team kontroluje vybraný proces či produkt.
- **automated software assessment** - automatická kontrola metrik.

Stupne zralosti softwarových týmu (CMM)

Process improvement je porozumění stávajicímu procesu a jeho změna k lepšímu a má následující klíčové fáze:

- process analysis
- improvement identification
- proces change introduction
- process change training
- change tuning

Proces má podobně jako produkt **charakteristiky**, např:

- understandability
- visibility
- reliability
- maintainability

Při snaze o zlepšování procesu je klíčové zjistit, **co a jak měřit** - GQM paradigm:

- Goals
- Questions
- Metrics

The Capability Maturity Model Integration

Jedná se o soubor pravidel, požadavků a doporučení, které mají splňovat firemní procesy a co je třeba dodržovat, aby procesy vývoje byly efektivní, účinné a spolehlivé. Důležitou charakteristikou modelu, která jej odlišuje od norem primárně zaměřených na procesy a kvalitu výroby je právě zaměření na procesy vývoje.

Zásadní odlišností CMMI od jiných norem jakosti je, že se nejedná o model, který organizace zavede, získá „certifikát“ a dál už může jen udržovat a interně vylepšovat. CMMI je nástroj, který pomáhá k neustálému zlepšování úrovně řízení a vývoje.

Firmy jsou v rámci CMMI vedeny doporučeními organizovanými buď do 5 úrovní (maturity levels), v rámci kterých se postupně zavádí činnosti plnící jakostní cíle, nebo do tzv. kontinuálního modelu, ve kterém se sledují cíle, které si firma zvolí a postupně zlepšuje jejich plnění.

Vyvinut by Software Engineering Institute (na Carnegie-Mellon University, financuje to americké ministerstvo obrany). Původní záměr bylo mít prostředky k zjištění schopností potenciálního contractora, který bude něco dělat pro US DoD. Klasifikace sw. procesů na 5 úrovní:

Úrovně CMM

- **Počáteční (Initial level)** - procesy se organizují náhodně.
- **Řízená (Repeatable level)** - definované procesy řízení projektů.
- **Definovaná (Defined level)** - standardiazace procesů řízení.
- **Řízená (Managed level)** - kvantitativní měření.
- **Optimalizující (Optimising level)** - trvalé zlepšování procesů.

1. Počáteční úroveň (initial)

- Softwarové procesy jsou náhodné a chaotické.
- Organizace na této úrovni zralosti nemají stabilní prostředí pro vývoj a údržbu software, reagují pouze na vzniklé problémy.
- Schopnost softwarových procesů na úrovni 1 nelze předpovídat, protože se procesy neustále mění.
- Schopnost softwarových procesů na úrovni 1 závisí na schopnostech jednotlivců a liší se s jejich znalostmi a kvalifikací.

2. Opakovatelná úrov. (repeatable)

- Organizace na této úrovni zralosti mají definovány procesy řízení projektů a vybudovány procedury pro jejich implementaci. Plánování a řízení nových projektů je založeno na zkušenostech s podobnými projekty.
- Cílem úrovně 2 je zavedení efektivního řízení softwarových procesů. Efektivní proces je dokumentovaný, vyžadovaný, měřitelný, lidé jsou pro něj vyškoleni a proces je možné zlepšovat. Cíle projektu jsou realistické, jsou založeny na výsledcích předcházejících projektů a požadavcích současného projektu. Manažeři sledují náklady, plán a funkcionalitu, problémy jsou identifikovány hned při vzniku, jsou definovány projektové standardy a dohlíží se na jejich dodržování.
- Schopnost softwarových procesů na úrovni 2 může být definována jako **disciplinovaná**.

3. Definovaná úroveň (defined)

- Organizace na této úrovni zralosti má definovány, dokumentovány a standardizovány procesy pro řízení i inženýrské činnosti, které jsou navzájem integrovány v rámci celé organizace.
- Je definován **„standardní softwarový proces organizace“**, který je potom přizpůsobován na podmínky jednotlivých projektů.
- Tím vzniká **„softwarový proces projektu“**.
- Zaměstnanci i manažeři jsou vyškoleni pro plnění rolí v procesech.
- Schopnost softwarových procesů na úrovni 3 může být definována jako standardní a konzistentní, protože jak činnosti inženýrské, tak činnosti řízení jsou stabilní a opakované, tj. proces je **definovaný**.

4. Řízená úroveň (managed)

- Organizace na této úrovni zralosti má stanoveny detailní **metriky** softwarových procesů a kvality produktu.
- Data z projektů se uchovávají v databázi a analyzují.
- Kontrola projektů je zajištěna odhalováním odchylek od definovaných mezí.
- Schopnost softwarových procesů na úrovni 4 může být definována jako predikovatelná, protože **proces je měřitelný** a operuje v rámci měřitelných mezí.

5. Optimalizovaná (optimizing)

- Organizace na této úrovni zralosti disponuje prostředím pro kontinuální zlepšování procesů. Organizace identifikuje slabé a silné stránky procesů předem ve snaze zabránit vzniku problémů. Data o efektivnosti softwarových procesů jsou používána pro analýzy přínosů nových technologií a navrhovaných změn softwarových procesů v organizaci.
- Jsou identifikovány inovace, které využívají nejlepší praktiky softwarového inženýrství a jsou přenášeny do celé organizace.
- Týmy na úrovni 5 analyzují defekty, aby určily jejich příčiny.
- Softwarové procesy jsou vyhodnocovány, aby se předešlo defektům, které nastaly v jiných projektech. Schopnost softwarových procesů na úrovni 5 může být definována jako **kontinuálně se zlepšující**.

Problemy CMM

Dá se dobře aplikovat na velkých organizacích (pro malé organizace může být už příliš byrokratický).

Tento model má ale i své nevýhody: zaměřuje se **výhradně na project management**, ne na product development (nezohledňuje možnosti jako prototyping, formální metody ..). Vůbec se **nezabývá analýzou rizik**. Není dobře popsáno v kterých typech organizací se dá CMM využít a kde už moc ne. Organizace která splňuje 80% levelu 2 a 70% levelu 3 dostane pořád level 1 rating (= k postupu je potřeba splnit 100%).

