

Okruhy ke státní závěrečné zkoušce pro bakalářský studijní program Informatika

A. Předmět Informatika

Programování (FPR, UPR, OOP, SJ, C#/JAVA I a II, UTI, ZDS)

<https://mrlvsb.github.io/upr-skripta/>

- Paradigmata programování (deklarativní a imperativní, funkcionální, strukturované a objektově orientované paradigma, specifické vlastnosti a principy jednotlivých paradigm).

Deklarativní

- Deklarativní programování je založeno na myšlence programování aplikací pomocí definic co se má udělat, a ne jak se to má udělat.
- algoritmizace (způsob vykonání) je ponechána na programu (interpretu) daného jazyka
- SQL, Prolog, Scheme, Haskell
- deklarativní jsou i funkcionální a logické programovací jazyky
- uživatel definuje co se má provést, ale neřeší jak to bude provedeno

Imperativní (procedurální)

- Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit.
- Program je sadou proměnných, jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav.
- Imperativní přístup je blízký člověku - návody, postupy, kuchyňské recepty
- velké množství programovacích jazyků jsou imperativní (tzn. i strukturované nebo OOP) - Fortran, C, C++, Java, C#, Python

Funkcionální

- Funkcionální programování je deklarativní programovací paradigma, které chápe výpočet jako vyhodnocení matematických funkcí
- Vychází z výpočetního modelu lambda calcul
- Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce. -> program se skládá z funkcí
- neobsahuje cykly, využívá se rekurse

Strukturované

- je speciálním případem imperativního
- cílem dosáhnout lepší srozumitelnosti, vyšší kvality a kratší doby vytváření programů z

řídících struktur s jedním vstupním a jedním výstupním bodem místo neomezeného užívání skoků

- definuje programovací struktury které vedou ke zlepšení přehlednosti kódu
- místo skoků definuje rozhodování (větvení) a cykly
- příkazy jsou vykonávány postupně
- příklad: jazyk C

Objektově orientované (<http://lucie.zolta.cz/index.php/statnice-vsbs?layout=edit&id=141>)

- Výkonný kód je v objektovém programování přidružen k datům (metody jsou zapouzdřeny v objektech), což umožňuje snadnější přenos kódu mezi různými projekty
- Popisujeme objekty reálného světa jeho data a funkce
- příklad objektově orientovaného jazyka: C++, C#

-
- Při řešení úlohy vytváříme model popisované reality - popisujeme entity a interakci mezi entitami
 - Abstrahujeme od nepodstatných detailů - při popisu/modelování entity vynescházáme nepodstatné vlastnosti entit
 - Postup řešení je v řadě případů efektivnější než při procedurálním přístupu (ne vždy), kdy se úlohy řeší jako posloupnost příkazů

Cíle OOP

- Je vedeno snahou o znovupoužitelnost komponent.
- Rozkládá složitou úlohu na dílčí součásti, které jdou pokud možno řešit nezávisle.
- Přiblížení struktury řešení v počítači reálnému světu (komunikující objekty).
- Skrytí detailů implementace řešení před uživatelem.

Třída (Class)

Třída slouží jako šablona pro vytváření instancí tříd - objektů. Seskupuje objekty stejného typu a podchycuje jejich podstatu na obecné úrovni. (Samotná třída tedy nepředstavuje vlastní informace, jedná se pouze o předlohu; data obsahují až objekty.) Třída definuje data a metody objektů.

Popisuje jaká data bude instance obsahovat a jaké bude mít metody.

Objekt

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

Rysy OOP

- **Abstrakce** – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže

provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

- **Zapouzdření** (Encapsulation) – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.
- **Skládání** – Objekt může obsahovat jiné objekty.
- **Delegování** – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.
- **Dědičnost** (Inheritance)
 - Každá třída může dědit z jiné třídy. Tím přebere všechny její vlastnosti a může ji rozšířit o další proměnné a metody.
 - To umožní znovuvyužitelnost kódu. Nedefinuje se společné chování do 1 jedné třídy a další třídy z ní dědí a rozšiřují ji
 - (objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).)
- **Polymorfismus** – odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle implementace. U polymorfismu podmíněněho dědičnosti to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy. U polymorfismu nepodmíněného dědičnosti je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní. Polymorfismus bývá často vysvětlován na obrázku se zvířaty, která mají všechna v rozhraní metodu Speak(), ale každé si ji vykonává po svém.
 - výhoda dědičnosti

Více tříd dědí z jedné třídy, která určuje co mají společné, případně co musí mít na implementováno. Tyto třídy lze uchovávat v proměnných třídy, ze které se dědí. Lze vyvolat společné metody. U metod, které musí každý potomek implementovat sám se rozhoduje o jakou třídu se jedná a tedy jaká implementace se použije.

- **Genericitá** - je možnost programovacího jazyka definovat místo typů jen „vzory typů“, konkrétní datový typ se následně specifikuje až při vytváření instance.

[https://cs.wikipedia.org/wiki/Imperativn%C3%AD_programov%C3%A1_n%C3%dí](https://cs.wikipedia.org/wiki/Imperativn%C3%AD_programov%C3%A1_n%C3%AD)

[https://cs.wikipedia.org/wiki/Funkcion%C3%A1ln%C3%AD_programov%C3%A1_n%C3%dí](https://cs.wikipedia.org/wiki/Funkcion%C3%A1ln%C3%AD_programov%C3%A1_n%C3%AD)

[https://cs.wikipedia.org/wiki/Strukturovan%C3%A9_programov%C3%A1_n%C3%dí](https://cs.wikipedia.org/wiki/Strukturovan%C3%A9_programov%C3%A1_n%C3%AD)

[https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1_n%C3%dí](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1_n%C3%AD)

- **Datové typy (statická a dynamická alokace, dynamické typování, životní cyklus paměti, jednoduché, strukturované, abstraktní a generické typy, kolekce, soubory, streamy).**

Datový typ určuje rozsah a druh hodnot, kterých může proměnná nabývat. Dále definuje povolené operace prováděné s proměnnou.

statická a dynamická alokace

statické proměnné

https://docs.google.com/document/d/1zbOMJI26-NNKcVrBupMUjY_pVW-ieEUayeNmjlYIRY8/edit#

- Lokální, globální proměnné
- Vytváří se při překladu
- při překladu je známa jejich velikost a adresa
- Alokují se v zásobníku.

dynamické proměnné

- Nemají předem stanovenou velikost, alokace se provádí speciálním příkazem za chodu programu. Nemají identifikátor. Přistupuje se k nim pomocí pointerů (ukazatel na pozici v haldě). Alokují se v haldě.
- Při překladu nemusí být známa jejich velikost
- Za jejich správu je odpovědný programátor tj. za jejich uvolnění, v novějších programovacích jazyčích provádí uvolnění paměti Garbage Collector

dynamické typování

- Dynamická typová kontrola probíhá za běhu programu.
- Jazyky používající dynamické typování nepožadují specifikaci datového typu u proměnných a ty mohou tudíž odkazovat na hodnotu jakéhokoli typu.
- např. Python

životní cyklus paměti

- paměť alokovaná staticky je uvolněna po přesně stanovené době (při zániku platnosti proměnné - konec bloku kódu), dynamicky alokovanou paměť musí uvolnit sám programátor nebo příslušný garbage collector.

datové typy

-jednoduché

- zabudováný přímo do programovacího jazyka
 - celé číslo (integer)
 - reálné číslo, číslo s plovoucí řádovou čárkou (float,real)
 - znak (char)
 - logická hodnota (bit, boolean)
- lze z nich vytvářet složitější datové typy

-strukturované

- programátorem definované datové typy (mohou být heterogenní) skládající se z více proměnných jednoduchých datových typů
- struktury - heterogenní, pole - homogenní

-abstraktní a generické

- nezávislé na vlastní implementaci - nemají určen datový typ, datový typ se určí až při vytváření instance
- umožňuje vytvářet i složitější datové typy
 - zásobník, fronta,
- jsou realizovány pomocí jednoduchých, obecných operací - přiřazení, sčítání, ... ?
- Krátká definice: Abstraktní datový typ je implementačně nezávislá specifikace struktury dat s operacemi povolenými na této struktuře.
- Šablony v C++ (template <typename T> T max(T a, T b){...})
- parametrizované datové typy v C# (public class Dictionary<K, V>{...})

kolekce

- abstraktní datový typ obsahující sadu hodnot jednoho nebo různých typů a umožňující přistupování k těmto hodnotám
- umožňuje do sebe zapisovat hodnoty a získávat je
- Cíl kolekce je sloužit jako úložiště objektů a zajišťovat k nim přístup
- Pro přistupování ke konkrétnímu elementu kolekce mohou používat různé metody v závislosti na její logické organizaci (programově vytváří různé datové struktury)
 - pole, zásobník, fronta, strom, lineární seznam

soubory

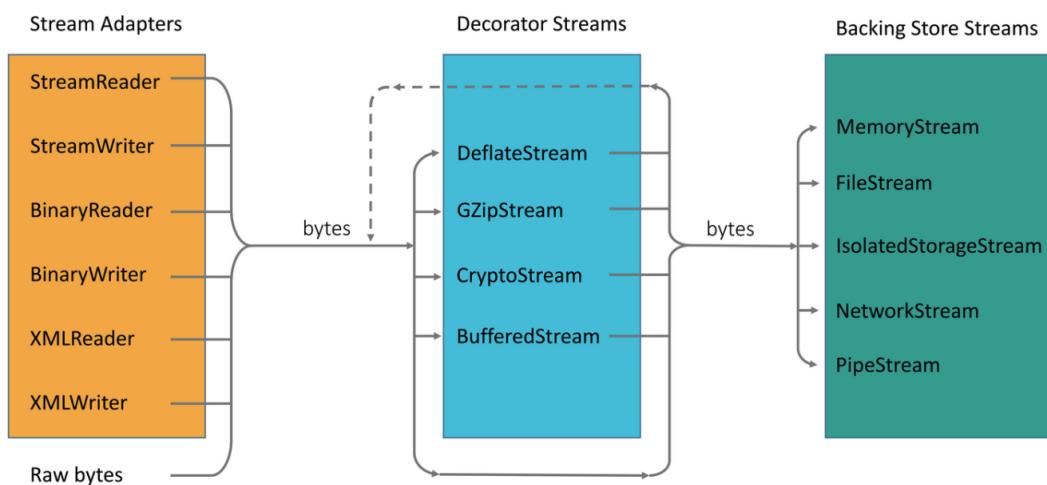
- sada dat uložena na datovém médiu
- nutné využít pro uchovávání dat mezi spuštěními programu
- přístup k nim spravuje OS a poskytuje systémová volání
- v moderních programovacích jazycích k nim lze přistupovat pomocí streamů

streamy (c# I prezentace 11)

- "Datovody" - data neukládají, pouze je zpracovávají, transformují, přijímají a předávají
- Datové streamy
- - pevně připojené k určitému typu datového úložiště, např. FileStream, NetworkStream.

- - pracují výhradně s bajty
- Dekorátory
- - nějakým způsobem transformují data, např. DeflateStream nebo CryptoStream
- Adaptéry
- - překlenou mezeru tím, že zabalí stream do třídy se specializovanými metodami typizovanými na určitý formát
- - např. StreamReader obsahuje binární stream a metody pro převod na text, xmlReader má stream a metody pro rozbalení xml dat?

Stream Architecture



https://cs.wikipedia.org/wiki/Dynamick%C3%A1_alkace_pam%C4%9Bti

https://cs.wikipedia.org/wiki/Datov%C3%BD_typ

https://cs.wikipedia.org/wiki/Abstraktn%C3%AD_datov%C3%BD_typ

[https://cs.wikipedia.org/wiki/Kolekce_\(abstraktn%C3%AD_datov%C3%BD_typ\)](https://cs.wikipedia.org/wiki/Kolekce_(abstraktn%C3%AD_datov%C3%BD_typ))

<https://cs.wikipedia.org/wiki/Soubor>

<https://cs.wikipedia.org/wiki/Datovod>

- Programovací techniky (**řídící struktury, funkce, jejich parametry a návratové hodnoty, rekurze, polymorfismus, kolekce, výjimky a jejich strukturovaná obsluha, paralelismus a vlákna**).

Řídící struktury

- konstrukce pro zápis počítačového programu - větvení a cykly,

funkce

- každá funkce má identifikátor(název) kterým je volána
- funkce musí mít unikátní název, pokud mají dvě funkce stejný název, musí se lišit počet jejich parametrů, jejich datové typy nebo návratová hodnota - přetěžování funkcí
- void nebo 1 návratová hodnota
- libovolný počet parametrů,
 - v c# speciální parametr params - pole parametrů které jsou při volání zadávány oddělené čárkou
 - parametry můžou mít nastaveny implicitní hodnoty - při volání nemusí být zadány a použijí se hodnoty implicitní

rekurze

- funkce volá sebe samu
- náhrada za cyklus
- vhodné pro některé matematické operace např. faktorial nebo průchody dat např stromové struktury
- musí mít "zarážku", která zajistí ukončení rekurze (nebude volat sebe samu ale vrátí hodnotu nebo se ukončí)

polymorfismus viz OOP

kolekce viz datové typy

výjimky

- když v programu ve funkce nastane chyba (výjimečná situace za běhu programu) bylo nutné aby funkce vracela chybovou hodnotu a ta se musela testovat - komplikované na použití a nepřehledné
- v moderních programovacích jazycích tento problém řeší výjimky = zobecnění vnitřního přerušení programu vyvolané chybou
- obvykle je výjimka spojena s objektem nesoucím informace o chybě
- blok try a catch - pokud v bloku try dojde k výjimce vstoupí se do bloku catch, kde může být výjimka zpracována a reagováno na ni
- při vyvolání výjimky se prochází call stack dokud se nenajde try-catch blok (probublávání výjimek)
- výjimku lze vyvolat příkazem throw

strukturovaná obsluha výjimek

- lze vyvolávat/mohou být vyvolány různé typy výjimek
- Na každou z nich lze reagovat jinak a na jiné úrovni programu (v nadřazeném try catch atd.)
- V první skupině jsou výjimky, jejichž výskyt představuje vážný problém v činnosti aplikace a které by programátor neměl zachycovat. Druhou skupinu tvoří výjimky, jejichž ošetření má

smysl - jde například o výjimky jako pokus o otevření neexistujícího souboru, chybný formát čísla apod. Do této skupiny by měly patřit také veškeré výjimky definované uživatelem. Zvláštní podtřídu třídy Exception tvoří třída RuntimeException, která zahrnuje veškeré ošetřitelné výjimky vyvolané samotným systémem.

<http://www.cs.vsb.cz/benes/vyuka/upr/texty/java/ch01s07s01.html>

parallelismus a vlákna C# II 2

- souběžné vykonávání různých částí programu - Vlákno je podmnožina procesu. Jeden proces může mít více vláken a je mezi ně rozdělován procesorový čas procesu. (Proces je vykonávání programu (instrukcí) spolu s daty, instrukčním čítačem atd. OS přepíná procesy)
- parallelismus může zvýšit rychlosť programu (na více procesorovém hw) - souběžné vykonávání programu ve více vlánech, reálně je více vláken než jader procesoru - dochází k přepínání vláken (běží za sebou)
- vytváření vláken je vhodné pro vykonávání časově náročných operací, aby nedošlo k zablokování hlavního vlákna např. v grafických aplikacích - práce se soubory, se sítí, náročné výpočty
- vlákna sdílejí adresový prostor mohou přistupovat ke stejným proměnným
- čistý parallelismus - velké množství procesorů pracující na stejně úloze - grafické karty

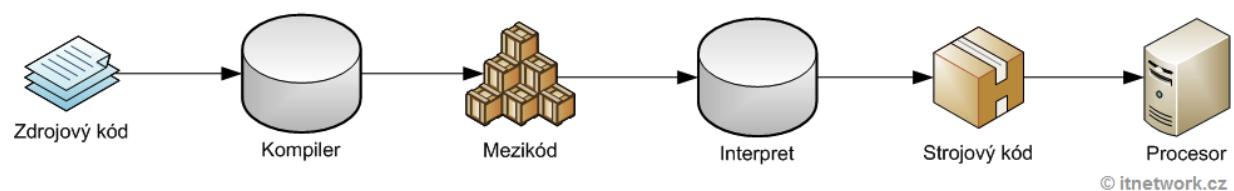
<https://editit.wordpress.com/2016/05/05/4-programovani-ridici-struktury/>

• **Vývoj aplikací (C#/Java, typový systém, virtuální stroj a komplikace, tvorba uživatelského rozhraní, delegáty a události, přístup k databázím, práce s textovými daty, asynchronní programování, serializace, síťová komunikace).**

- C# je staticky typovaný jazyk, všechny proměnné musíme nejprve deklarovat s jejich datovým typem

Virtuální stroj a komplikace

- Zdrojový kód je nejprve přeložen do tzv. mezikódu CIL
- Jedná se v podstatě o strojový (binární) kód, který má ale o poznání jednodušší instrukční sadu a přímo podporuje objektové programování
- Je nezávislý na platformě a hw
- Tento mezikód je potom díky jednoduchosti relativně rychle interpretovatelný tzv. virtuálním strojem (tedy interpretem, v případě .NET je to tzv. CLR - Common Language Runtime).
- Výsledkem je strojový kód pro konkrétní procesor a OS.