Při srovnání s ISO 9000 se dá říct, že organizace na levelu 2 až 3 jej splňují, ale dá se najít i organizace na levelu 1 která ISOu odpovídá. Přesto mnoho key areas s ISOem koresponduje.

CASE systémy

Computer **A**ided **S**oftware **E**ngineering – je počítačová podpora softwaroveho inzenyrstvi. CASE nástroje pomáhají zlepšit produktivitu práce tím, že nasledujici činnosti a potřeby automatizují a podporují:

- Modely softwarových systémů jsou příliš složité
  - podpora úrovní abstrakce, různých pohledů
  - nutnost rozdělení mezi vývojáře
- Iterativní vývoj vs. konzistence produktu
  - sběr požadavků
  - návrh systému
  - údržba
- Standardizace používaných postupů

Rozdeleni CASE systemu

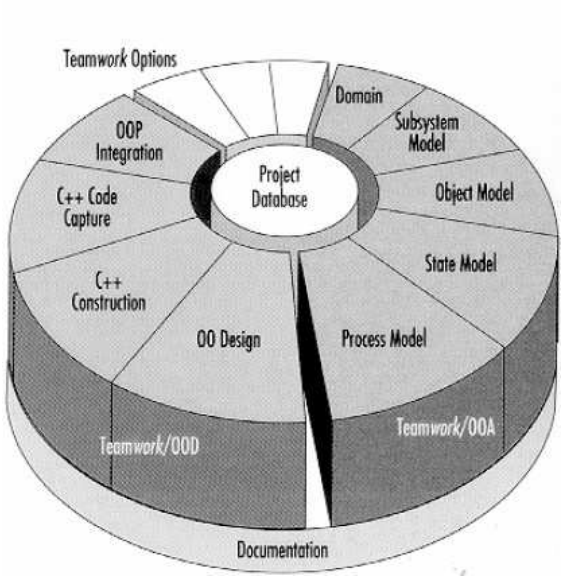
Podle stupne integrace

- **Nižší CASE** – podpora ***tvorby*** software. Návrhy obrazovek, formulářů, menu. Jazyková podpora.
- **Vyšší CASE** – podpora **analýzy** a návrhu; Tvorba diagramů a modelů. Kontrola konzistence modelu. *Příklad*: AxiomSys: strukturovaná analýza.
- **Integrované CASE** – podpora **životniho cyklu**. Od analýzy po generování kódu. Round-trip engineering. Podpora tvorby dokumentace. *Příklad*: Oracle Designer.
- **Komponentové CASE** – otevřenost. Repository s rozhraním (⇒ SCM, testování). Integrace nástrojů od různých výrobců. *Příklad*: Rational Suite Enterprise.

Podle rozsahu možností

- **Jedna fáze** – podpora jedné fáze životního cyklu (analýza, prog.).
- **Jedna metodika** – podpora dané metodiky přes cely zivotni cyklus.
- **Více fází, více metodik** – transformace modelů, vlastní metodiky.
- **Vývoj + management** – včetně podpůrných funkcí pro řízení

STRUKTURA CASE



Repository (databáze projektu) - pracuje s **metamodelem** (typy objektů a vazeb), **modelem systému** (objekty, vazby), s **prototypy a kódem** (návrhy obrazovek apod.).

Nástroje – stoja kolem repository. Prodporuji tvobnu modelu, kontrolu konzistence, generovani reportu, dokumentace a kodu.

Vyvoj pomoci CASE

- ***Analýza* systému** − kreslení modelů, datový slovník. Specifikace požadavků.
- ***Návrh*** − diagramy struktury, nasazení. Návrh uživatelského rozhraní, databáze.
- ***Implementace*** − návrhy obrazovek, fyzický model databáze. Generování kódu, round-trip inženýrství. Testování.
- ***Údržba*** − reverzní inženýrství, reuse.
- ***Dokumentace*** − požadavky, design systému.

Rizeni projektu

- ***Plánování*** − odhady pracnosti, rozdělení práce
- ***Konfigurační management*** − rozdělení modelu a synchronizace změn
- ***Sledování* stavu projektu**

Vyhody pouzivani - **Zvýšení produktivity** (automatizace prací ⇒ čas na podstatné věci, lepší přehled o modelu a implementaci, podpora rozdělení práce, snazší údržba dokumentace i systému). **Zvýšení kvality** (podpora analýzy ⇒ lepší znalost požadavků, kontroly konzistence a úplnosti, synchronizace reality a dokumentace, podpora používání standardů).

Je však třeba pocitat i s vysokou cenou CASE nastroju (100000czk). Doba navratnosti investice je po 2. - 3. projektu (je třeba zaskolit lidi). Změna stylu práce – nutnost podridit sa CASE nastroju.

Trívrstvá struktura informacních systému, klient/server

Důvody vzniku vícevrstvých architektur

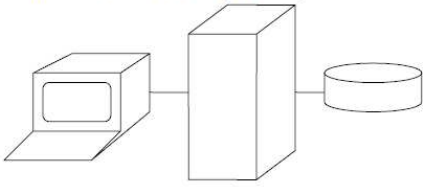
Monolitické aplikace (není rozdělena na více částí, aplikace dělá vše sama - veškeré zpracování dat, vytváří uživatelské rozhraní, komunikuje s uživatelem) se nehodí (tolik) pro distribuované prostředí – pro síť. Pokud např. zpracovává větší objemy dat, musí je mít "u sebe" a tím pádem je nutné přenášet velké objemy dat a zpracovávat je jinde, než kde vnikají a jsou standardně uloženy.

Typy služeb

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

Varianty architektur

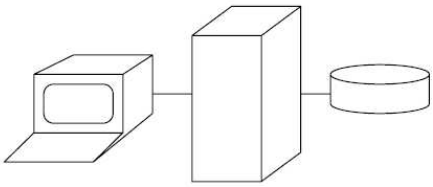
Klient/server se vzdálenými daty



- prezentační služby
- prezentační logika
- logika aplikace
- logika dat
- datové služby
- ovládání souborů

⊖ komunikační zátěž, zatížení stanice

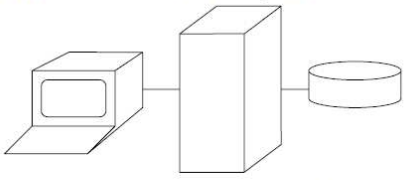
Klient/server se vzdálenou prezentací



- prezentační služby
- prezentační logika
- logika aplikace
- logika dat
- datové služby
- ovládání souborů

⊖ zatížení serveru

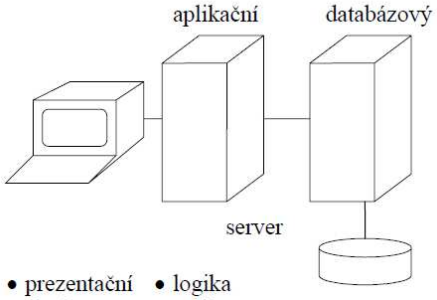
Klient/server s rozdělenou logikou



- prezentační služby
- prezentační logika
- logika aplikace
- logika dat
- logika aplikace
- logika dat
- datové služby
- ovládání souborů

⊕ vyvážená zátěž
⊖ horší rozšiřitelnost

Třívrstvá architektura



- prezentační služby
- prezentační logika
- logika aplikace
- logika dat
- datové služby
- ovládání souborů

⊕ správa aplikace, sdílené objekty aplikací, tenký klient, rozšiřitelnost

Architektura Client/server

Architektura, která popisuje aplikaci, jenž je rozdělená na dvě části: **klientskou** a **serverovou**. Klient (často aplikace s grafickým uživatelským rozhraním) a server, spolu komunikují **přes počítačovou síť**. Model klient/server se stal jedním z hlavních myšlenek síťové technologie. Tento model používá například většina obchodních či firemních aplikací, dále ho pak používají například i tyto internetové protokoly HTTP, SMTP, Telnet, DNS, apod.

Architektura Client/Server obecně

Data se zpracovávají tam, kde se nachází. Výstupy pro uživatele se budou generovat tam, kde se nachází uživatel. Musí dojít k rozdělení původní monolitické aplikace na dvě části: **serverovou část** - zajišťuje zpracování dat a na **klientskou část** - zajišťuje uživatelské rozhraní. Klient a server si posílají data představující **dotazy** a **odpovědi**. Pokud se klient a server dobře dohodnou, mohou účinně minimalizovat objem přenášených dat, mají výrazně menší přenosové nároky a mohou pracovat i v prostředí rozlehlých sítích. Navíc: klient a server **mohou stát na různých platformách**.

Charakteristika klienta

- Aktivní
- Iniciuje komunikaci se servrem zasláním požadavku. Vzájemná komunikace musí být definována nějakým komunikačním protokolem (např. HTTP). Musí být definován formát dat (zpráv, …), které si server a klient vyměňují (např. jazyk HTML).
- Čeká a dostává odpovědi
- Obvykle je připojen k malému počtu serverů najednou
- Obvykle komunikuje přímo s koncovými uživateli, pomocí grafického uživatelského rozhraní

Charakteristika serveru

- Pasivní
- Naslouchá na síti a reaguje na žádosti připojených, autorizovaných klientů
- Při přijetí požadavku jej obslouží
- Může vzdáleně instalovat/odinstalovat aplikace a přenášet data ke klientům

Thick client – většina práce probíhá u klienta, snižuje se režie na přenos po síti, vytíženost serveru, komplikovanější aktualizace SW na všech klientech (např. distribuovaný filesystem).

Thin client – většina práce probíhá na serveru, umožňuje to velmi zjednodušit klienta a jeho zařízení, snadná aktualizace SW (např. klienti jako terminály).

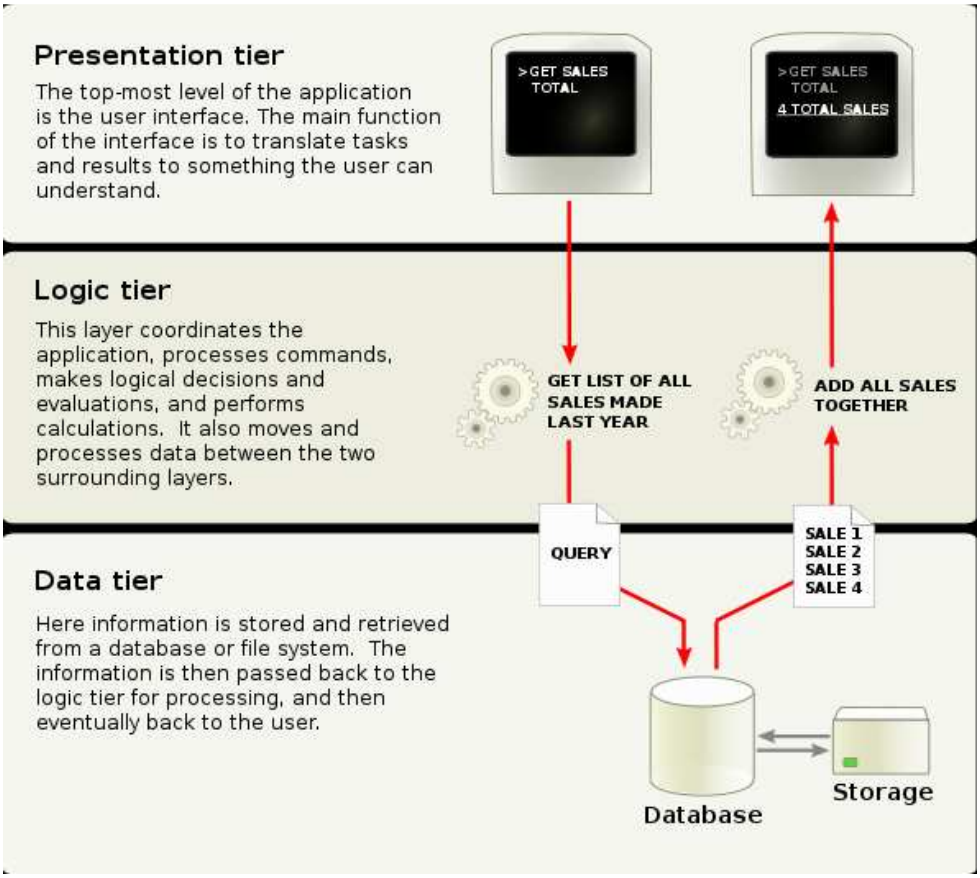
Trívrstvá architektura

Sklada se z prezentační vrstvy, aplikační vrstvy a databáze. Je to vlastne client-server architektura s přidanou vrstvou – **middle tier**, ta implementuje aplikační logiku nebo takové vlastnosti aplikace, které stojí na pomezí mezi vývojářem aplikace a síťařem.