Tvorba uživatelského rozhraní C# II 3,4

- C# obsahuje řešení pro vytváření GUI aplikací, které definují standardní grafické komponenty
- poskytují možnost snadného napojení grafických komponent na programový kód
- řešení Windows Forms, WPF, MAUI

Delegáty C# I 14

- typ, který představuje odkazy na metody s konkrétním seznamem parametrů a návratovým typem
- do instance delegátu lze uložit libovolnou metodu odpovídající delegátu návratovým datovým typem a parametry

Události

- náhrada za návrhový vzor observer
- Událost je sada delegátů
- Lze do ní přiřadit více metod a následně je hromadně vyvolat - lze hromadně notifikovat přihlášené sledující

Přístup k DB C# II 5

- Databázové servery, lokální databázové soubory, XML, JSON,
- Přístup k SQL databází:
- Objektově relační mapování - Data z datové vrstvy jsou rovnou přístupné v podobě objektů, jsou definovány CRUD operace
- V případě implementace vlastního ORM je potřeba řešit připojení a dotazování se na DB. Pro přístup k DB potřebuji znát *connectionString* k dané databázi. Ten potom předám třídě *SqlConnection*. Jednotlivé příkazy se realizují pomocí třídy *SqlCommand*.
- Nepoužívá se přímé dotazování (SQL), ale SQL se automatizovaně generuje (např. pomocí reflexe - zjišťování názvů a datových typů tříd za běhu programu)
- Existuje několik ORM frameworků - Entity Framework, NHibernate, LINQ to SQL

práce s textovými daty C# II 6

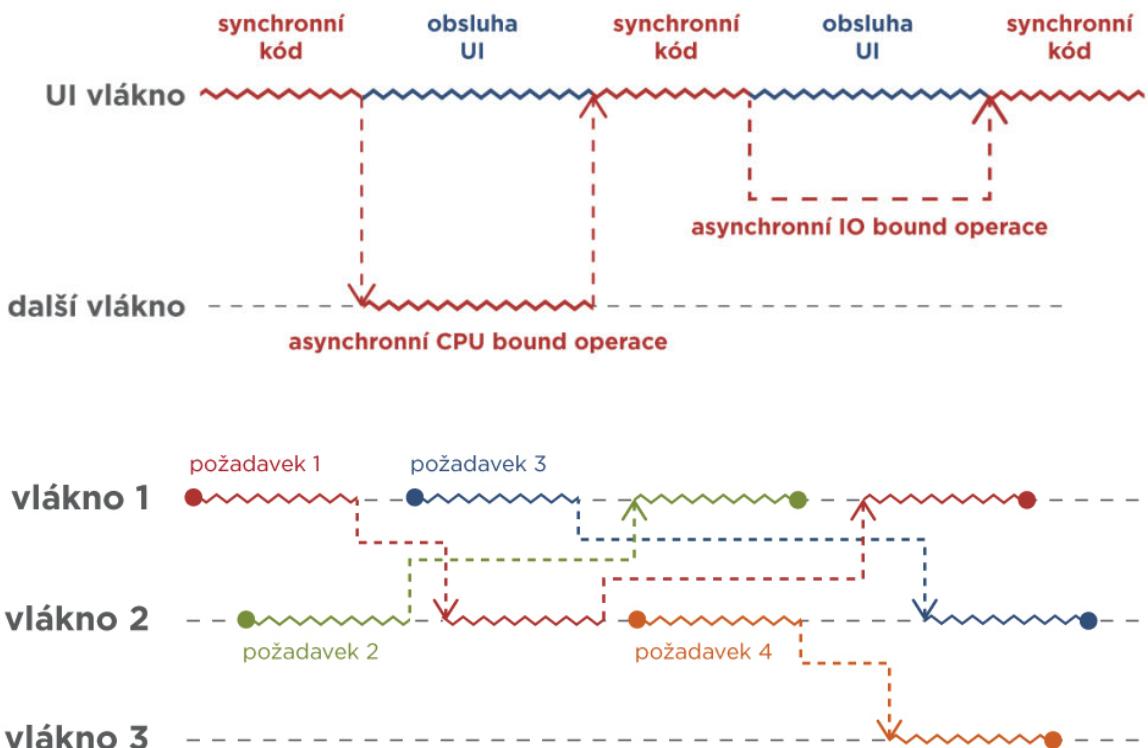
- regulární výrazy - třída *Regex*, vyhledávání v řetězci na základě zadанého vzoru (pravidla)
- JSON - serializace a deserializace dat do textové podoby v objektovém zápisu, vhodné pro přenos nebo ukládání dat
- XML - serializace a deserializace dat strukturovaný zápis, jazyk pro popis jazyků např html
- Xpath - Dotazovací jazyk pro XML dokumenty

asynchronní programování C# II 2

- Asynchronní programování se snaží implementovat déle trvající části kódu přímo asynchronně – vyšší efektivita, jednoduchost. Kdežto u synchronního programování se implementuje standardně a následně se využívají asynchronní principy běhu (Thread, Task).
- abstrakce běhu části kódu založená na konceptu „příslibu výsledku v budoucnu“
- Nemusí reálně dojít k vytvoření nového vlákna, jedná se o uvolnění vlákna pro provedení nějaké operace (úkony se prolínají mezi vlákny aby se vykonávaly souběžně)
- realizace pomocí tříd Thread a Task nebo async metody a čekání na návratovou

hodnotu metody příkazem await

- Příklad použití - spustíme asynchronní metodu vracející task s hodnotou a task si uložíme do proměnné, v hlavním vlákně provádíme operace, await proměnná ts taskem a čekáme dokud se nedokončí a nezískáme hodnotu
<https://youtu.be/X9N5r6kMOxw>



<https://profinit.eu/blog/ponorme-se-do-async-await-v-jazyce-c-1-cast/>

serializace

- Serializace je uchování stavu objektu
- konvertování objektu na proud bytů a poté uložení někde do paměti, databáze nebo souboru
- Deserializace je opak serializace. Dalo by se říci, že tedy převedete zpátky proud bytů na kopii objektu.
- využívá se pro uložení aktuálního stavu aplikace při ukončení. Při spuštění je možno deserializovat a načíst tedy objekty aplikace s konkrétními daty a stavem

síťová komunikace C# II 7

- mnoho implementací síťové komunikace
- WebClient - Základní třída (fasáda) ulehčující tvorbu kódů, např. možnost přímého přístupu k jiným typům dat, než v podobě streamů
- HttpClient - Specializuje se na implementaci využívající HTTP protokol s ohledem na přístup k API, příjem Http dat na klienta
- HttpListener - Třída umožňující realizaci server-side funkcionality, tj. implementaci HTTP serveru
- protokoly TCP a UDP - protokoly na transportní síťové vrstvě, tyto třídy využívají i výše

zmíněné pro komunikaci

https://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-promenne-typovy-system-a-parsova_ni

<https://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework>

<https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/delegates/>

<https://www.itnetwork.cz/csharp/soubory-a-sit/tutorial-csharp-serializace-a-deserializace/>

- Reprezentace čísel v počítači (číselné soustavy a převody, celá čísla, čísla s pevnou řádovou čárkou, čísla s pohyblivou řádovou čárkou vyjádřená v binárním, decimálním a hexadecimálním základu, aritmetika s čísly v různých reprezentacích, kódování znaků).

číselné soustavy a převody ZDS 4

- v počítačích je využívaná binární soustava
- číslo v libovolné soustavě lze převést do desítkové polynomem -

$$N_B = a_{n-1} \cdot B^{n-1} + \dots + a_i \cdot B^i + \dots + a_1 \cdot B^1 + a_0 \cdot B^0$$

- a - číslice, B základ číselné soustavy na váhu tj. pozici číslice
- převod kladných čísel z desítkové do libovolné soustavy
- (přímý převod - jednička na nejvyšší bity a jejich odčítání od čísla)m,
- postupné odečítání vah - celé i desetinné kladné čísla

Váha	Rozdíl	Koeficient
$8^2 = 64$	$187 - 64 = 123$	$a_2 = 1$
	$123 - 64 = 59$	$a_2 = 2$
$8^1 = 8$	$59 - 8 = 51$	$a_1 = 1$
	$51 - 8 = 43$	$a_1 = 2$
	$43 - 8 = 35$	$a_1 = 3$
	$35 - 8 = 27$	$a_1 = 4$
	$27 - 8 = 9$	$a_1 = 5$
	$19 - 8 = 11$	$a_1 = 6$
	$11 - 8 = 3$	$a_1 = 7$
$8^0 = 1$	$3 - 1 = 2$	$a_0 = 1$
	$2 - 1 = 1$	$a_0 = 2$
	$1 - 1 = 0$	$a_0 = 3$

- odečítání snižujících se vah dokud to lze a jako číslice se použije kolikrát bylo odečteno $187 \rightarrow 273$ (číslice shora dolů)

- postupné dělení základem - pouze celé čísla

$$13 / 2 = 6 \text{ zbytek } 1$$

$$6 / 2 = 3 \text{ zbytek } 0$$

$$3 / 2 = 1 \text{ zbytek } 1$$

- $1 / 2 = 0 \text{ zbytek } 1$ dělení základem číselné soustavy dokud není rovno 0 - zbytky po dělení jsou číslice čísla v převedené číselné soustavě (číslice zdola nahoru) $13 \rightarrow 1101$
- postupné násobení základem - desetinná čísla

celá čísla zds 5

- bez znaménka rozsah čísel $<0, 2^n-1>$ n počet bitů
- se znaménkem
- **přímý kód** - první bit určuje jestli je kladné 0 nebo záporné 1
- problém jsou 2 nuly kladná a záporná
- **jednotkový doplněk** - záporné číslo je bitovou negací kladného čísla, také 2 nuly
- **dvojkový doplněk** -záporné číslo je bitovou negací kladného čísla + 1
- pouze jedna 0, funguje dobře v aritmetických operacích
- u obou typů doplňků mají záporná čísla první bit 1
- pro převod zpět pokud je první bit 1, tak provede stejnou operaci jako při vytváření doplňku
- **zobrazení čísel posunutím** - k číslu se přičte konstanta reprezentující nulu - posunutí b
- záporná první bit 0
- **BCD** kód - Binárně kódované dekadické číslo
- Číslice dekadického čísla kódovány do 4 bitů - neefektivní (6 kombinací nevyužito) ale pro člověka snadno čitelné

čísla s pevnou řádovou čárkou zds 7

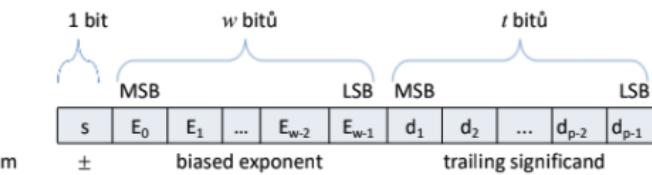
- Řádová čárka má pevně definovanou pozici
- může docházet ke ztrátě přesnosti
- formát Qm.f m počet bitů celočíselné části, f - počet bitů zlomkové části
- převod - klasický převod z dekadické do binární, každý bit má hodnotu 2^{pozice}
- nebo vynásobení čísla 2^f f-počet bitů desetinné části a zaokrouhlení a převádíme takto vzniklé celé číslo
- Převod zpět buď převod bitů před a za , na desítkové číslo nebo celé převod na desetinné číslo a podělit 2 na f
- znaménková čísla - převod stejný - převeďte se jako kladné číslo a následně se vypočte dvojkový doplněk, znaménkový bit je bit navíc nad rámec m.f
- rozsah kladné $0 \leq x \leq (2^m - 2^{-f})$
- rozsah znaménkové $-2^m \leq x \leq (2^m - 2^{-f})$
- Znaménkové mají počet bitů m+f+1, využívá se 2 doplněk - první bit 1 záporné číslo

čísla s pohyblivou řádovou čárkou vyjádřená v binárním, decimálním a hexadecimálním základu
zds 9

- plovoucí desetinná čárka
- číslo je převedeno do binárního tvaru 1,... a desetinná část se ukládá
- exponent je hodnota posunu desetinné čárky aby se dostalo do tohoto tvaru tj. 2^n posun doleva kladný exponent doprava záporný

► $(-1)^s \times 2^e \times m$

s - znaménkový bit (0 plus, 1 minus),
e – exponent kódovaný aditivním kódem (Biased Code),
m – significand - kódovaná přímým kódem (Sign&Magnitude)

- 

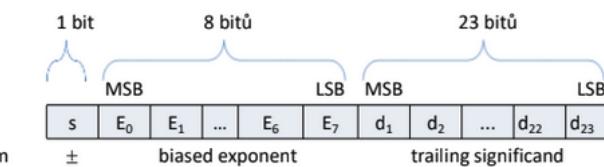
bit
význam
 \pm
biased exponent
trailing significand
- formáty binary definují počty bitů které části zabírají a bias(posun) který se přičítá k exponentu aby byl kladný

FORMÁT	BINARY 16	BINARY 32	BINARY 64	BINARY 128
Počet bitů	16	32	64	128
Z toho bitů:				
pro znaménko	1	1	1	1
pro hodnotu E	5	8	11	15
pro hodnotu T	10	23	52	112
p (přesnost)	11	24	53	113
Bias	15	127	1 023	16 383
$E = e + \text{bias}$	$\langle 1, 30 \rangle$	$\langle 1, 254 \rangle$	$\langle 1, 2 046 \rangle$	$\langle 1, 32 766 \rangle$
e_{\min}	-14	-126	-1 022	-16 382
e_{\max}	15	127	1 023	16 383

Formát binary 32 – rozložení bitů

- Číslo v pohyblivé řádové čárce zabírá 32 bitů:

1 bit pro určení znaménka,
8 bitů pro zakódování exponentu ($w = 8$, bias $2^{w-1} - 1 = 127$),
23 bitů pro zakódování significandu ($p - 1 = 23$, $p = 24$)

- 

bit
význam
 \pm
biased exponent
trailing significand
- převod čísel
 - převod celočíselné a zlomkové části na binární čísla
 - **Převod na normalizovaný binární tvar $(-1^S) \times m \times 2^e$:**
 - m - spojení celočíselné a zlomkové části pouze bity za prvním bitem hodnoty 1
 - e - exponent posouvající desetinnou tečku za první 1 tj do formátu 1,0001110 atd, hodnota je počet bitů celočíselné části - 1
 - $e + \text{bias}$ pro konkrétní kódování
 - dá se vše dohromady (čísla co nejvíce doleva nuly se doplňují vpravo)
 - při převodu z binary tvaru se rozdělí na části, e - bias, získám normalizovaný

binární tvar s tím že před m dám 1,m jednička se při kódování vynechala

- b) Určete hodnotu čísla $(B980)_H$.

	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	...	2^{-10}	
1	0	1	1	1	0	0	1	1	0	...	0 0
\pm	biased exponent					trailing significand					

- Číslo $(B980)_H$ ve formátu BINARY 16: $1011\ 1001\ 1000\ 0000$
- Exponent E v posunutí: $E = 01110_B = 14_D$
- Exponent e: $e = E - \text{bias} = 14 - 15 = -1$
- Normalizovaný tvar čísla binárně: $-1,011 \cdot 2^{-1}$
- Hodnota čísla dekadicky: $-0,6875$
- Normalizovaný tvar čísla dekadicky: $-6,875 \cdot 10^{-1}$

o

aritmetika s čísly v různých reprezentacích zds 6, 10

- celá čísla a pevná řádová čárka

Sčítání

- ▶ Algoritmus je stejný jako v dekadické soustavě:
 - čísla sčítáme postupně po jednotlivých rádech,
 - pokud je součet číslic dvouciferný, je významnější číslice přenesena a přičtena při sčítání číslic vyššího rádu
 - $0+0+0=0$, $0+0+1=1$, $0+1+1=10$, $1+1+1=11$

Odčítání

- ▶ Algoritmus může být stejný jako v dekadické soustavě:
 - pokud je na dané pozici menšenec menší než menšitel, pak si lze od vyššího rádu vypůjčit jedničku,
 - výpůjčka je následně odečtena od vyššího rádu.
- ▶ Operaci odčítání převádíme v počítači na sčítání.