Přínos architektury klient/server a třívrstvé architektury

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

Schema třívrstvé architektury



XML a znackovací jazyky

Značkovací jazyk (Markup Language) - je jazyk, jehož zdrojový text obsahuje současně jak vlastní text, tak instrukce pro jeho zpracování. Ty se zpravidla vyskytují v podobě příkazů (commands) či značek (tags). Zdrojovým textem bývá obyčejný ASCII soubor, což umožňuje jeho snadnou editaci i nejjednoduššími textovými editory. Typickým rysem značkovacích jazyků jsou **znaky se speciálním významem**. Ty slouží k vymezení řídicích konstrukcí - příkazů či značek.

Rozdělení značkovacích jazyků

Jazyky popisné (deskriptivní) - jejich konstrukce slouží k popisu, co jsou informace obsažené v dokumentu zač. Typickými představiteli jsou **XML** či **HTML** - jejich prostřednictvím lze například vymezit na stránce nadpis, odkud kam sahají jednotlivé odstavce či popsat odkaz na jinou stránku. Je ponecháno na zpracovávajícím programu, jak s těmito informacemi naloží a jak je promítne například do zobrazení dokumentu.

Jazyky výkonné (procedurální) - obsahují i výkonné instrukce na úrovni programovacího jazyka - typicky určitou formu pamětí či proměnných a nástroje pro přiřazování a využívání jejich hodnot. Výkonné jazyky zpravidla také umožňují velmi detailně popsat vizuální charakteristiky výstupu. Uživatel může tedy velmi přesně řídit vzhled výsledného dokumentu. Mezi procedurální jazyky patří **TeX** či **PostScript**. K demonstraci vyjadřovací síly prvního z nich v něm byl dokonce vytvořen interpret jazyka BASIC, přestože se jedná o typografický program.

XML

XML (e**X**tensible **M**arkup **L**anguage) - je formát pro přenos a výměnu obecných dokumentů. Je podmnožina (aplikace) **SGML (S**tandard **G**eneralized **M**arkup **L**anguage - ISO 8879) umožňující přenos dokumentu spolu s popisem jeho struktury (syntaxe). XML se nezabývá způsobem prezentace dokumentů – je to syntaktický nástroj.

XML procesor – je modul, který umí číst XML dokumenty a zpřístupňuje entity aplikacím. Musí umět kontrolovat, zda je XML dokument dobře vytvořený. Může být i **validující** - pak umí kontrolovat další podmínky kladené na XML dokument - tzv. **omezení správnosti** XML dokumentů, jinak kontroluje pouze omezení pro dobře vytvořený dokument.

Definice: XML text (dokument) je **dobře vytvořený** (well-formed), když je dobře uzávorkován.

Definice: XML text (dokument) je **správný** (platný, validní), když je dobře vytvořený a ještě vyhovuje nějakému schématu (např. DTD).

XML dokument

Zpráva jako XML dokument má definovánu syntaktickou strukturu, ale není definována její sémantika, ani sémantika jejích částí. Sémantiku je možno popsat pomocí **XSL (transformaci)** - jako sadu pravidel, kterými se XML dokument převede na dokument používající známý **sémantický slovník.**

Dobře vytvořený XML dokument **lze reprezentovat stromem**, jehož kořenový element je rodičem (predkem) všech elementů.

Každý správný XML dokument obsahuje **prolog** s deklarací verze XML, případně doplněné informací o kódování a samostatnosti dokumentu. Deklarace verze má tvar: **<?xml version="1.1" encoding=”iso-8859-2" standalone=”yes"?>**

Každý správný XML dokument obsahuje **deklaraci typu dokumentu (document type declaration)**, kterou je definováno omezení logické struktury dokumentu a povolená sada značek. Omezení logické struktury je stanoveno pomocí gramatiky dokumentu, která může být vyjádřena jako definice typu dokumentu (**DTD**) nebo jako **XML schéma**, např. v jazyce XML Schema.

Datové modely XML dokumentů

Potřebujeme-li s XML dokumentem pracovat, musíme jej nějak reprezentovat. Existují dva základní modely: **stromová reprezentace** (předpokládá vnitřní reprezentaci dokumentu ve formě stromu) a **událostmi řízený model** (převádí XML dokument na posloupnost událostí).

- **DOM** (**D**ocument **O**bject **M**odel) - analyzátor XML dokumentu čte (dobře vytvořený) XML dokument a vytváří v paměti (příp. virtuální) **stromovou reprezentaci dokumentu**. Strom má různé uzly, podle složky, kterou reprezentuje. Po načtení dokumentu do paměti jsou jeho složky přístupné pomocí traverzovacích funkcí (accessors). Výhodné, pokud zpracováváme dokument na přeskáčku a několikrát.
- **SAX** (**S**imple **A**PI for **X**ML) - analyzátor XML dokumentu čte (dobře vytvořený) XML dokument a generuje **posloupnost událostí reprezentující dokument**. Dokument nemusí vůbec být uložen v paměti, jeho složky jsou přístupné v okamžiku odpovídající události. Výhodné, pokud zpracováváme dokument sekvenčně, případně pouze jeho část a jednou.

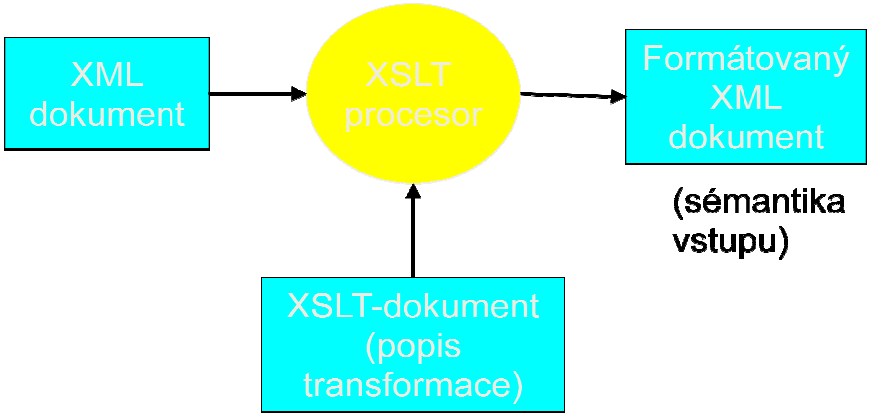
Dotazovací jazyky nad XML daty

- **XPath** - je syntaxe použitá pro výběr částí XML dokumentu. Způsob jakým XPath popisuje cesty k elementům je podobný způsobu, jakým OS popisuje cesty k souborům. Notace XPath není vyjádřena v XML.
  Priklad: **/kniha[@třída=“H.3.3” and autor=“Jiří Kosek”]**
- **XQuery** - větší vyjadřovací síla než XPath 1.0. Využiva XML Schéma (popis struktury, datove typy). Formální popis všech dat a operací nad nimi. Všechny (i částečné) výsledky těchto operací během dotazu.

XSLT

XSL (e**X**tensible **S**tylesheet **L**anguage) - je nástroj pro transformaci a formátování XML dokumentů. **XSL** = jazyk transformací (**XSLT**) + svět formátovaných objektů (**XSL:FO**).

XSLT dokument - je sada pravidel zapsaná v XML, kterou popisujeme transformaci vstupních XML dokumentů do výstupních dokumentů (formátovaných dokumentů). **Pravidlo** v XSLT (template, šablona) popisuje transformaci - výstup transformace je obvykle řízen strukturou vstupního XML dokumentu.



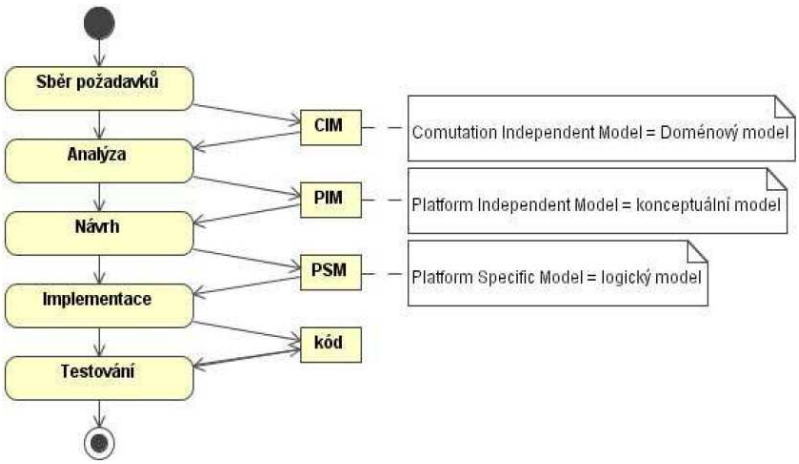
Objektová analýza a návrh (UML)

Metodika – je soubor prezentačních technik (výrazových prostředků) a metodických postupů, které podporují systematický vývoj a údržbu programového díla, včetně plánování a řízení prací na programovém projektu.

UML - je unifikovaný vyjadřovací prostředek pro dokumentaci obsahové části informatických projektů (dokumentace projektu obsahuje ještě projektovou dokumentaci, která popisuje projekt jako takový). UML umožňuje vyjádření dokumentace analýzy, návrhu i implementace.

Postup při tvorbě software - zdá se, že vývoj software přešel od kódem řízeného stylu ke specifikací řízenému stylu. Nejnovější trend konsorcia OMG (Object Management Group) je modelem řízený vývoj (MDD – Model Driven Development), nebo také modelem řízená architektura (MDA).

Modelem řízený vývoj v UML



UML (**Unified Modeling Language)** - je v softwarovém inženýrství **grafický jazyk** pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. UML nabízí **standardní způsob zápisu** jak návrhů systému včetně konceptuálních prvků jako jsou business procesy a systémové funkce, tak konkrétních prvků jako jsou příkazy programovacího jazyka, databázová schémata a znovupoužitelné programové komponenty.

UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu programových systémů. UML **neobsahuje způsob**, jak se má používat, ani neobsahuje metodiku (y), jak analyzovat, specifikovat či navrhovat programové systémy.

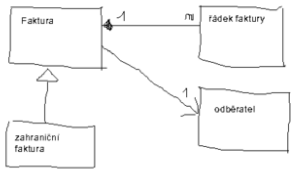
Používá se jako de facto standard. Byl přijat jako průmyslový standard ISO. Zdá se, že nastal čas, kdy lidé okolo tvorby programů postupně akceptovali existenci tohoto prostředku a začínají ho skutečně používat, nebo jsou k jeho používání nuceni. Pomocí UML komunikujeme so širokou škálou lidí, počínaje zákazníky, ale zejména se dohadujeme mezi sebou.

Tři úrovně využívání UML

- Jako **notaci** pro zachycení artefaktů, na kterých se je potřeba domluvit. Lze jej využít jak při přímo (nakreslíme obrázek, abychom si upřesnili imlementaci), tak zpětně (obrázek nám poslouží k pochopení existujícího kódu).
- Jako **dokumentační prostředek**, ve kterém dokumentujeme systém.
- Jako **programovací jazyk**, ze kterého se skutečná implementace (přesněji asi jen její kostra) generuje.

Způsoby použití UML : Kreslení konceptu

Při tomto použití je UML podpůrným nástrojem pro komunikaci mezi vývojáři a pro zaznamenání myšlenek a návrhů. Do diagramů se kreslí pouze věci podstatné pro grafické vyjádření návrhu, části návrhu před tím, než se začne programovat. Důležitá je srozumitelnost, rychlost nakreslení a snadnost změny či navržení alternativ řešení.



Způsoby použití UML : Kreslení detailních návrhů

Cílem je zaznamenat kompletní návrh či kompletní realizaci. Při kreslení návrhu by měl analytik obsáhnout všechny prvky tak, aby programátor byl schopen vytvořit program bez velkého přemýšlení nad věcnou oblastí (pro programátora by neměla vzniknout potřeba konzultace s uživatelem). Při kreslení detailních návrhů se obvykle používají specializované programy (**CASE**), které jsou schopny sdílet informace mezi jednotlivými modely a kontrolovat konzistenci návrhu. Při dokumentaci programu se často používání nástroje pro generování diagramů z vlastního kódu aplikace.