Násobení

- ▶ Algoritmus je podobný jako v dekadické soustavě:
 - Stanoví se znaménko součinu na základě znamének činitelů.
 - Čísla se násobí jako čísla kladná:
 - každým bitem čísla B vynásobíme číslo A
 - mezi výsledky postupně sčítáme.

Dělení

- ▶ Algoritmus je podobný jako v dekadické soustavě:
 - Stanoví se znaménko podílu na základě znamének dělence a dělitele.
 - Podíl části dělence a dělitele může mít hodnotu 0 nebo 1:
 - část dělence \geq dělitel \Rightarrow výsledek je 1
 - část dělence $<$ dělitel \Rightarrow výsledek je 0

- při sčítání v pevné řádové čárce se ve výsledku zvýší počet bitů pro celočíselnou část o 1
- v pevné řádové čárce může dojít k přetečení celkem mám např. Q4.3 i se znaménkovým bitem 8 bitů, jakýkoliv bit navíc vzniklý při výpočtu ignoruji
- v pohyblivé řádové čárce

Sčítání a odčítání čísel ve výměnném binárním formátu

- ▶ Operandy vyjádříme ve tvaru $v = (-1)^s \times 2^e \times m$
 - » Operandy jsou zadány ve výměnném binárním formátu.
- ▶ Sjednotíme exponenty obou operandů, jsou-li různé:
 - » Menší exponent je upraven na vyšší (significand je posouván doprava, exponent se inkrementuje).
- ▶ Použijeme klasickou binární sčítačku:
 - » bity significandu bereme jako integer číslo (v měřítku), přidáme znaménkový a integer bit,
 - » pro záporná čísla se používá dvojkový doplněk,
 - » operaci odčítání převедeme na sčítání.
- ▶ Výsledek upravíme závěrečnými kroky, tj. provede se:
 - » normalizace,
 - » nastavení výjimek,
 - » zaokrouhlení.
- ▶ Výsledek se uvede ve výměnném binárním formátu. [Dec](#)
-
- převedu číslo do tvaru $1, \dots * 2^e$
- musí být stejné exponenty aby se dalo sčítat - sjednotím exponenty a posunu tak, exponenty se tedy nesčítají sčítá se ve formátu $1,..$ tedy po přenásobení exponenty
- pokud je záporné číslo musím před sčítáním převést na 2 doplněk

kódování znaků zds 8

- **ASCII**
- tabulka znaků, 7 bitový kód, každá hodnota reprezentována znakem v tabulce
- rozšíření ascii code page - rozšíření o 1 bit obsahující znaky konkrétního jazyka, byly vytvořeny různé code page
- **UNICODE**
- tabulka všech existujících abeced
- prostor je 21 bitů - skládá se z 17 rovin o 2^{16} kódových pozic
- každá rovina obsahuje jiné znaky (1 - používané abecedy, 2 - hieroglyfy, ..)
- různé typy kódování UTF-8, 16, 32 - podle velikosti částí
- v **UTF-8** stačí pro nejčastěji používané znaky zakódování do 1 bajtu

Kódování UTF-8 2/3

- ▶ Princip kódování:
 - Znaky jsou kódovány dle tabulky do bytů. Pro kód znaku **musí být použita nejkratší možná sekvence bajtů**.
 - Znaky U+0000 až U+007F jsou kódovány jako bajt, nejvíce významový bit je nula. UTF-8 přímo odpovídá Unicode pozici.

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

- Kódové pozice vyšší než 127 jsou kódovány do sekvence bajtů.
- první bajt určuje z kolika bajtů se skládá, další začínají prefixem 10
- při vytváření kódu se prvotní nuly můžou vynechat, 0 se doplňují dopředu aby byly zaplněny bity použitých bajtů
- **UTF-16**

Kódování UTF-16 1/3

- ▶ Znak je kódován do jednoho nebo dvou 16-ti bitových slov.
- ▶ Výhody:
 - úspornější než UTF-32
- ▶ Nevýhody:
 - proměnlivá délka kódování, některé znaky jsou široké 2 bajty, jiné 4.
- ▶ Využití - základní kódování ve velkém množství OS.

Unicode kód od - do	Binární zápis znaku v UTF-16		
U+000000 – U+00FFFF	xxxxxxxx xxxxxxxx		
U+010000 – U+10FFFF	110110xx xxxxxxxx	110111xx xxxxxxxx	(Surrogate pair)
	Vedoucí slovo (Leading)	Koncové slovo (Trailing)	

-
- UTF-8 min velikost znaku 1 B maximální 6 B - nejfektivnější protože většina znaků je krátká
- UTF-16 min velikost znaku 2 B maximální 4 B
- UTF-32 min velikost znaku 4 B maximální 4 B

Příklad otázky: Jako programátor máte vyřešit úlohu předávání dat mezi dvěma aplikacemi. Jaké formy předávání dat/komunikace mezi aplikacemi znáte, na základě čeho byste vybrala/vybral nevhodnější formu a jak byste ji realizovala/realizoval ve vámi zvoleném programovacím jazyce?

Softwarové inženýrství (SWI, VIS)

• Životní cyklus vývoje software (typické fáze, jejich náplň, základní modely vývoje).

- softwarové inženýrství je aplikování systematického, disciplinovaného, kvantifikovaného přístupu k vývoji a údržbě softwaru
- softwarové inženýrství spojuje požadavky od zákazníka a computer science - výstupem je návrh a vývoj softwaru
- získání požadavků, návrh a analýza (analýza a design), vývoj, testování, nasazení, údržba

str 126

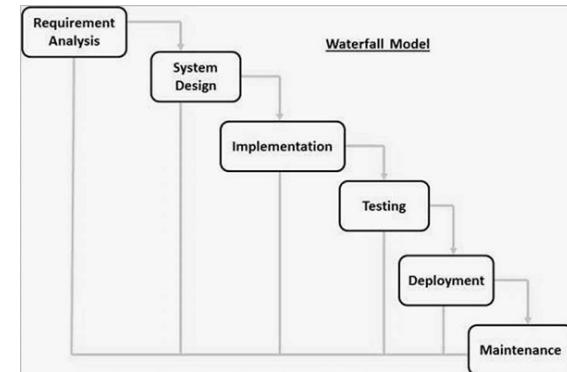
- získání požadavků - kompletní analýza a zaznamenání požadavků zákazníka, oproti těmto požadavkům se bude validovat výsledný sw, důležité porozumět jak to má fungovat a zaznamenat to, UseCase diagramy, Activity diagramy, class diagramy
 - funkční požadavky - co to má dělat
 - nefunkční požadavky - na čem to má fungovat, použitelnost, spolehlivost, přesnost
 - FURPS model (klasifikaci atributů kvality software)
 - Functionality - jaké funkce systém nabízí, bezpečnost
 - Usability - interakce s uživatelem, vzhled, dokumentace
 - Reliability - dostupnost, četnost/závažnost chyb
 - Performance - rychlosť, efektivita, spotřeba zdrojů
 - Supportability - životnost, rychlosť opravy, flexibilita-modifikace systému

str 181

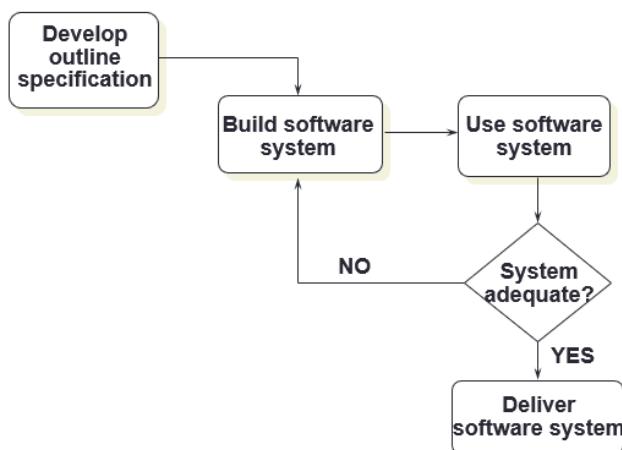
- analýza a návrh - specifikuje co bude systém umět, dle požadavků zákazníka, class diagramy, sekvenční diagramy (diagramy spolupráce)
 - popis objektů (samostatná identifikovatelná jednotka má identitu a chování, vnitřní a vnější pohled tj. vnitřní reprezentace a chování, rozhraní), systém je tvořen komunikací objektů
 - řeší jak bude sw implementován
- návrh (design) - specifikuje jak bude systém realizován
 - Mapování analytických modelů do souboru softwarových komponent s přesně definovanými interakcemi na základě architektury systému a již existujících komponent - systémová architektura, návrhové vzory, softwarové komponenty
 - řeší jakým způsobem bude analytický model implementován
- vývoj (implementace)
 - vývoj sw dle návrhu
 - přetvoření designové architektury do programovacího jazyka
- testování - funkčnosti a jestli splňuje všechny požadavky
 - verifikace - testování jestli funguje správně
 - validace - jestli splňuje všechny požadavky
- nasazení aplikace do reálného provozu - diagramy nasazení a komponent
- údržba - odstraňování chyb, případné rozšiřování nebo úpravy

Modely vývoje str 54

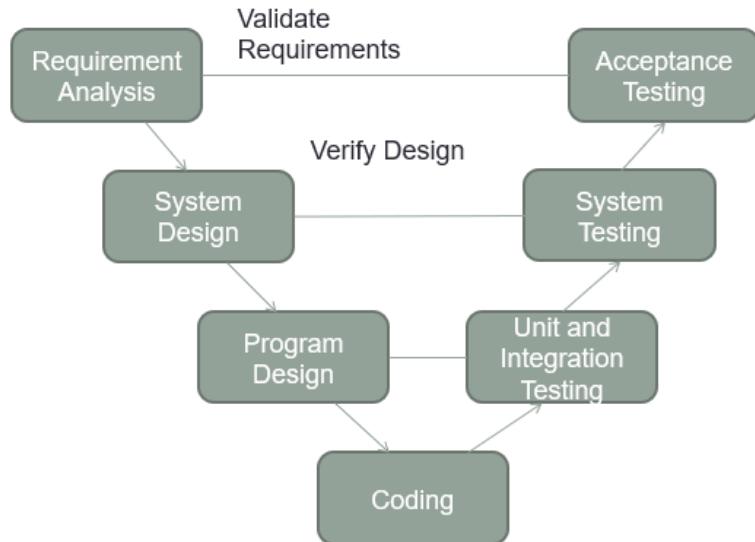
- existuje několik sw procesů tj. souhrn kroků pro vývoj sw které jej zefektivňují
 - vodopádový model



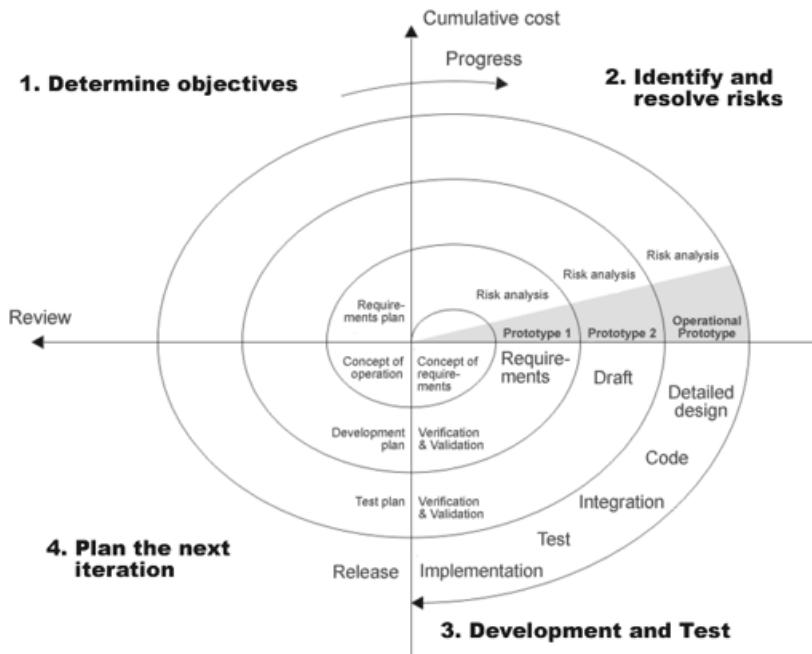
-
- Přesná posloupnost kroků vývoje, k dalšímu se jde až je dokončen předchozí krok
- musí být neměnné požadavky, trvá dlouho než vidím výsledek nebo chybu
- průzkumnické programování



-
- Princip spočívá v tom, že se vytvoří nějaký nástřel specifikace systému, systém se implementuje, pak se začne nějakým testerem testovat, najdou se nedostatky a ty se zakomponují do systému. Takto se pokračuje dokud se už neobjevují v systému nedostatky a jsou splněny všechny požadavky zákazníka
- V model



- - Varianta vodopádového modelu s kladeným důrazem na testování. Dle diagramu se nejprve začínají s unit (jednotka např. třída jestli funguje sama o sobě správně) a integračním (komunikace mezi unit) testováním a postupně se jde výš až k acceptance testování (funguje systém tak jak má?).
 - spirálový model



- - Spirálový model představuje tzv. iterativní vývoj, kde každá iterace obsahuje 4 fáze: stanovení cílů, identifikace a kalkulace rizik, vývoj a testování, plánování další iterace.
 - Jednotlivé iterace můžou vypadat takto: Výsledkem první iterace je koncept systému. Výsledkem druhé iterace jsou definované požadavky. Třetí iterace vytvoří sw návrh a čtvrtá iterace detailní návrh.
 - RUP - [SZZ \(kajfosz.cz\)](http://SZZ.kajfosz.cz)
- U standardních metod vývoje trvá dlouho než získáme prototyp softwaru, který je možno

- ukázat zákazníkovi a dlouho trvá než se projeví chyby v analýze, proto vznikly agilní metody
- Agilní metody vývoje - bez striktních předpisů, větší zaměření na vývoj než dokumentaci, nové požadavky v průběhu vývoje :
 - extrémní programování
 - Agilní způsob vývoj, který umožňuje tvořit rychle kvalitní software a zároveň udržuje vysokou kvalitu života vývojářského týmu (rychlosť vývoje NENÍ založena na tom, že každý programátor programuje 13 hodin denně 7 dní v týdnu aby to bylo rychle).
 - založeno na 12 principech
 - plánování,
 - malé release každé 2 týdny,
 - tests-first přístup(první testy pak metody které jimi projdou),
 - metafore(komponenty systému mají přezdívky),
 - jednoduchý návrh(jen nezbytný kód, nic navíc),
 - refactoring(vylepšování návrhu v každé fázi vývoje),
 - párové programování(2 programátoři na 1 mašině),
 - kolektivní vlastnictví(kdokoli muze zmenit cokoli kdykoli),
 - neustálá integrace(build systému několikrát denně),
 - 40-hodinový týden(programatori pracují max 40 hodin za týden),
 - on-site zákazník(zákazník je neustále k dispozici k zadovězení otázek), programovací standard(všichni programátoři píšou kód stejným stylem)
 - SCRUM
 - Agilní přístup k vývoji. Založen na extrémním programování a rozvíjí jej.
 - Týmové aktivity řeší SCRUM Master. Vývojářský tým se podílí na plánování. Úkoly si vývojáři berou sami, nejsou jim přidělovány. 15minutový SCRUM meeting každý den.
 - Iterační vývoj, rozdělen na sprints (max 30 dní), na konci každého sprintu musí být spustitelná verze sw.
 - SCRUM meetingy
 - daily SCRUM, sprint planning meeting, sprint review meeting, sprint retrospective
 - Velký SCRUM meeting na začátku sprintu - plánovací a na konci - zhodnocení co se udělalo + možná prezentace zákazníkovi
 - SCRUM artefakty
 - product backlog
 - Vysokoúrovní dokument popisující, co má systém dělat, jeho požadavky.
 - sprint backlog
 - Detailní dokument, popisující aktuální sprint.
 - burn down
 - Veřejný dokument popisující co je ještě potřeba udělat.
- Rigorózní metody vývoje
 - najednou požadavky, najednou analýza a design, najednou celá implementace, najednou celé testování
 -

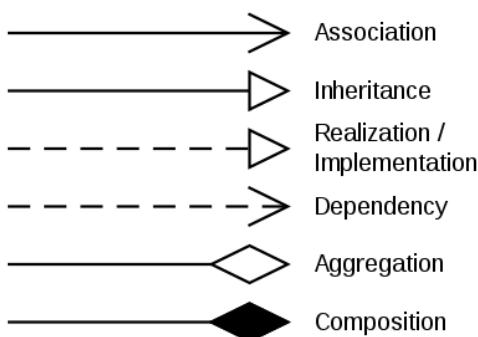
- využitím sw procesů bych se měl vyhnout těmto problémům : složité rozšíření a údržba, skryté chyby, špatná kvalita, nekoordinované týmové úsilí, problémy s nasazením tyto problémy vznikají z těchto důvodů - nedostatečná specifikace požadavků, nedostatečná komunikace, špatné testování, nekonzistence v požadavcích, nekontrolovaní změn
- sw. procesy řeší aby tyto “symptomy” pro vznik problémů nenastaly
- je také vhodné využívat úspěšné postupy - best practices - iterativní vývoj, správa požadavků, využití component based architecture, vizuální modelování, verifikace sw kvality, řízení změn
- popis viz. sešit

- **UML (typy diagramů, statický náhled na systém, dynamický náhled na systém, mapování na zdrojový kód, využití v rámci vývoje).**

Unified modeling language je grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci softwaru. Nabízí způsoby pro návrh systémů včetně prvků jako business procesy, systémové funkce až po databázová schémata. UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu softwaru.