Způsoby použití UML : UML jako programovací jazyk

Při tomto použití vývojář nakreslí UML diagramy, ze kterých se **vygeneruje přímo spustitelný kód**. Toto vyžaduje specializované nástroje a velmi přesné vyjadřování v UML diagramech. V této souvislosti se velmi často používá pojem Model Driven Architecture (MDA), což je další standard skupiny OMG, který se snaží standardizovat použití UML jako programovacího jazyka.

Způsoby použití UML : Metamodel

Tento pohled používají autoři UML a autoři CASE nástrojů - nedívají se na UML jako na diagramy, pro ně je základem UML metamodel (diagramy jsou pouze grafickou reprezentací metamodelu). Při tomto přístupu se často používá pojem model místo pojmu diagram, např. místo diagramu tříd se používá pojem model tříd. Metamodel se popisuje pomocí Meta-Object-Facility (MOF) - abstraktního jazyka pro specifikaci, vytváření a správu metamodelů (další standard OMG). Pro výměnu metamodelů se používá XMI - na XML založený standard (součást standardu UML).

Specifikace UML obsahuje 4 základní části, které popisují:

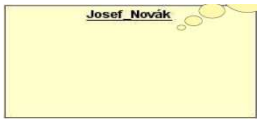
- **Infrastruktura** - metamodel stojící v pozadí za UML, specifikovaný pomocí Meta-Object Facility (MOF) (zhruba: **základní elementy a vztahy**).
- **Superstruktura** - popis UML z hlediska uživatele (analytik/programátor). Tato část popisuje jednotlivé diagramy. (zhruba: **diagramy a jejich význam**).
- **Výměnný formát** - opis XML struktur pro výměnu konkrétních modelů mezi jednotlivými modelovacími nástroji. (**jak zápis v UML exportovat a importovat**).
- **Definici jazyka OCL** - jazyk pro specifikaci vstupních a výstupních podmínek, invariantů v jednotlivých diagramech. (Object Constraint Language), ve kterém lze formálně přesně zapisovat **integritní omezení modelů**.

Lexikální elementy UML - čtyři základní druhy elementů:

- **Řetězec** - posloupnost znaků, znaková sada není omezena, v některých případech doporučeno ASCII.
- **Ikona** - grafický symbol zastupující element, neobsahující žádné složky.
- **2D-symbol -** grafický symbol zastupující element, který má obsah, případně rozdělený na oblasti.
- **Spojnice** - path - posloupnost úseček, které navazují, mohou mít zvýrazněny koncové body a mohou incidovat s hranicí 2D-symbolů - slouží k vyznačení propojení.

Příklad

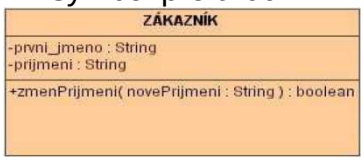
Ikona pro objekt



Ikona pro třídu



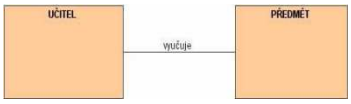
2D-symbol pro třídu



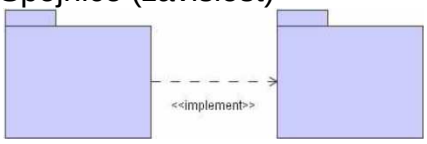
Ikona pro package



Spojnice (vztah)



Spojnice (závislost)



Výrazy v UML

Jsou řetězce sestavené podle jistých pravidel (syntaxe výrazů není striktní, je doporučeno používat OCL). Výraz představuje formuli, kterou musí být možno v daném jazyce vyhodnotit (má sémantiku). Definice UML zahrnuje definici OCL (Object Constraint Language), který je použit pro definici sémantiky UML:

objekt.selektor - označuje položku daného objektu (atribut, jméno role)
objekt.selektor [kvalifikátor] - označuje objekt vybraný podle kvalifikátoru
sada -> select (logická podmínka) - označuje objekty vybrané z dané sady podle logické podmínky.
Např: Firma.zamestnanci -> select(funkce="ucetni")

Rozšiřitelnost UML

Standardní sémantiku UML je možno doplnit standardními omezeními, která umožňují různou interpretaci elementů:

- **příznaky** (tags) – umožňují přidat k prvku diagramu další informace.
  { název atributu = hodnota }
- **stereotypy** (stereotypes) – umožňují klasifikovat elementy.
  <<include>>, <<extend>>
- **omezení** (constraints) – umožňují specifikovat omezení pro elementy.
  { počet_zaznamu < 10000 }

Diagramy

Diagramy sú najznámejšie a najpoužívanejšie časti štandardu. Nasleduje prehľad diagramov v UML 2.0 vrátane ich rozčlenenia do skupín:

Struktúrne diagramy

- diagram tried (class diagram)
- diagram komponentov (component diagram)
- diagram zloženej štruktúry (composite structure diagram)
- diagram nasadenia (deployment diagram)
- diagram balíčkov (package diagram)
- diagram objektov (object diagram), nazýva sa aj diagram inštancií

Diagramy chovania

- diagram aktivít (activity diagram)
- diagram prípadov použitia (use case diagram)
- stavový diagram (state machine diagram)
- **Diagramy interakcie**
  - sekvenčný diagram (sequence diagram)
  - diagram komunikácie (communication diagram) - predtým diagram spolupráce (collaboration diagram)
  - diagram prehladu interakcií (interaction overview diagram)
  - diagram časovania (timing diagram)

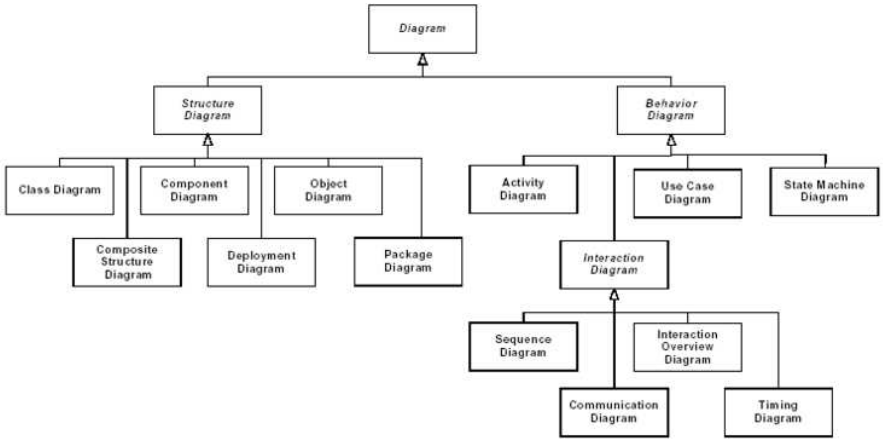
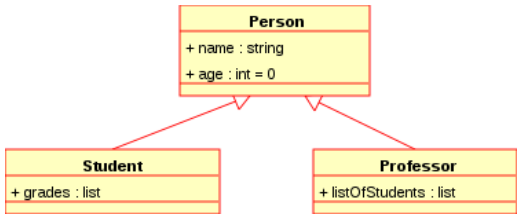