Nejpoužívanější součástí uml jsou diagramy

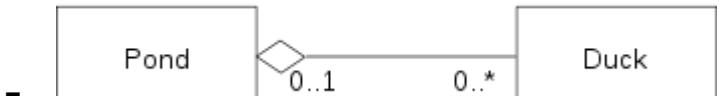
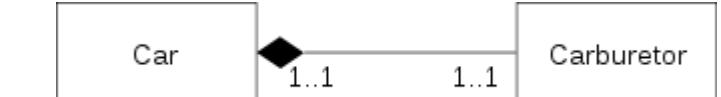
- diagramy
 - UseCase - případy užití - popisuje uživatele a funkce systému které užívají (přehled co kdo v systému dělá) - využití zápis požadavků
 - Activity - symbolický popsán pracovní postup nebo činnost, podobné vývojovému diagramu, operace, podmínky, paralelismus - využití zápis požadavků
 - Class - statický náhled na systém třídy a vazby mezi nimi - využití zápis požadavků nebo analýza a design
 - vystupují v něm entity(nějaký artefakt vzniklý při nějaké činnosti např. ORDER) a tzv. workers (lidi, kteří dělají nějaké úkoly, např. CUSTOMER)
 - vazby mezi třídami jsou většinou pojmenované (CUSTOMER -defines- ORDER)
 - vztah 1:N mezi třídami je zakreslen * u třídy, která se ve vztahu opakuje
 - existují různé typy vztahů mezi objekty



-
- asociace - jednoduchý vztah mezi 2 třídami - Zaměstnanec *pracuje pro Firmu*

- agregace

- vztah mezi komponentami, které tvoří nějakou skupinu/celek, a komponentou, která tuto skupinu komponent zastupuje
- dva druhy agregace



- aggregate

- část, která tvoří celek může vystupovat i v jiných aggregate
- Car - CarDatabase (Car může vystupovat i v jiných agregacích)

- composite

- část, která tvoří composite, nemůže být součástí jiného composite
- Carburator - Car

- dependency

- Jednosměrný vztah, který znamená, že změna nezávislého objektu může způsobit nutné změny v závislém objektu.
- vztah klient -> dodavatel

- generalization

- vztah mezi více obecným objektem (rodičem) a více specifických objektem(potomkem)
- Person <- Student

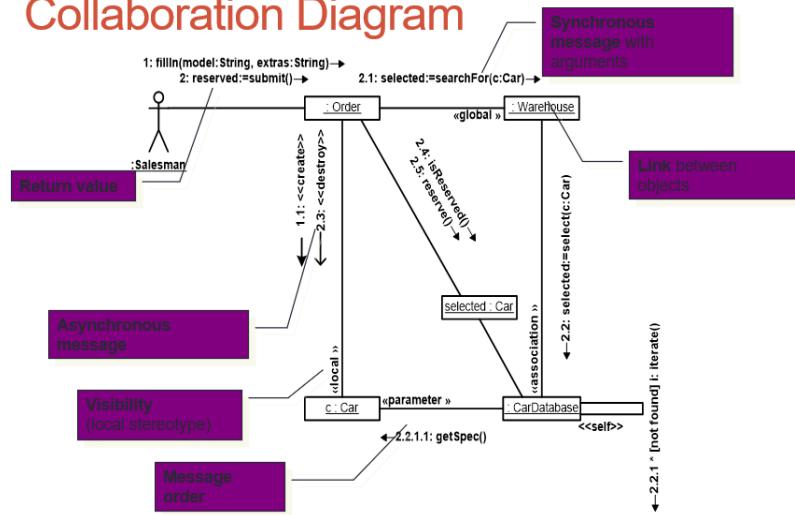
- sekvenční - popis aktivity jako objekty, komunikace mezi nimi a jejich životní cykly - využití analýza a design , popisuje to co activity diagram ale z pohledu objektů

- sekvenční diagram zobrazuje posloupnost kroků na časové ose, která jde shora dolů

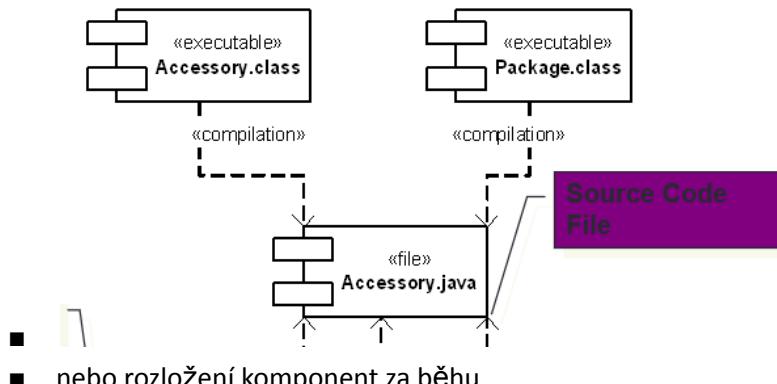
- spolupráce (collaboration) - jiný zápis sekvenčního diagramu

- kolaborační diagram zachycuje posloupnost kroků očíslováním
 - 1. fillIn (nazevAuta : String)
 - 1.1 <<create>> c:Car
 - 2. reserved = submit()
 - 2.1 selected = searchFor(c : Car)

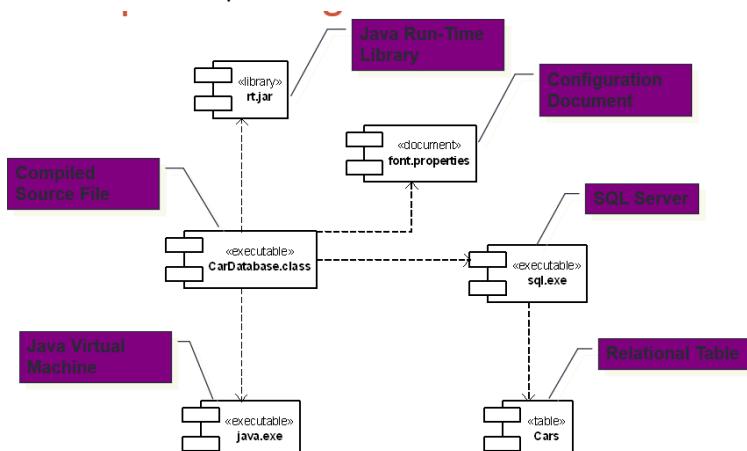
Collaboration Diagram



- stavový diagram - životní cyklus objektu nebo chování - stavy a přechody mezi nimi, dynamický náhled na systém - využití analýza a design
 - jsem schopný jej vytvořit na základě sekvenčního diagramu
- nasazení - kde jaká komponenta poběží - využití při nasazení
 - zachycuje strukturu fyzických komponent systému - servery, síť, pracovní stanice
- komponent - komponenty softwaru a vzájemná komunikace - využití při nasazení
 - může zachycovat binárky, které se komplilují do nějakého jednoho souboru

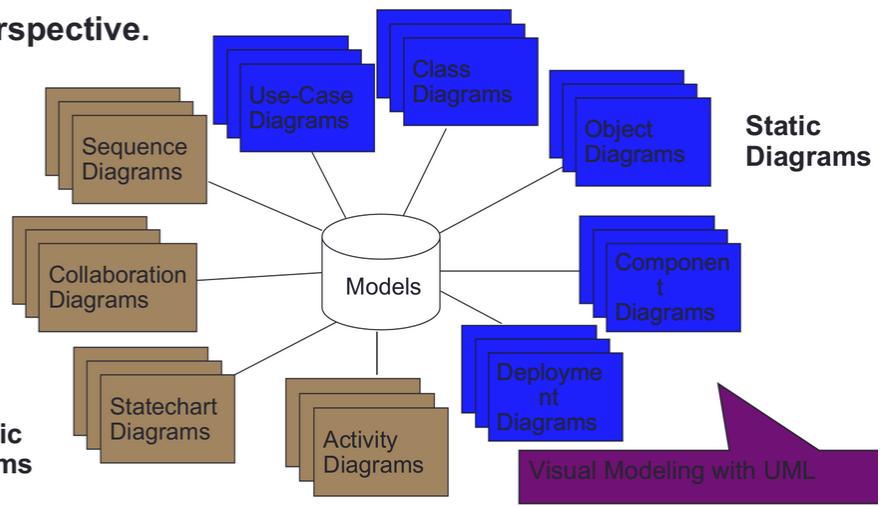


■ nebo rozložení komponent za běhu



- náhledy na systém

perspective.



- statický pohled na systém
 - v čase neměnné části
 - viz. diagramy, v čase neměnné stále stejné (nejsou žádné proměnné stavů)
 - Diagram tříd specifikující množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém.
 - zachycují se části které jsou neměnné
- dynamický pohled na systém vondrák kapitola 5.5
 - při běhu dochází ke změnám (stavů, instancí, větvení)
 - např. sekvenční diagram - vytváření a zánik objektů
 - diagram aktivit - větvení alternativy
 - stavový diagram - změny stavů objektů
 - popisuje celý životní cyklus objektů se změnami stavů
- Mapování na zdrojový kód
 - diagramy z analýzy a designu lze převést na zdrojový kód

Analysis & Design	Source Code (Java)
Class	Class
Role, Type, Interface	Interface
Operation	Method
Attribute (Class)	Static variable
Attribute (instance)	Instance variable
Association	Instance variables
Dependency	Local variable or argument in method or return value
Interaction between objects	Call to a method
Use Case	Sequence of calls
● Package/Subsystem	Package
● využití v rámci vývoje	<ul style="list-style-type: none"> ○ Diagramy umožňují členům vývojářského týmu lépe nebo rychleji pochopit chování/funkce systému. Např. návrh systému dělá jeden tým, ale samotnou implementaci už provádí jiný tým. Tak díky těm diagramům to můžou ti, co budou implementovat rychleji pochopit o co tam jde.

• **Informační systém (životní cyklus, návrh a architektura informačního systému). VIS 0,1**

- Životní cyklus - Analýza a návrh , Implementace, Testování a ladění, Dokumentace, Instalace, Údržba
- <http://www.cs.vsb.cz/benes/vyuka/pte/prednasky/01-zivcykl.pdf>
- Informační systém je propojení informačních technologií a lidských aktivit směřující k zajištění podpory procesů v organizaci.
- V širším slova smyslu se jedná o interakci mezi lidmi, procesy a daty. Informační systém je určen ke zpracování (získávání, přenos, uložení, vyhledávání, manipulace, zobrazení) informací.
- doména informačního systému - skupina souvisejících věcí z pohledu zákazníka, uživatele
- Architektura zahrnuje
 - pohled na doménu informačního systému (skupina souvisejících „věcí“ z pohledu zákazníka)
 - pohled vývojáře na globální strukturu systému a chování jeho částí, jejich propojení a synchronizace
 - pohled na přístup k datům a tokům dat v systému
 - propojení komponent, jejich fyzické rozmístění, jeho principy, vztahy s okolím
 - je snaha rozložit aplikační architekturu do více vrstev
- Softwarová architektura představuje především strukturu softwarového systému a pravidla jejího vývoje.
 - statická architektura – umožňuje zachytit pouze pevnou strukturu softwarového systému bez možnosti změn, struktura systému je dána při návrhu a neměnná za běhu systému
 - dynamická architektura – oproti statické architektuře navíc podporuje vznik a zánik komponent a vazeb za běhu systému podle pravidel určených při návrhu, struktura systému se dynamicky mění
 - mobilní architektura – rozšiřuje dynamickou architekturu o mobilní prvky, kdy se komponenty a vazby přesouvají za běhu systému podle stavu výpočtu
- ARCHITEKTURA se zabývá především technickými (jinými než funkčními) a částečně funkčními požadavky, zatímco NÁVRH vychází z čistě funkčních požadavků.
- Architektura řeší implementační záležitosti - jak budou uchovávána data, jak se k nim přistupuje, komponenty a provázání, rozdělení činností do komponent
- Návrh specifikuje části a jak budou fungovat
- systémová architektura (z SWI)
 - organizační struktura a chování systému
 - typické vzory architektur - client server, peer to peer
 - je to konceptuální model, který popisuje strukturu a chování několika softwarových aplikací, síťových zařízení, hardware.
 - pro popis se používá ADL (architecture description language)
 - architekturu popisují sekvenční diagramy? objekty a komunikace

- Návrhové vzory a praktiky pro vývoj informačních systémů (vzory GoF, doménová logika, datové zdroje, objektově-relační chování a struktury, doménově specifické jazyky).

(agregace - vazba mezi celkem a jeho součástí, kdy jeden objekt (celek) využívá služby dalších objektů (součástí). Součásti mohou existovat i bez celku. např. počítač - tiskárna, značeno prázdný kosočtverec

kompozice - stejná vazba, ale součásti nemohou existovat bez celku. např. stát - kraj, značeno černý kosočtverec)

[Design Patterns \(sourcemaking.com\)](http://sourcemaking.com)

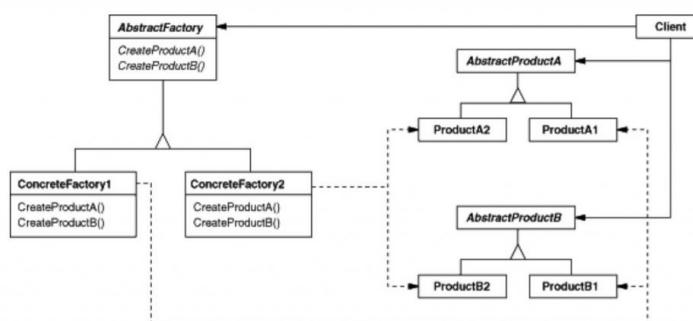
návrhové vzory

- představuje obecné řešení problému, které se využívá při návrhu počítačových programů
- řešení typických problémů v programování, ověřená standardizovaná řešení
- zvyšují přehlednost kódu - programátor který zná návrhové vzory snáze pochopí kód který je využívá
- typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy

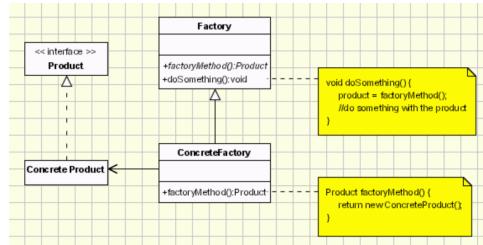
vzory Gof (gang of four - Erich Gamma, Richard Help, Ralph Johnson a John Vlissides)

<http://voho.eu/wiki/navrhovy-vzor/>

- Creational Patterns (vytvářející) -
 - Abstract Factory <http://voho.eu/wiki/abstract-factory/>
 - Obsahuje metody které vrací instanci třídy, deleguje vytváření instancí na třídu
 - je vhodné pro skupinu podobných tříd nebo pro třídu s více konstruktory
 - odstíní uživatele od implementačních detailů (metoda může vyplnit některé parametry konstruktoru)

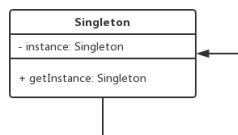


-
- Factory Method <http://voho.eu/wiki/factory-method/>
 - Metoda třídy která vrací novou instance
 - statická metoda - funguje jako abstract factory
 - normální metoda - vrácená nová instance může být ovlivněna stavem (hodnotami) aktuálního objektu



- Singleton <http://voho.eu/wiki/singleton/>

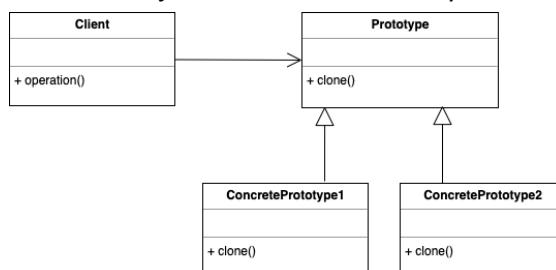
- V systému má existovat třída, která může mít nejvýše jednu globálně sdílenou instanci a zároveň sama centralizuje k této své instanci přístup
- privátní konstruktor a uchování instance ve statické proměnné třídy



■

- Prototype <http://voho.eu/wiki/prototype/>

- V systému se nachází třída, jejíž instance je přijatelnější či efektivnější nějakým způsobem replikovat než vytvářet nové
- využití pokud třída obsahuje nějaká unikátní data (id) nebo nelze vytvořit kopii přiřazením kvůli dynamicky alokovaným datům
- Třída obsahuje metodu **Clone** která vytvoří a vrátí kopii instance

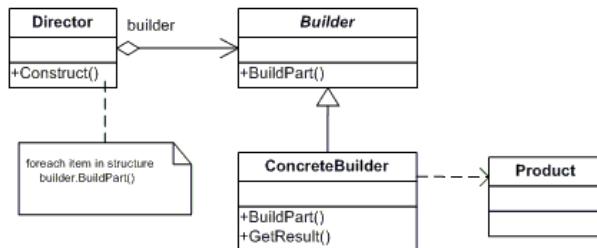


■

- Builder <http://voho.eu/wiki/builder/>

- Využití pokud mám více tříd, které dělají podobnou věc, využívající stejný (samostatnou třídu) společný základ ale každý jej využívá trochu jinak
- Využívají společnou kostru (základ), společnou kostru postupu zachytit a jeho zobecněním snížit duplicitu kódu
- jsou vytvořeny třídy builder, které si vytváří instanci výchozí třídy a pracují s ní
- Builder má metody pro práci s využívanou třídou (zadává svoje parametry) a metodu getResult pro vrácení instance třídy se kterou pracuje
- instance buildera se předává do třídy director, která buildera využívá (mají společné rozhraní)
 - director ví v jakém pořadí má volat metody *buildera* aby sestavil *product*
- interakce 3 typů objektů
 - produkt

- řídící objekt (director)
- výkonný objekt (builder)



■

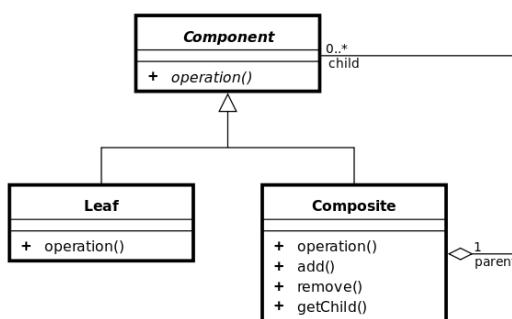
- Structural Patterns (strukturální)

- Adapter <http://voho.eu/wiki/adapter/>

- V systému existuje nevyhovující třída se zastaralým, nekompatibilním, nebo jinak neuspokojivým rozhraním. Funkcionalita existující třídy má být zachována a pouze s minimálním úsilím převedena pod rozhraní nové. Může se objevit i požadavek, aby se stávající třída již neměnila
 - zastaralá třída by v systému mohla dělat zmatek
 - Nevyhovující třídu lze od zbytku systému odstínit tak, že se celá zapouzdří do třídy nové. Tato třída již implementuje nové rozhraní a poskytuje infrastrukturu pro transformaci požadavků od klienta na příslušné metody zapouzdřené třídy. tj. třída obsahující metody v nichž se volají metody staré třídy

- Composite

- Tvoří hierarchickou strukturu skládající se z jednoduchých objektů leaf nebo objektů které mohou obsahovat další objekty composite (např souborový systém)
 - hlavní třída composite ze které se dědí, potomci mohou obsahovat instance composite nebo mohou být listy stromu (neobsahovat další instance)

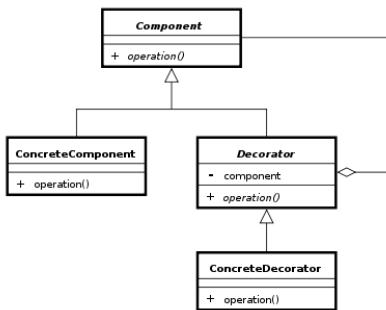


■

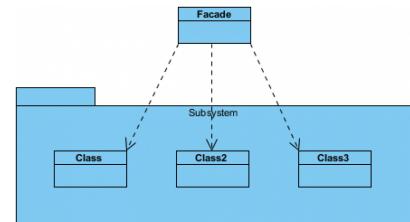
- Decorator <http://voho.eu/wiki/decorator/>

- používá se pro dynamické přidání či změnu funkčnosti nějakého objektu bez nutnosti jeho změny či použití dědičnosti. Jako dekorátor se označuje třída, která obdrží nějakou instanci, kterou obohatí o novou funkcionalitu nebo rozšíří stávající funkcionalitu.
 - Třída obsahuje instanci původní třídy, nabízí její rozhraní a přidává

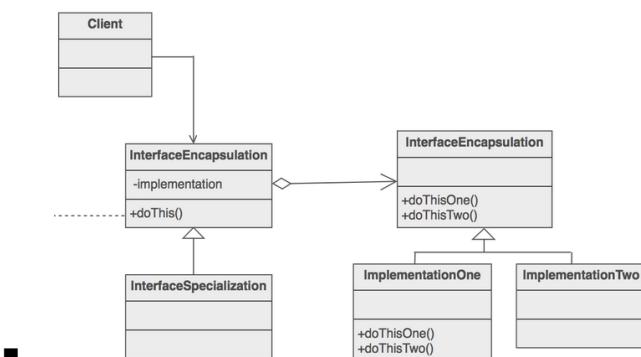
metody navíc

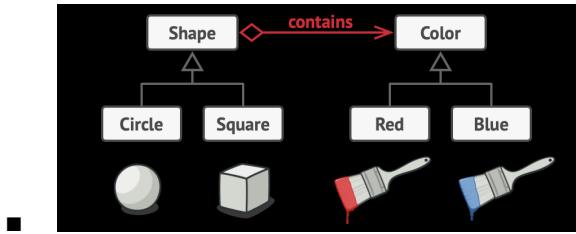


- - Facade (fasáda) <http://vocho.eu/wiki/facade/>
 - Mezi Vrstva, která zapouzdřuje funkčnost více tříd - uživatel nemusí znát více menších tříd, ale stačí mu fasáda která tyto třídy využívá a poskytuje rozhraní
 - např. třídy pro základní aritmetické operace > fasáda poskytuje funkce těchto tříd + složitější operace využívající elementárních operací tříd -
 - Klient by mohl vzít elementární výpočty a vše provést sám - fasáda však koncentruje všechny užitečné výpočty na jedno místo, zabrání duplicitě kódů a v neposlední řadě i odstraní nutnost o elementárních výpočtech vůbec vědět.

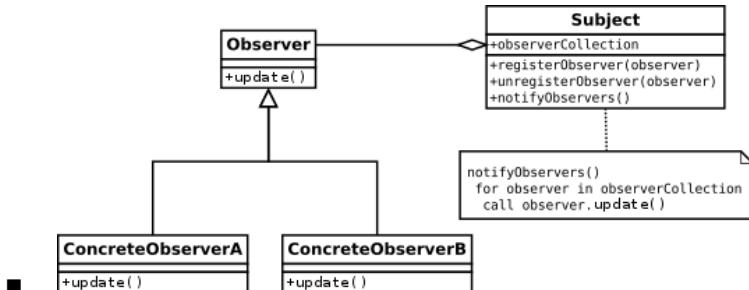


- - Bridge <http://vocho.eu/wiki/bridge/>
 - Chceme oddělit abstrakci od její implementace tak, aby se obě mohly měnit nezávisle.
 - vytvoříme rozhraní, které bude mít různé implementace řešící stejnou činnost ale jinak (např. dle OS, formátu dat, atd.)
 - Klient využívá rozhraní a neřeší o jakou konkrétní implementaci se jedná.



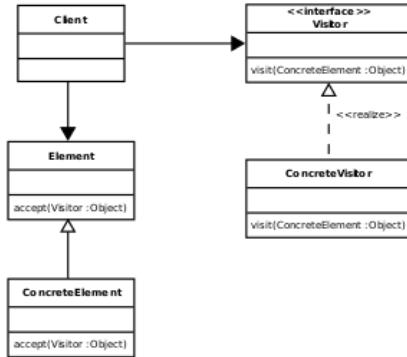


- Behavioral Patterns (chování)
 - Observer
 - používá se v situacích, kdy chceme, aby nějaké objekty dostávaly informaci o změně stavu jiného objektu a zároveň mít možnost za běhu ovlivňovat, které objekty to budou.
 - **pozorovaný** (observable, subject, publisher) - Objekt, jehož změny pozorujeme. Obsahuje množinu pozorovatelů a umožňuje jejich správu - tedy přidávání (*attach*) a odebírání (*detach*). Objekt upozorní všechny pozorovatele na každou zásadní změnu svého stavu.
 - **pozorovatel** (observer, listener) - Objekt, který dostává informaci o změně stavu pozorovaného objektu.
 - pozorovaný obsahuje pole pozorovatelů. Pozorovatel implementuje rozhraní s metodou pro notifikaci - každý pozorovatel si tuto metodu implementuje jinak. Pozorovaný při události volá nad každým pozorovatelem tuto metodu



- Visitor <http://voho.eu/wiki/visitor/>
 - Je definována hierarchie tříd se společným předkem. Tento předek deklaruje metodu, která je v každé podtřídě implementována jinak. Zdrojový kód všech konkrétních implementací se má z podtříd extrahovat a přesunout do jedné nové třídy.
 - Pokud programovací jazyk umožňuje multiple dispatch je jednoduché řešení použít přetěžování metod. Podle instance se použije příslušná metoda
 - V jazycích, které multiple dispatch nepodporují je lepší než velké množství podmínek testujících co to je za instanci využít visitor nebo pokud funkčnost nesouvisí s třídou a chci aby byla mimo
 - spoluprací dvou tříd, které jsou zde označeny jako **Place** a **Visitor**. Třída **Place** požádá vybranou instancí třídy **Visitor** o provedení akce a předá ji sama sebe jako parametr. Třída **Visitor** pak na této instanci zavolá požadovanou metodu, která odpovídá její třídě. (metoda může být rozlišena dle datového typu předané třídy - přetížené metody)
 - Place má metodu accept(Visitor v) - a v ní v.visit(this)

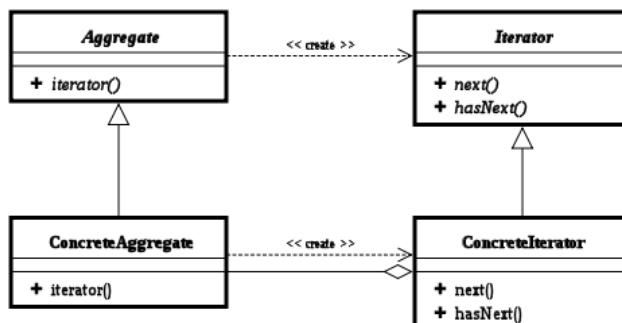
- Visitor má více přetížených metod visit přijímající různé objekty -
visit(Divadlo d) ... visit(Muzeum m)
- v závislosti na typu objektu může provádět různé operace
- k identitě příjemce metody se přidává ještě druhá chybějící informace o odesílateli



■

- Iterator <http://vocho.eu/wiki/iterator/>

- Existuje uspořádaná množina prvků, kterou chceme systematicky procházení. Nezajímá nás, jakou strukturu tato množina má, nebo jak přesně je uložena v paměti. Důležité pro nás je pouze správné pořadí prvků při jejím procházení. Procházení lze například seznam, pole, klíče nebo hodnoty hashovací tabulky, nebo uzly či hrany grafu.
- je mnoho způsobů procházení
- Vytvoříme tedy třídu, která bude zapouzdřovat tuto logiku procházení prvků v určitém pořadí a navenek nám bude poskytovat pouze další prvek v pořadí a informaci o tom, zda je takový prvek k dispozici, abychom mohli procházení ve správný čas ukončit.
- do iterátoru předáme datovou strukturu a ten implementuje procházení metodou next vracející prvek a hasNext
- Navenek nijak neřešíme procházení pouze voláme tyto metody

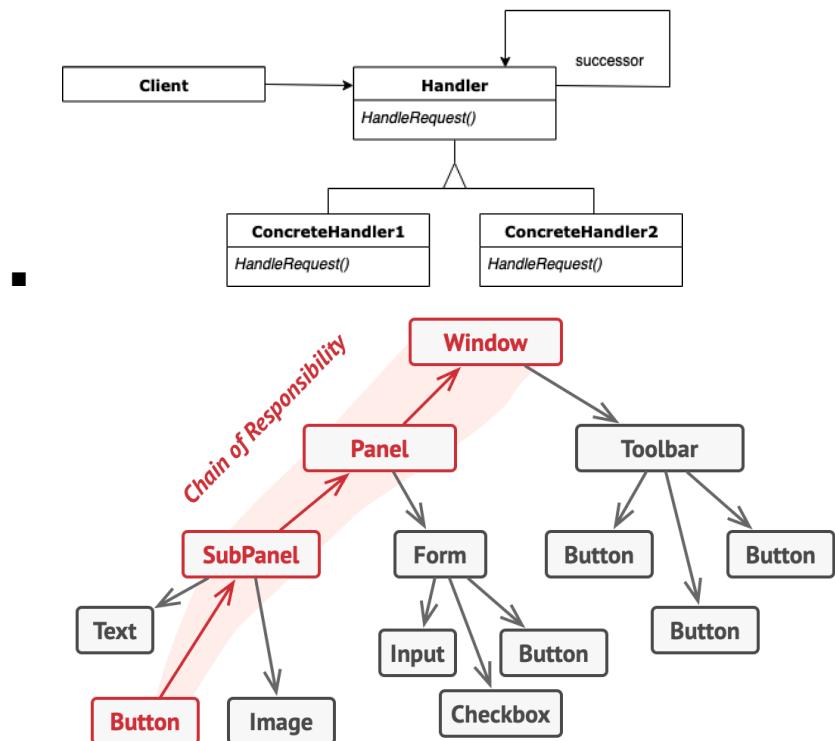


■

- Chain of Responsibility

- předávání požadavku v řetězci vzájemně propojených instancí
- instance zná následujícího v řetězci, kterému předá požadavek
- využívá se proto, aby klient nemusel znát reference na všechny články řetězce, ani pořadí, ve kterém se mají tyto články provolat. O dalším zpracování požadavku totiž mohou články rozhodovat samy.

- Handlery mají metody handle (zpracování) a uložený next Handler.
Provedou nějaké zpracování požadavku a poté jej předají next handleru.