Diagram tried (class diagram)

Každá trieda, zobrazená ako obdĺžnik so svojim názvom uvedeným vnútri, reprezentuje časti systému alebo softvéru. Trieda môže zobrazovať svoje vlastnosti, nazývané v UML atribúty pod svojim názvom. Každý atribút je zobrazený minimálne svojim menom, môžu byť pri ňom uvedené aj jeho parametre a typ návratovej hodnoty.



Jazyk UML definuje nasledovné vzťahy v diagramoch tried:

Generalizácia - je vzťah medzi nadradenou a podradenou triedou. Grafickou reprezentáciou dedičnosti je trojuholník na konci nadradenej (rodičovskej) triedy.

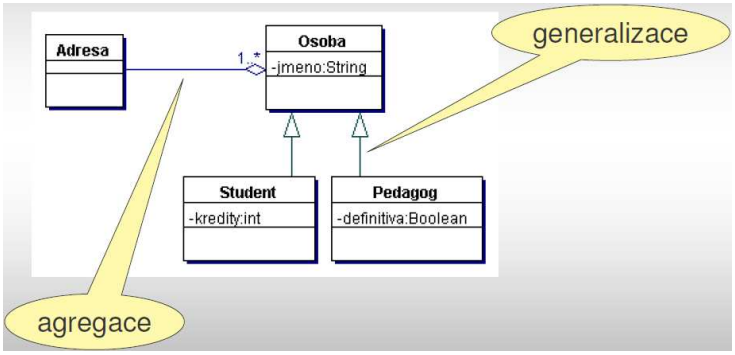
Asociácia - zakresľujú sa medzi triedami, ale reprezentujú vzťahy medzi objektami. Úloha objektu vo vzťahu asociácie môže byť špecifikovaná na konci asociácie, ako aj na multiciplite (počet objektov, ktoré sa zúčastňujú asociácie).



Agregácia - nastáva, keď je trieda vytvorená ako kolekcia alebo kontajner inej triedy bez silnej závislosti životného cyklu jej inštancií. Jej grafická reprezentácia je prázdny kosoštvorec na konci triedy.

Kompozícia - je silne závislá na životnom cykle medzi objektami daných tried. Ak je zničený kontajner, je zničený aj objekt v tejto triede. Grafickou reprezentáciou kompozície je čierny kosoštvorec na konci podriadenej triedy.

Asociačná trieda - je konštrukcia, ktorá umožňuje asociačnej väzbe mať operácie a atribúty. Grafickou reprezentáciou je trieda pripojená k asociácii prerušovanou čiarou.



Informacní bezpečnost

Cil informacni bezpecnosti

- **Utajení** - objekty systému jsou přístupné pouze autorizovaným subjektům.
- **Integrita** - objekty mohou být modifikovány pouze oprávněnými subjekty.
- **Dosažitelnost** - objekty jsou dostupné pro autorizované subjekty.

Zdroje ohrožení

- Vyšší moc (požár, povodeň, zemětřesení, blesk, …)
- Závady technického zařízení
- Neúmyslné lidské chyby
- Záměrné útoky

Možné hrozby pro cil informacni bezpecnosti

- **Přerušení** - některá část systému je ztracena nebo nedosažitelná.
- **Zachycení** - neautorizovaný subjekt získá přístup k nějakému objektu systému.
- **Modifikace** - neautorizovaný subjekt získá možnost pozměňovat některé části systému.
- **Fabrikace** - neautorizované vytvoření nového objektu.

Objekty ochrany – aktiva

Technická zařízení – prinaseji nejrůznější problémy počínaje ukradením daného zařízení, konče klávesnicí prolitou džusem či kabely překousanými od myší. Záměrné poškození komponent sytému, přírodní katastrofy.

Programy - **Dostupnost** - úmyslná/náhodná ztráta softwaru se stane zřejmou až při pokusu o spuštění. Pokud systém poskytuje rozumnou ***správu konfigurací***, lze snížit riziko náhodné ztráty. **Modifikace** - nemusejí být poznatelné, určení jejich rozsahu a následků je obtížné. **Krádež** - programy bývají cennou součástí systému, krádeže jsou těžko detekovatelné. Právní problémy.

Data - hodnota dat je těžko vyčíslitelná. Povětšinou však značná hodnota. Schopnost udržet data v tajnosti je často životně důležité, hodnota může být v čase značně proměnná.

Záznamová media je nutný vhodný plán tvorby a správy záložních kopií.

Sítě - přinášejí specifický problém pro ochranu komunikace jednotlivých uzlů, vzájemné identifikace a ochrany přenosových medií.

Způsoby ochrany

Cílem ochrany je zajistit utajení, integritu a dostupnost všech objektů. Některé metody zajišťují prevenci před útoky, jiné pouze detekují napadení.

Šifrování - Zakódování dat tak, aby nebyla běžnými prostředky čitelná. I v případě odcizení takto zakódovaných dat nedojde k úniku informací. Dále lze použít k

zajištění integrity dat, nebo k vytvoření speciálních protokolů pro výměnu informací. Elektronické podpisy (nejen) elektronických dokumentů.

Softwarové kontroly - Programy musí být bezpečné a spolehlivé. Protože SW kontroly přímo ovlivňují přístup uživatele k systému, musí být navrženy tak, aby neznemožňovaly práci. Patria sem:

- **Kontrola vývoje** - soubor standardů, jak vytvářet, testovat a udržovat programy pro správu chráněných dat.
- **Kontroly operačního systému** - různá omezení vynucená operačním systémem, jejichž cílem je chránit jednotlivé uživatele navzájem.
- **Interní programové kontroly** - omezení vynucená samotným programem, např. omezený přístup do jednotlivých částí databáze.