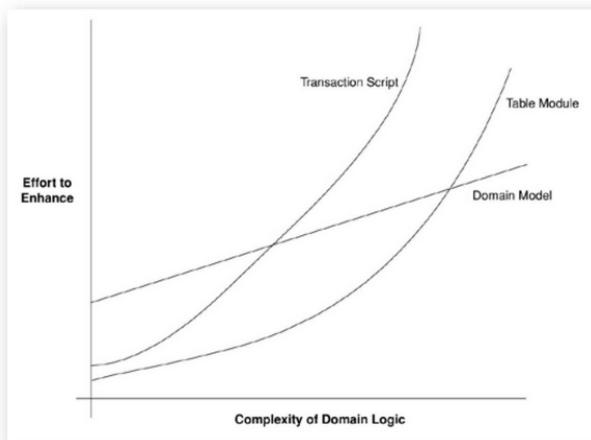


- Interpreter
 - vyhodnocování výrazů jednoduchého formálního jazyka
 - vykonání operace (logické, aritmetické) na základě vstupu
 - (projekt PJP)

doménová logika VIS 2 (vzory Fowler)

- doménová logika popisuje algoritmy nebo byznys logiku pro výměnu informací mezi databází a uživatelským rozhraním
- transaction script
 - organizuje business logiku pomocí procedur, kde každá procedura řeší jeden požadavek z prezenční vrstvy
 - neobjektové programování
- table module
 - jediná instance, která řeší business logiku pro všechny řádky v tabulce/pohledu databáze
- domain model
 - objektový model, který zahrnuje chování i data
 - Doménový model vytváří síť vzájemně propojených objektů, kde každý objekt představuje nějakého smysluplného jednotlivce (třídu a metody)
- service layer
 - definuje hranice aplikace s vrstvou služeb, která stanovuje skupinu dostupných operací a koordinuje odezvu aplikace na každou operaci

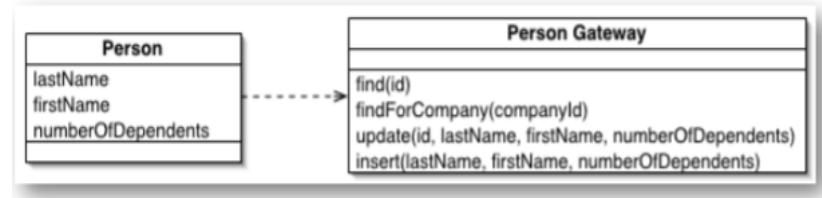
- předávání požadavků mezi službami



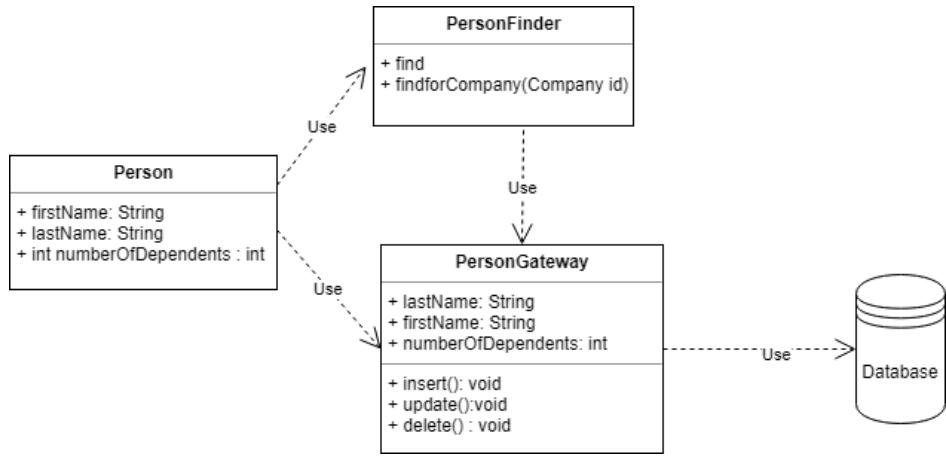
- transaction script a table module je vhodné použít pro malé projekty, čím je projekt větší tím složitější je vývoj
- pro komplikovanější projekty je vhodný domain model, náročnost implementace roste lineárně

datové zdroje VIS 3

- nezávislost doménové logiky na logice přístupu k datům
- přístup k datům z aplikace - přístup do databáze
- Table Data Gateway
 - brána se rozumí objekt, který zapouzdřuje přístup k externímu systému nebo prostředku
 - 1 třída pro veškerou práci s 1 tabulkou, objekt, který se tváří jako vstupní brána do databázové tabulky
 - obsahuje všechny CRUD operace pro přístup k jedné tabulce nebo pohledu
 - Hlavní myšlenkou je poskytnout pro každou databázovou tabulku gateway třídu.

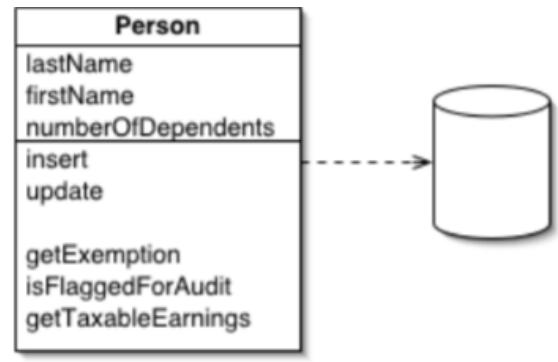


- Row Data Gateway
 - funguje jako objekt, který přesně napodobuje jeden záznam, jeden řádek databáze.
 - objekt který se tváří jako vstupní brána k jednomu záznamu v datovém zdroji, jedna instance na každý řádek
 - Finder načte konkrétní záznam z db a vytvoří instanci třídy Gateway která obsahuje data i CRUD operace (z gateway může získat data třída Person?)



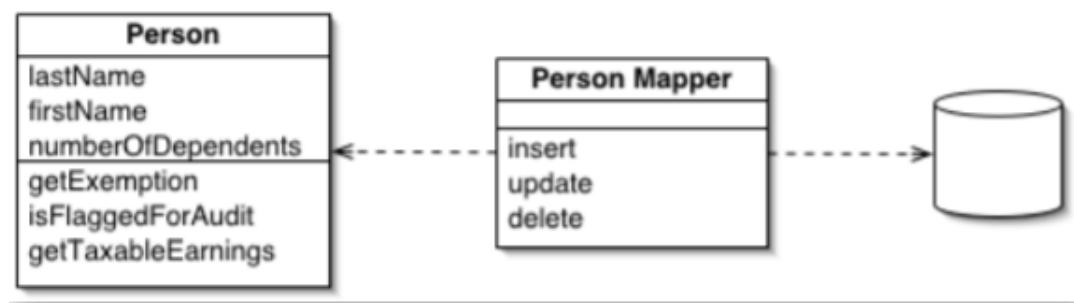
- Active Record

- objekt reprezentuje řádek databázové tabulky a zapouzdřuje přístup k databázi (CRUD) a doménovou logiku - objekt nese data i chování
- objekt, který zachycuje řádek v tabulce/pohledu, zapouzdřuje přístup k databázi (CRUD) a přidává doménovou logiku na data



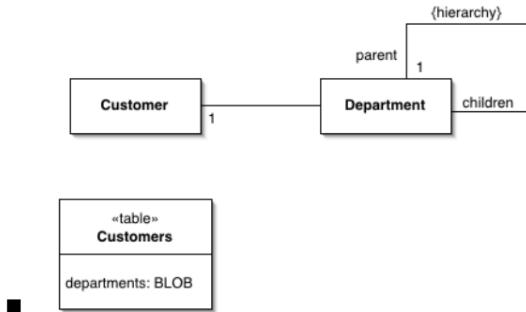
- Data Mapper

- vrstva mapovačů, kteří přesouvají data mezi objekty a databází, zatímco udržují jejich nezávislost samotných od sebe i od mapovačů, objekty neví nic o db a opačně, veškerou komunikaci zajišťují mapovače
- person mapper se staví na úrovni doménové vrstvy, když ridici objekt potřebuje uložit objekt person
- Hlavní odpovědností je přenášet tato data mezi objektem a relační databází a také je od sebe izolovat. S Data Mapperem nemusí objekty v paměti ani vědět, že existuje relační databáze, nepotřebují mít žádnou znalost databázového schématu. - vše řeší mapper



Objektové relační chování a struktury VIS 4

- Unit of Work
 - Udržuje seznam objektů ovlivněných změnou a koordinuje zápis změn a řešení problému souběžnosti
 - Je vytvořena třída, která bude zaznamenávat změny a až na požadání tyto změny uloží i v trvalém úložišti
 - Třída bude obsahovat informaci o každé změně entity (instance). Bude obsahovat 3 stavy new, dirty(změněná), removed tj. proměnné s kopií instance v určitém stavu. (když provedu změnu uloží se do dirty a v new budu mít pořád původní data)
 - nespouštím spoustu pristupu k db rozmístěných různě po kodu, ale nejakým způsobem si je hromadím a v jistý moment je všechny spustím najednou => větší efektivita
- Identity Map
 - Zajišťuje že každý objekt bude načten pouze jednou, takže udržuje mapu načtených objektů. Vyhledává objekty v mapě když na ně odkazuje
 - Využití když v různých částech programu potřebujeme upravovat stejnou entitu
- Lazy Load
 - Objekt, který neobsahuje všechna data, která potřebujete, ale ví jak je získat.
 - Využití pro načítání entit, dat z databáze až v době kdy je to potřeba - zvyšuje se rychlosť aplikace protože se nemusí vždy načítat všechny entity, data, ale např. až nastane určitá situace (spustí se Use Case atd.?)
 - Lazy Load se hodí použít pokud potřebná data mám ve více tabulkách a bylo (i při načítání najednou) potřeba více dotazů
- struktury ?
 - identity field
 - třída namapovaná na tabulku v DB obsahuje ID
 - foreign key mapping
 - třída namapovaná na tabulku v DB obsahuje odkaz (např. pointer) na třídu, se kterou je v tabulce v relaci
 - association table mapping
 - stejně jako *foreign key mapping*, akorát zachycuje M:N relaci v DB
 - dependent mapping
 - třída, která řeší DB operace pro tabulku je řeší taky pro další tabulku, která je s tou první v relaci
 - např. tabulka Album je v relaci s tabulkou Pisnicka
 - AlbumMapper řeší DB operace jak pro Album tak pro Pisnicka
 - embedded value
 - zabalení vícero sloupců z tabulky do třídy
 - např. tabulka Employments má sloupce startDate a endDate a já je zabalím do třídy DateRange a ta bude jako property třídy Employments
 - Serialized LOB
 - v kódu mám dědičností nebo skládáním nějakou hierarchii tříd (třeba i tvořící cyklus) ale v DB místo toho abych dělal tu samou hierarchii v tabulkách tak celou tu hierarchii serializuju jako BLOB



Doménově specifické jazyky

- programovací jazyk zaměřující se na řešení obecných problémů
- sloužící pro lepší **komunikaci v týmu**
- pomáhá přemostit propast mezi zúčastněnými stranami softwarového vývoje a programátory
- může zjednodušit porozumění komplikovaného bloku kódu, a tak zlepšit produktivitu vývojářů
- zlepšuje komunikaci s doménovými experty
- měl by být srozumitelný i pro neprogramátory
- umožňuje lepší komunikaci se zákazníkem

Externí

- vlastní syntaxe
- jazyk oddělený od hlavního jazyka aplikace

Interní

- zvláštní způsob použití univerzálního jazyka
- vnořené do hostovacího jazyka
- syntaxe se řídí gramatikou hostovacího jazyka

Příklad otázky: Vysvětlete, v jaké situaci je vhodné použít vzor Composite, použijte UML diagram pro jeho schématické vyjádření a promítněte tento diagram do zdrojového kódu ve vámi zvoleném programovacím jazyce.

Databázové systémy (DS I, DS II)

- **Datové modely (relační datový model: definice, normální formy, funkční závislosti; objektově relační datový model: základní rysy).**

Datový model - množina konceptů, které mohou být použity pro popis struktury databáze

- Konceptuální - model nezávislý na použité technologii databází
- Databázový - závislý na technologii, ale ne konkrétním jazyce (pro relační databáze -> Relační datový model)
- Fyzický - závislý na jazyce, reprezentuje fyzické uložení dat na médiích

Konceptuální model obsahuje

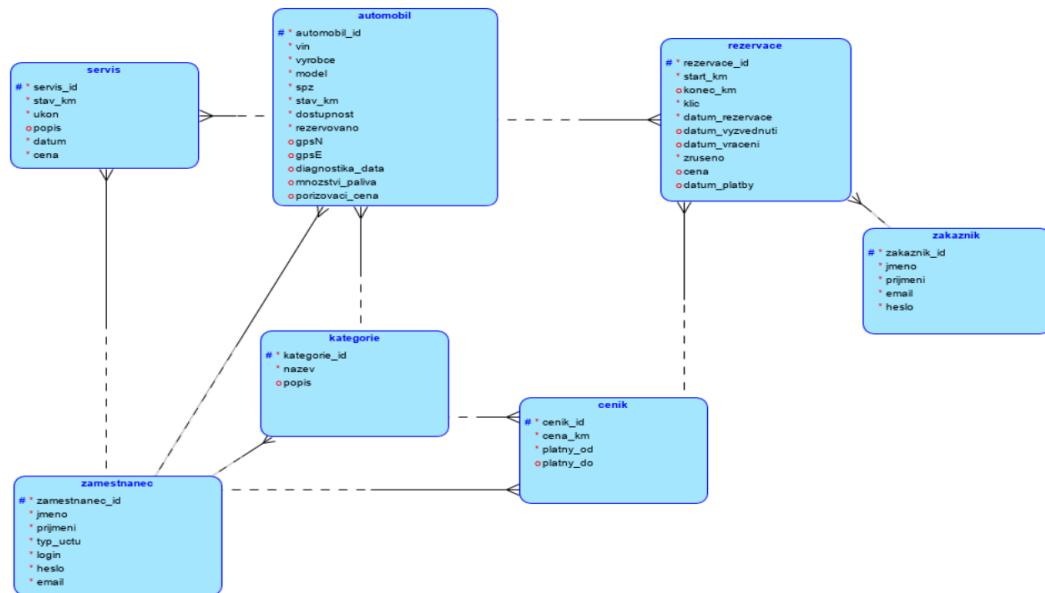
- Entita - objekt reálného světa - prakticky záznam v tabulce
- Atribut - vlastnost entity - prakticky sloupec v tabulce
- Entitní typ - množina entit se stejnými atributy - prakticky datová tabulka
- Klíč - jeden či více atributů, které jednoznačně určují entitu v množině entit
- Mezi entitními typy se nazývá **vztah** (relace)
 - Kardinalita vztahu - dělení vztahů na 1:1, 1:M, M:M
 - Povinnost vztahu
- Konceptuální model je tedy zobrazován ER diagramem

Relační model

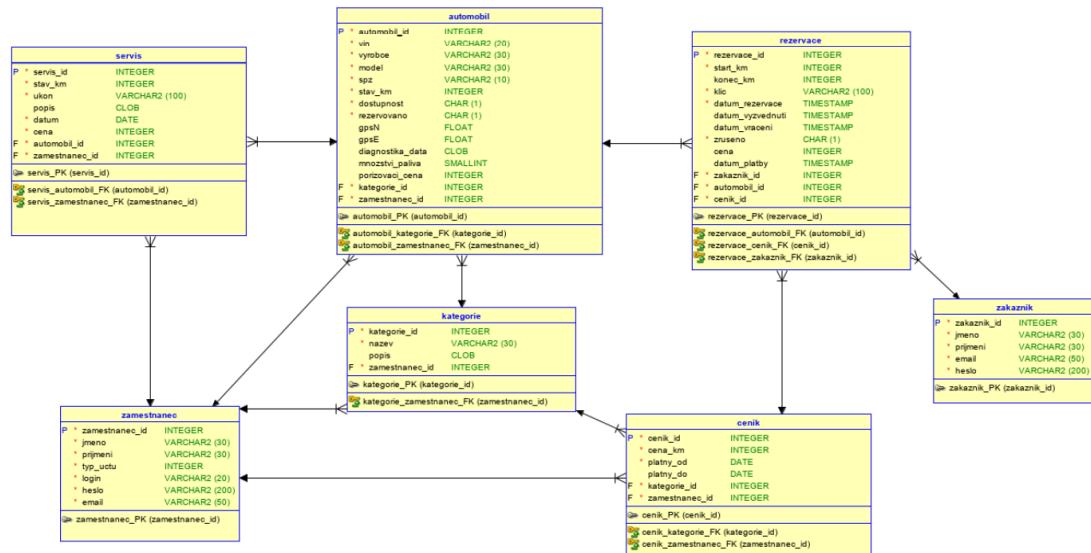
- nejrozšířenější způsob uložení dat v db
- sdružuje data do relací (tabulek), které obsahují n-tice (řádky s daty)
- tabulka definuje strukturu záznamů - přesně stanovené sloupce/atributy s jednoznačným názvem, datovým typem a rozsahem
- vazby mezi tabulkami lze realizovat pomocí sloupců stejného datového typu
- Relační model klade velký důraz na zachování integrity dat. Zavádí pojmy referenční integrita, cizí klíč, primární klíč, normální tvar apod.

Relační databáze využívají jako základu tabulek (relací), které jsou propojeny předem nastavenými vztahy. Relace je konečná podmnožina kartézského součinu domén jednotlivých atributů. Reálná databázová tabulka se od definice mírně liší. Relace je množina a to znamená, že vylučuje možné duplicity. V běžné databázové tabulce ale duplicitu existovat mohou např. pokud není definovaný PK nebo výsledek funkce SELECT bez použití funkce DISTINCT. Atributy tabulky určují vlastnosti objektů, které se do tabulky budou vkládat. Při návrhu se definuje struktura tabulek - datové typy atributů, integrní omezení. Každá databázová tabulka by měla mít také definovaný primární klíč, jehož hodnotou lze jednoznačně identifikovat záznam. Vztahy se řeší pomocí cizích klíčů, které odkazují na jiné primární klíče z jiných tabulek.

Konceptuální datový model



Relační datový model



Normální formy ds1 - 10

- Pod pojmem normalizace rozumíme proces zjednodušování a optimalizace navržených struktur databázových tabulek.
- Hlavním cílem je navrhnut databázové tabulky tak, aby vykazovaly minimum redundancy.
- Správnost navržení struktur v tomto smyslu lze ohodnotit některou z normálních forem, aby mohla být v NF musí splňovat předchozí NF
- 1NF – žádný sloupec (atribut) tabulky nelze dále dělit na části nesoucí nějakou informaci, prvky musí být atomické
- 2NF – neobsahuje funkční závislosti, tabulka obsahuje pouze atributy (sloupce), které jsou závislé na celém klíči. Složené klíče, rozdělení na více tabulek
- 3NF – mezi neklíčovými atributy (sloupcí) tabulky neexistují žádné závislosti (vztahy).
- Boyce-Codd Normal Form - každá funkční závislost musí mít na levé straně klíč

Funkční závislosti ds1 - 9

<http://lucie.zolta.cz/index.php/ifarmacni-systemy-databaze/47-funkcni-zavislosti>

- Funkční závislost je v databázi vztah mezi atributy takový, že máme-li **atribut Y je funkčně závislý na atributu X píšeme $X \rightarrow Y$** , pak se nemůže stát aby dva řádky mající stejnou hodnotu atributu X měly různou hodnotu Y.
- Pomocí funkčních závislostí můžeme automaticky navrhnut schéma databáze a předejít problémům jako je **redundance, nekonzistence** databáze, zablokování při vkládání záznamů, apod.
- Postup datové analýzy s automatickým navržením struktury databáze je následující:
 - ze zadání zjistíme, co je třeba v databázi evidovat (objekty, atributy, vztahy, integritní omezení,...)
 - funkční analýzou určíme závislosti (vztahy) mezi atributy
 - vytvoříme jednu obrovskou tabulku (relaci), obsahující všechny atributy
 - pomocí funkčních závislostí provedu dekompozici (rozbití) velké relace na menší, které vytvoří výsledné schéma databáze.
-
- Armstrongovy axiomy jsou odvozovací pravidla funkčních závislostí. Pomocí axiomu získáme uzávěr funkčních závislostí spolu s klíči schémat.
- **Uzávěr X^+** je množina všech atributů funkčně závislá na atributech množiny X.
 - **Reflexivita** - je-li $Y \subset X \subset A$, pak $X \rightarrow Y$
 - **Tranzitivita** - pokud je $X \rightarrow Y$ a $Y \rightarrow Z$, pak $X \rightarrow Z$
 - **Pseudotranzitivita** - pokud je $X \rightarrow Y$ a $WY \rightarrow Z$, pak $XW \rightarrow Z$
 - **Sjednocení** - pokud je $X \rightarrow Y$ a $X \rightarrow Z$, pak $X \rightarrow YZ$
 - **Dekompozice** - pokud je $X \rightarrow YZ$, pak $X \rightarrow Y$ a $X \rightarrow Z$
 - **Rozšíření** - pokud je $X \rightarrow Y$ a $Z \subset A$, pak $XZ \rightarrow YZ$
 - **Zúžení** - pokud je $X \rightarrow Y$ a $Z \subset Y$, pak $X \rightarrow Z$

<https://www1.cuni.cz/~obo/skola/databaze.html.cz>

Funkční závislosti představují koncept, který nám umožňuje správně definovat databázová schémata. znamená, že jakmile udáme hodnotu ve výchozích attributech, je tím jednoznačně určen řádek tabulky a tedy i hodnota v těch cílových sloupečcích. Rodné_číslo->{Jméno, Příjmení} značí, že jméno a příjmení člověka je jednoznačně určeno (funkčně závisí) jeho rodným číslem.

Objektově relační datový model db prez 13

<http://lucie.zolta.cz/index.php/ifarmacni-systemy-databaze/74-objektovy-datovy-model>

- (čistě objektové databáze se příliš nevyužívají)
- Objektově-relační datový model, je klasická relační databáze rozšířená o objektově-relační rysy
- Objektové typy a jejich metody jsou uloženy spolu s daty v databázi, programátor tedy nemusí vytvářet podobné struktury v každé aplikaci.
- Metody jsou spouštěny na serveru
- Objektové rysy jsou dnes implementovány ve všech velkých SŘBD.
- Objektové typy a jejich metody jsou uloženy spolu s daty v databázi, programátor tedy nemusí vytvářet podobné struktury v každé aplikaci.

- Programátor může přistupovat k množině objektů jako by se jednalo o jeden objekt.
- Objekty mohou jednoduše reprezentovat vazby kdy jedna entita se skládá z jiných entit (bez nutnosti použít vazeb).
- Metody jsou spouštěny na serveru – nedochází k neefektivnímu přenosu dat po síti.
- objektové datové typy mohou obsahovat data a metody - uživatelsky definované datové typy, triggers, uložené procedury, ..

https://cs.wikipedia.org/wiki/Rela%C4%8Dn%C3%AD_model

https://cs.wikipedia.org/wiki/Rela%C4%8Dn%C3%AD_datab%C3%A1ze#Norm%C3%A1ln%C3%AD_fo rmy

- Dotazovací jazyky (**SQL: jazyk pro definici dat, jazyk pro manipulaci s daty, SELECT – spojení, poddotazy, agregační funkce; procedurální rozšíření SQL: PL/SQL, T/SQL, obecné rysy: uložené procedury, kurzory, triggers**).

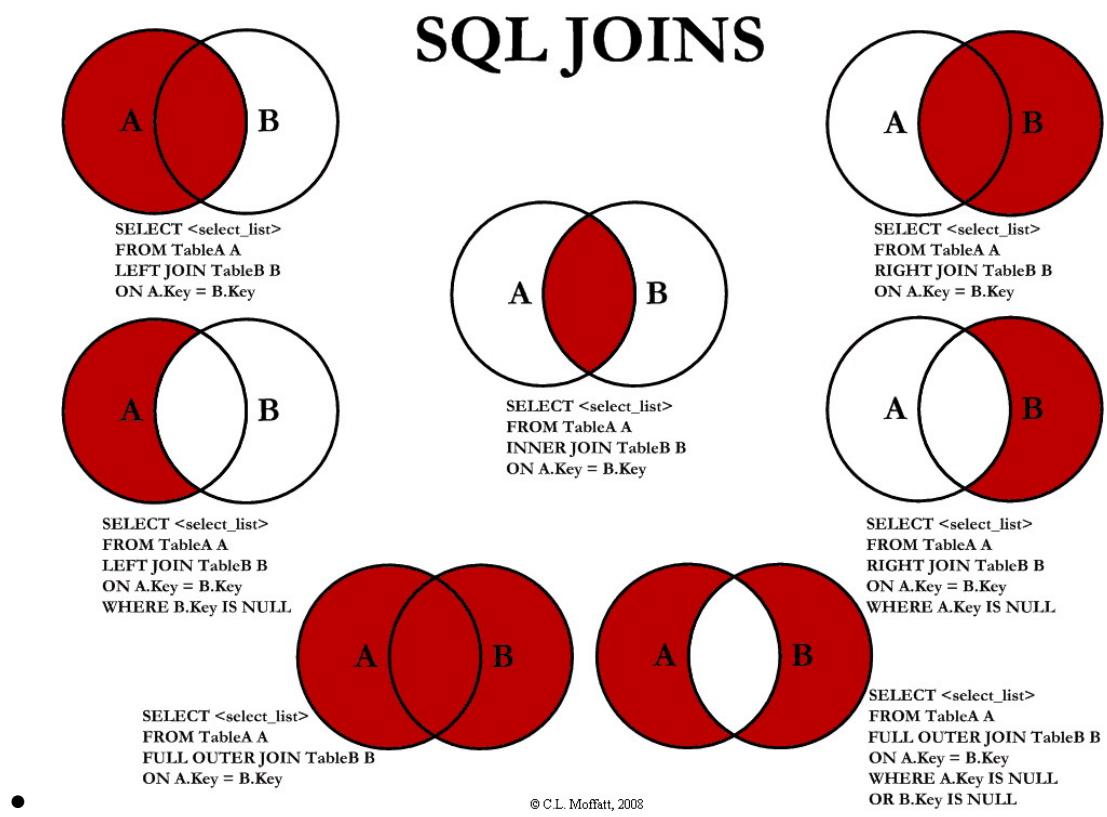
<http://lucie.zolta.cz/index.php/iformacni-systemy-databaze/46-sql>

- SQL (Strucrured Query Language) je *relační jazyk* založen na predikátovém kalkulu. Na rozdíl od jazyků založených na relační algebře, kde se dotaz zadává algoritmem, tyto jazyky se soustředí na to co se má hledat, ne jak, tzn. jedná se o deklarativní jazyk.
- SQL obsahuje příkazy pro:
- DDL - data definition language
 - vytváření a modifikaci relačního schématu (tabulek, databází) - CREATE, ALTER (MODIFY, ADD, DROP), DROP = vytvoř, uprav, smaž
- DML - Data Manipulation Language
 - modifikace dat - INSERT, UPDATE, DELETE = vlož, uprav, smaž
- DQL - Data Query Language
 - vyhledávání v relacích - SELECT, ORDER BY, GROUP BY, JOIN = vyhledej, seřaď, shlukuj, spoj
- DCL - Data Control Language
 - správa databázových systémů
 - transakce - COMMIT, ROLBACK = úspěšně provedená transakce, save-point uvnitř transakce ke kterému se dá vrátit byla-li transakce přerušena

SELECT

- získání dat z databáze
- struktura příkazu select
 - SELECT výběr sloupců nebo agregační funkce (count, sum, avg, ..)
 - FROM tabulky s daty + jejich případné spojení

- WHERE podmínka výběru řádků
 - GROUP BY seskupení záznamů aby se daly využít agregační funkce
 - HAVING podmínka agregačních funkcí
 - ORDER BY způsob setřízení řádků
- SQL dotaz se může skladat z více SELECTů, můžou být ve WHERE, FROM i SELECT
<https://www.itnetwork.cz/csharp/databaze/ms-sql-databaze-poddotazy-v-prikadech>
- Agregační funkce
 - statistické funkce, které umožňují nad seskupenými řádky spočítat aritmetické nebo statistické funkce
 - tj. pokud select obsahuje nějaký atribut je potřeba použít GROUP BY se všemi atributy selectu, Agregační fce fungují bez GROUP BY pouze pokud v select není žádný sloupec
 - AVG, SUM, COUNT, MIN, MAX
 - nabídka funkcí se může lišit podle databázového systému
 - v HAVING mohou být podmínky HAVING COUNT(*)>1
- lze spojit data z více tabulek (spojení podle primárních a cizích klíčů)
- podminka ve WHERE nebo klauzule JOIN ve FROM
https://www.w3schools.com/sql/sql_join.asp



procedurální rozšíření SQL: PL/SQL, T/SQL

<http://lucie.zolta.cz/index.php/iformacni-systemy-databaze/71-proceduralni-rozsireni-sql>

- PL/SQL je procedurálně rozšířené SQL. Kromě základních příkazu pro vytváření a modifikaci dat obsahuje PL/SQL triggery, funkce, procedury, kurzory. To umožňuje přenést část aplikační logiky přímo do databází.
- Procedury

- Program v databázi jehož příkazy jsou uložen v databázi.
- **Anonymní procedury** - nepojmenované procedury které nejdou volat. Mohou být uloženy v souboru nebo spuštěny přímo z konzole. Jsou pomalejší než pojmenované procedury, protože nemohou být předkompilovány.
- **Pojmenované procedury** - obsahují hlavičku se jménem a parametry. Díky tomu se dají volat z jiných procedur či triggrů nebo spuštěny příkazem EXECUTE. Jelikož jsou komplikovány jen jednou, jsou rychlejší než anonymní.
- procedura je spouštěna uživatelem a nemá návratovou hodnotu. (funkce je stejná akorát má navíc návratovou hodnotu)
- procedura

```
CREATE OR REPLACE PROCEDURE VlozStudenta2 (
    p_login IN Student.login%type,
    p_fname IN Student.fname%type,
    p_lname IN Student.lname%type,
    p_tallness IN Student.tallness%type
)
AS
BEGIN
    INSERT INTO Student(login, fname, lname, tallness)
    VALUES (p_login, p_fname, p_lname, p_tallness);
    tisk('Student s loginem ' || p_login || ' a jmenem ' || p_fname || ' ' || p_lname || ' byl uspesne vlozeny');
EXCEPTION
    WHEN OTHERS THEN
        tisk('Nastala chyba pri vkladani studenta ' || p_login);
END;
EXECUTE VlozStudenta2('pav955','Petr','Pavel',120);
```

- funkce má jinou hlavičku a obsahuje RETURN

```
CREATE OR REPLACE FUNCTION FVlozStudenta3 (
    p_login IN Student.login%type,
    p_fname IN Student.fname%type,
    p_lname IN Student.lname%type,
    p_tallness IN Student.tallness%type
)
RETURN VARCHAR -- datovy typ navratove hodnoty
AS
BEGIN
    INSERT INTO Student(login, fname, lname, tallness)
    VALUES (p_login, p_fname, p_lname, p_tallness);
    RETURN 'ok';
EXCEPTION
    WHEN OTHERS THEN
        RETURN 'error';
END;
```

- Triggery

- Trigger je v podstatě procedura spojená s tabulkou. Přesněji s operací nad tabulkou, protože se spouští ve chvíli kdy je na tabulkou zavolaný příkaz **INSERT, UPDATE, nebo DELETE** (může se ještě specifikovat podmínkou WHERE).
- **BEFORE, AFTER** - jsou příkazy triggeru definující zda se má trigger spustit před nebo po provedení aktivačního příkazu
- **ON nazev_tabulky**
- **FOR EACH ROW** - tělo triggeru se provede pro každý řádek tabulky které se týká
- **BEGIN tělo END**
- **OLD, NEW** - označuje staré či nové hodnoty.

```
CREATE OR REPLACE TRIGGER PocitatdloInsertu
AFTER insert on Student
BEGIN
    update Statistiky set pocet = pocet + 1 WHERE operace = 'insert';
END;
```

- Kurzory

- Kurzor je ukazatel na řádek víceřádkového výběru. Je třeba jej v programu deklarovat pokud budeme zpracovávat víceřádkové výběry. Kurzorem mohu pohybovat a tak se dostanu na další řádky výběru.
- **implicitní** - vytváří se automaticky po provedení příkazu INSERT, UPDATE, DELETE

- **explicitní** - ručně vytvořený kurzor. Vytváří se nejčastěji ve spojením s příkazem select
- Příkazy pro práci s kurzorem
- **OPEN** kurzor - otevře kurzor, tedy nastaví ho na první řádek
- **FETCH** kurzor **INTO** proměnná - příkaz pro pohyb kurzoru. Načte aktuální záznam do proměnné a posune se na další záznam. Nebo lze procházet v cyklu.
- **CLOSE** kurzor - uzavře kurzor.

```

CREATE OR REPLACE PROCEDURE VytvorZalozniTabulku(p_nazevtabulky IN VARCHAR)
AS
    p_prikaz varchar(1000);
    CURSOR tableCols IS SELECT * FROM USER_TAB_COLUMNS WHERE table_name = UPPER(p_nazevtabulky);
BEGIN
    p_prikaz:= 'CREATE TABLE ' || p_nazevtabulky || '_backup (';
    FOR tableCol IN tableCols LOOP
        p_prikaz := p_prikaz || tableCol.column_name || ' ' || tableCol.data_type || '(' || tableCol.data_length || ')' || ', ';
    END LOOP;
    p_prikaz := p_prikaz || ')';
    p_prikaz := REPLACE(p_prikaz, ', )', '');
    p_prikaz := p_prikaz || ')';
    EXECUTE IMMEDIATE p_prikaz; -- dynamické PL/SQL
EXCEPTION
    WHEN OTHERS THEN
        Tisk('chyba');
END;

```

- Statické a dynamické PL/SQL

- **Statické** - klasické procedury, které mají vázané proměnné a jsou předkompilované
- **Dynamické** - kód SQL příkazu je vytvářen dynamicky za běhu - vytvoření textového řetězce a jeho spuštění příkazem EXECUTE IMMEDIATE
- je lepší využívat statické je rychlejší, pro některé operace je nutné využívat dynamické

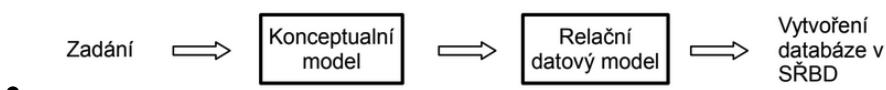
- **Analýza informačního systému (konceptuální a datový model, stavová analýza, funkční analýza – minispecifikace, návrh formulářů).**

<http://lucie.zolta.cz/index.php/iformalni-systemy-databaze/31-modelovani-databazovych-systemu>

SZZ (kajfosz.cz)

Modelování databázového modelu.

- Ve fázi analýzy se používá konceptuální model, který modeluje realitu na logickou úroveň databáze. Konceptuální model je výsledkem datová analýza a je nezávislý na konkrétní implementaci.
- V implementační fázi si pak pomáháme databázovými modely, kde modelujeme vazby a vztahy (realitu) na konkrétní tabulky (obecně SŘBD). Databázový model můžeme dále dělit na relační a síťový model. Fyzickým uložením dat na paměťové médium se zabývá interní model.