Kontroly hardwaru - technická zařízení používaná ke zpracování utajovaných dat musí vyhovovah specifickým normám. Dále je nutne jejich vhodné zajištění proti útokům a celkové zabezpečení přístupu k nim.

Bezpečnostní politika - velmi účinná bývají nejjednodušší bezpečnostní opatření. Vysvětlit sekretářce, že si nemá svůj hardwarový klíč nechávat v horní zásuvce psacího stolu, přimět uživatele k občasným změnám hesel. Je vhodné uživatelům vštěpovat **postupy vedoucí k zajištění bezpečnosti** a základy etiky ve vztahu k počítačům. Toto působení na uživatele by mělo být ucelené co do formy a obsahu.

Fyzické kontroly - obyčejné zámky, stráže, záložní kopie dat a programů, náhradní technické vybavení, ale i plánování umístění jednotlivých komponent systému. Fyzické kontroly bývají velmi účinné, bohužel jsou však často přehlížené.

Bezpecnostni politika - je dokument, ve kterem se naplanuje, jakym zpusobem se budou resit vsechny oblasti bezpecnosti, kdo je za co zodpovedny a jak to bude implementovano a provozovano. Bezpecnostni politika rika, jak zvladnout problem zajisteni IS proti incidentum. Stejne jako kazda cinnost, i provozovani systemu pro spravu informaci je spojeno s jistym rizikem (chyba zarizeni, obsluhy, programu, vandalismus, kradez). Provedeni kvalifikovaneho odhadu rizik prinasi vyhody.

Odhad rizik

Provozovani systemu pro spravu informaci je vzdy spojeno s urcitym rizikem (pr. chyba zarizeni, obsluhy, programu, vandalismus, kradez). Prinos odhadu rizik:

- **Zlepseni obecneho povedomi** - Pracovnici si problem uvedomi a maji sanci jej pochopit.
- **Identifikace hodnot**, slabin a moznych kontrol celeho systemu - Ne vzdy je jasne, které casti systemu maji nejvetsi hodnotu a odkud prameni nejvetsi nebezpeci.
- **Zlepseni vychodiska pro strategicka rozhodnuti** - nektere ochranne a kontrolni mechanismy velmi snizuji produktivitu systemu, pricemz jejich prinos neni zrejmy, ruzne druhy nebezpeci jsou ruzne realne a predstavuji mnohdy daleko vetsi hrozbu, nez by se dalo ocekavat.
- **Lepsi rozlozeni vydaju na bezpecnost** - nektere velmi drahe ochranne mechanismy poskytuji pouze male zvyseni bezpecnosti a popripade i naopak.

Provedeni odhadu rizik lze rozdelit do nekolika kroku:

- Identifikace hodnot
- Urceni slabin
- Odhad pravdepodobnosti zneuziti
- Vypocet ocekavanych rocnich ztrat
- Prehled pouzitelnych ochrannych mechanismu
- Nastin rocnich uspor ze zavedenim ochrannych mechanismu

Druhy útoků

- **Vnitřní útoky**
  - **Trojský kůň** - zdánlivě neškodný program obsahuje „zlý“ kód.
  - **Login spoofing** - falešná „logovací“ obrazovka.
  - **Logická bomba** - zaměstnanec vpraví kus kódu do systému, který musí být pravidelně informován o tom, že zaměstnanec je stále zaměstnancem.
  - **Zadní dvířka (trap door, back door)** - kód při nějaké podmínce přeskočí normální kontroly.
  - **Přetečení vyrovnávací paměti (buffer overflow)** - ve velkém množství kódu nejsou dělány kontroly na přetečení polí pevné velikosti. Při přetečení se typicky přepíše část zásobníku a lze tam umístit adresu kódu i samotný kód, který se vykoná při návratu z funkce.
- **Vnější útoky**
  - **Virus** - vytvoří se nakažený „žádaný“ soubor. Po spuštění infikuje další soubory. Přenos přenosnými médii, sítí. Možné řešení jsou antiviry.
  - **Internetový červ (worm)** - samoreplikující se program (červ) využívá nějaké chyby systému.
  - **Mobilní kód** - applety, agenti, PostScript.

Další pojmy a postupy v informační bezpečnosti

ACL (Access Control List) -ke každému objektu seznam práv pro uživatele/skupiny.

C-list (Capability List) - ke každému uživateli/skupině seznam práv pro objekty.

Autentikace - Identifikace něčím, co uživatel ví, má nebo je. Může být zajištěna:

- **Heslo** – hrozí slovníkový útok (hrubá síla), protože 80-90% hesel je jednoduchých. Řeší se vynucováním délky, složitosti a doby trvání hesla. Lze použít hesla na jedno použití (pomocí knihy hesel, nebo speciální kalkulačky).
- **Model otázka/odpověď**
- **Fyzický objekt** - smart cards, USB klíče.
- **Biometrika** - rozeznávání papilárních čar, rohovek, hlasu.

Kryptografie

Cíle kryptografie jsou důvěrnost, celistvost, autentizace (vím, od koho to skutečné pochází) a nemožnost popření (když jednou potvrdím nějakou akci, už to nemohu popřít).

Symetrický klíč

Symetricky klíčované šifry se dělí na **blokové šifry** (např. DES, IDEA) a proudové šifry (SEAL). Výhody šifrování symetrickým klíčem spočívají ve vysoké datové propustnosti, v nízké pamětové náročnosti pro uložení klíče (krátké klíče), a fakt, že slučování více šifer vytváří silnější šifru. Nevýhody spočívají v tom, že klíče na obou koncích musí zůstat utajeny, je třeba často měnit klíče a potřeba ověřené TTP (Trusted Third-Party).

Asymetrický klíč

Jsou šifry s veřejným klíčem (např. RSA). Výhody spočívají v tom, že pouze privátní klíč je (musí být) tajný, klíče je nutné měnit méně často a není zapotřebí úplně ověřená TTP. Nevýhody spočívají v tom, že šifrování/dešifrování je mnohem pomalejší než symetrické šifry, klíče jsou mnohem delší a o žádném schématu veřejného klíče nebylo dokázáno, že je bezpečné.