Konceptuální a datový model ds1 - 1

<http://lucie.zolta.cz/index.php/iformalni-systemy-databaze/30-konceptualni-model>

- Konceptuální model/Logický model - logický popis struktury databáze, který nezohledňuje použitý databázový systém

- řeší se jak by měla databáze vypadat
 - tabulky, atributy, vazby
 - Databázové schéma/Datový model (např. relační datový model) - popis databáze definované pro konkrétní databázový systém a pro konkrétní datový model
 - Data mohou být uspořádána (strukturována) mnoha různými způsoby: relační model, semistrukturovaný model (xml, json), graf, nestrukturovaná data
-
- ```

graph LR
 Z[Zadání] --> KM[Konceptuální model]
 KM --> RD[Relační datový model]
 KM --> S[Strom/les]
 KM --> G[Graf]
 RD --> RDS[Relační databázový systém]
 S --> XDS[XML databázový systém]
 G --> GDS[Grafový databázový systém]

```
- datový model řeší jak budou data uchovávána, definuje strukturu
  - Části relačního datového modelu
    - Struktura - definice relací a jejich obsahu
    - Integrita - obsah databáze musí splňovat podmínky
    - Operace - jak můžeme přistupovat a manipulovat s daty
  - Datová analýza zkoumá objekty reálného světa, jejich vlastnosti a vztahy. Výsledkem datové analýzy je konceptuální schéma. V rámci datové analýzy zpracujeme zadání (specifikaci požadavků na IS):
    - podstatná jména = identifikujeme objekty
    - slovesa = identifikujeme vazby mezi objekty
    - najdeme vlastnosti a stavy nalezených objektu = identifikujeme atributy
  - Z takto získaných informací sestavíme konceptuální model:
    - ER Diagram
  - Následně se vytvoří datový model který obsahuje navíc ještě:
    - Lineární zápis entit
    - Lineární zápis vztahů
    - Datového slovníku
      - název atributu, dat typ, délka, klíč, null, IO, popis
    - Popis IO (integritních omezení)
      - podmínka, která musí být splněna, aby entita byla validní
      - popis všech pravidel integritních omezení
  - ER diagram
    - Grafické znázornění objektů a vztahů mezi nimi.
    - Zobrazuje konceptuální i relační datový model
  - Datový slovník
    - podrobný rozpis jednotlivých atributů. Tabulka obsahuje typ atributů, velikost, integritní omezení
    - Integritní omezení obsahují další specifikaci atributů, které není dáno typem a délkou. Nejčastěji se týká formátu atributu. Např. login se skládá z třech čísel a třech písmen, nebo rodné číslo je složeno z data narození, apod.

- <https://dbedu.cs.vsb.cz/cs/SubjectsInYear/564/333>
  - Časová (dynamická, stavová) analýza: Identifikujte netriviální stavy entit a popište je ve stavové analýze (pokud se takové ve vašem projektu nacházejí). Například: Objednávka, může mít stavy: nová, přijatá, zaplacená, stornovaná, expedovaná, vrácena, dobropisovaná, reklamovaná, atd. Je zřejmé, že z nové objednávky se nemůže přímo stát objednávka expedovaná, dokud objednávka nebyla přijata,
  - Popisuje název stavů a jakých hodnot nabývají určité atributy
- Stav automobilu:**
- dostupný - #dostupnost = 1
  - nedostupný - #dostupnost = 0
  - připraven k rezervování - #dostupnost = 1 AND #rezervovano = 0
  - rezervovaný - #rezervovano = 1

## Funkční analýza

<http://lucie.zolta.cz/index.php/ifinformacni-systemy-databaze/32-funkcni-analyza>

- Popis funkcí, procedur, triggerů, složitých selectů
- Zatímco datová analýza se zabývá strukturou obsahové části systému (strukturou databáze), funkční analýza řeší funkce systému. Funkční analýza tedy vyhodnocuje manipulaci s daty v systému. Analyzuje základní funkce systému a aktéry, kteří se systémem pracují. Výstupem jsou pak minispecifikace - podrobné analýzy elementárních funkcí systému.
- Minispecifikace
  - Podrobná analýza algoritmů elementárních funkcí. Při tvorbě minispecifikace se používá přirozený jazyk, tak aby byla minispecifikace nezávislá na implementaci v konkrétním prostředí. Měla by však být strukturovaná a používat standardní programové struktury (větvení, cykly,...).
    - IF všechny výrobky v objednávce jsou rezervovány,  
THEN pošli objednávku k dalšímu zpracování oddělení prodeje.  
OTHERWISE,  
FOR EVERY nezarezervovaný výrobek v objednávce DO:  
    Zkus najít volný výrobek a rezervuj ho.  
    IF výrobek není na skladě,  
        THEN informuj správce.
  - Strukturovaným popisem (v bodech) vysvětlujeme algoritmus uložené procedury
  - Každý bod vede k jednomu příkazu v PL/SQL nebo T-SQL.
  - Jestliže s nějakým bodem souvisí standardní SQL příkaz (SELECT, INSERT, UPDATE, DELETE), musí být tento příkaz součástí popisu

návrh formulářů ds2 - 10, 11 ?? co k tomu říct ??

- uživatelské rozhraní pro uživatele
- lze vytvářet v ASP .NET
- Výpis dat z tabulky - možnost výběr sloupců, (po kliknutí detail)
- Z db načítat pouze data která budou zobrazena uživateli - minimalizace dat získávaných z db
- minimalizovat počet dotazů na db
- uživatel by měl mít možnost vložení nového záznamu, GUI prvky pro zadání dat
- k návrhu formulářů nevím ale obecně k návrhu GUI třeba toto

- princip prvořadosti uživatele
  - zachovávat zvyklosti uživatele ve vzhledu a uspořádání formulářů
- princip jednotnosti
  - jednotný styl a ovládání v celém systému
- princip vlídnosti
  - jemná upozornění na chyby ( ne "CHYBA!!!" ale "Zadej celé číslo")
- respektovat kontext ve zprávách směrem k uživateli
  - ne "ERROR" ale "Záporný příjem nelze evidovat")
- respektovat úroveň zkušeností uživatele
- minimalizovat čas pro získání informací
  - uživatel co nejméně kroků k dosažení požadovaného cíle
- úplnost a správnost
  - každý vstup zkонтrolovat
- maximalizovat spolehlivost komunikace
  - odlišit zprávy a data od uživatele a od systému
- poskytnout návod v každé situaci
- umožnit návrat
  - umožnit uživateli změnit svoji volbu
- optimalizovat množství výstupních informací
  - informovat o extrémním množství případně zobrazených informací
    - "vašemu dotazu vyhovuje 12 566 záznamů, chcete je zobrazit?
    - "vašemu dotazu nevyhovuje žádný záznam"

• **Transakce (definice, ACID, serializovatelnost transakcí, zotavení, řízení souběhu, úroveň izolace transakcí).**

- **Transakce** je základní nedělitelná jednotka zpracování dat, která musí proběhnout buď celá, nebo (v případě že je přerušena) obnovit původní stav databáze a spustit se znova. Pokud k tomu nedojde, je ohrožena konzistence databáze.
- Transakce je logická (nedělitelná, atomická) jednotka práce s databází, která začíná operací BEGIN TRANSACTION a končí provedením operací COMMIT nebo ROLLBACK.
  - COMMIT – úspěšné ukončení transakce. Programátor oznamuje transakčnímu manageru, že transakce byla úspěšně dokončena, databáze je nyní v korektním stavu, a všechny změny provedené v rámci transakce mohou být trvale uloženy v databázi. Jinými slovy všechny změny jsou potvrzeny (angl. committed) pro trvalé uložení v databázi.
  - ROLLBACK – neúspěšné provedení transakce. Programátor oznamuje transakčnímu manageru, že databáze může být v nekorektním stavu a všechny změny provedené v rámci transakce musí být zrušeny (angl. roll back nebo undo).
- Jedná se o skupinu příkazů, která musí převést databázi z jednoho konzistentního stavu do druhého (syntaxe Begin Commit Rollback)
- Transakce musí splňovat vlastnosti **ACID**: db2- 5
  - **A = Atomičnost** - transakce musí být atomická => proběhnou buď všecky operace

- transakce, nebo žádná.
- **C = Korektnost** - transakce převádí korektní stav databáze do jiného korektního stavu databáze, mezi začátkem a koncem transakce nemusí být databáze v korektním stavu.
- **I = Izolovatelnost** - transakce jsou navzájem izolovány: změny provedené jednou transakcí jsou pro ostatní transakce viditelné až po provedení COMMIT.
- **D = Trvalost** - jakmile je transakce potvrzena, změny v databázi se stávají trvalými i po případném pádu systému.
- př. transakce
 

Chceme převést 100Kč z účtu číslo 345 na účet číslo 789.

```
BEGIN TRANSACTION;

try {
 UPDATE Account 345 { balance -= 100; }
 UPDATE Account 789 { balance += 100; }
 COMMIT;
}
catch(SqlException) {
 ROLLBACK;
}
```

### Serializovatelnost transakcí DB2 - 7

- Pokud je plán transakcí serializovatelný, pak se neprojevují negativní vlivy souběhu
- potřebujeme zajistit aby se souběžně vykonávané transakce neovlivňovaly, aby nevznikaly problémy souběhu
- pokud jsou transakce prováděny za sebou mluvíme o sériovém plánu
- Plán vykonávání dvou transakcí je korektní tehdy a jen tehdy pokud je serializovatelný - Transakce pracující se stejnými daty musí být vykonávány postupně

Jelikož nad jednou databází většinou pracuje mnoho uživatelů, musí být databáze schopna řešit velké množství požadavků a transakcí. Zde může docházet k různým problémům vlivem špatné koordinace jednotlivých transakcí. Je třeba se zabývat **řízením souběhu** požadavků.

Hlavním problémem ke kterému dochází je, že si dvě transakce probíhají zároveň nad stejnými daty budou přepisovat data. Nejjednodušším řešením problému souběhu transakcí je spouštět transakce sériově, tedy jednu po druhé. Jenže to by bylo zdlouhavé a zbytečně by se během dlouhých přenosů dat nevyužíval procesor, který by mohl zpracovávat další transakce. Proto se transakce provádějí paralelně, ale tak aby byl splněn **požadavek na sériovost transakcí**, který říká aby výsledek paralelního zpracování byl stejný jako při zpracování sériovém.

### Řízení souběhu

<http://lucie.zolta.cz/index.php/iformalni-systemy-databaze/70-zamykani>

- řízení souběhu zabraňuje vzniku problémů souběhu
- problémy souběhu

- ztráta aktualizace

| Transakce A    | Čas                   | Transakce B    |
|----------------|-----------------------|----------------|
| READ <i>t</i>  | <i>t</i> <sub>1</sub> | -              |
| -              | <i>t</i> <sub>2</sub> | READ <i>t</i>  |
| WRITE <i>t</i> | <i>t</i> <sub>3</sub> | -              |
| -              | <i>t</i> <sub>4</sub> | WRITE <i>t</i> |

- problém nepotvrzené závislosti

| Transakce A | Čas   | Transakce B |
|-------------|-------|-------------|
| -           | $t_1$ | WRITE $t$   |
| READ $t$    | $t_2$ | -           |
| -           | $t_3$ | ROLLBACK    |

- problém nekonzistentní analýzy - A načítá více hodnot a B je během načítáním změní

| Transakce A   | Čas   | Transakce B        |
|---------------|-------|--------------------|
| READ $acc_1$  | $t_1$ | -                  |
| $acc_1 = 30$  |       |                    |
| $summa = 30$  |       |                    |
| READ $acc_2$  | $t_2$ | -                  |
| $acc_2 = 20$  |       |                    |
| $summa = 50$  |       |                    |
| -             | $t_3$ | READ $acc_3$       |
| -             | $t_4$ | WRITE $acc_3 = 60$ |
| -             | $t_5$ | READ $acc_1$       |
| -             | $t_6$ | WRITE $acc_1 = 20$ |
| -             | $t_7$ | COMMIT             |
| READ $acc_3$  | $t_8$ | -                  |
| $acc_3 = 50$  |       |                    |
| $summa = 110$ |       |                    |

- Konflikty čtení a zápisu
  - RR konflikt není problém
  - RW = neopakovatelné čtení

| Transakce A | Čas   | Transakce B |
|-------------|-------|-------------|
| READ $t$    | $t_1$ |             |
|             | $t_2$ | WRITE $t$   |

■ WR

| Transakce A | Čas   | Transakce B |
|-------------|-------|-------------|
| WRITE $t$   | $t_1$ |             |
|             | $t_2$ | READ $t$    |
| ROLLBACK ?  | $t_3$ |             |

■ WW

| Transakce A | Čas   | Transakce B |
|-------------|-------|-------------|
| WRITE $t$   | $t_1$ |             |
|             | $t_2$ | WRITE $t$   |

ROLLBACK ?  $t_3$

- Nejznámější techniky **Řízení souběhu** tj. techniky které garantují serializovatelnost transakcí
- zamykání - nejčastější řešení
  - pokud transakce A chce provést čtení nebo zápis nějakého objektu v databázi (nejčastěji tedy entice), pak požádá o zámek na tento objekt.
  - Význam zámků spočívá v tom, že žádná jiná paralelní transakce nemůže získat zámek a nemůže tedy provést čtení či aktualizaci (které by způsobilo konflikt) do té doby než A tento zámek uvolní.
  - Transakce si při práci s daty může pozamykat části databáze proti přepisu. Většinou si zamkne záznamy které zpracovává, ale lze zamknout celou tabulku či dokonce databázi.
  - Zámek pro sdílený přístup - zamkne prvek pouze proti přepisu a ostatní transakce se mu můžou číst.
  - Exkluzivní zámek - zamkne prvek jak pro přepis, tak pro čtení.
  - zámkы sice řeší problém vzájemného přepisu, ale způsobují další problém a tím

je **problém uváznutí**. K tomu dochází když si transakce vzájemně zamknou záznamy s kterými potřebují pracovat a pak čekají až jim ta druhá ten jejich záznam odemkne.

- zamkne vše na začátku - neefektivní
- detekce uváznutí a ukončení jedné z transakcí
- plánovač pomocí časových razitek
- správa verzí
- časová razítka

### Úroveň izolace transakcí DB2 - 8

- Izolovanost transakcí je vykoupena nižším výkonem při souběhu (nižší propustnost) s řbd proto umožňuje nastavit úroveň izolace, která zvýší propustnost, ale sníží míru izolace transakce
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
- Vyšší úroveň značí vyšší míru izolace, ale nižší propustnost.
- Naopak nižší úroveň značí nižší míru izolace, ale vyšší propustnost.
- Pokud zvolíme maximální úroveň, SERIALIZABLE, pak jsou transakce maximálně izolovány od vlivů ostatních souběžných transakcí.
- Naopak při nižší úroveň izolace mohou nastat různé problémy souběhu.
- 

| Úroveň izolace   | Špinavé čtení | Neopakovatelné čtení | Výskyt fantomů |
|------------------|---------------|----------------------|----------------|
| READ UNCOMMITTED | Ano           | Ano                  | Ano            |
| READ COMMITTED   | Ne            | Ano                  | Ano            |
| REPEATABLE READ  | Ne            | Ne                   | Ano            |
| SERIALIZABLE     | Ne            | Ne                   | Ne             |

- Špinavé čtení = nepotvrzená závislost
  - V případě úrovně RU může nastat tzv. špinavé čtení, tedy transakce může načíst data změněná a dosud nepotvrzená jinou transakcí

| Transakce A                                | Čas   | Transakce B                                      |
|--------------------------------------------|-------|--------------------------------------------------|
| -                                          | $t_2$ | UPDATE student SET rocnik=1<br>WHERE id='jan001' |
| SELECT * from student<br>WHERE id='jan001' | $t_3$ | -                                                |
| COMMIT                                     | $t_4$ |                                                  |
| ○                                          | $t_5$ | COMMIT/ROLLBACK                                  |

- Neopakovatelné čtení = RW
  - V případě úrovně RC (a nižší) je umožněno tzv. neopakovatelné čtení.
  - tomto případě příkaz SELECT požaduje sdílený zámek na záznam, SŘBD ale nedodrží dvoufázový zamykací protokol a zámky mohou být uvolněny před ukončením transakce.
  - Výlučné zámky jsou ovšem uvolněny až na konci transakce.

| Transakce A                                | Čas   | Transakce B                                                |
|--------------------------------------------|-------|------------------------------------------------------------|
| SELECT * from student<br>WHERE id='jan001' | $t_1$ | -                                                          |
| -                                          | $t_2$ | UPDATE student SET rocnik=1<br>WHERE id='jan001'<br>COMMIT |
| SELECT * from student<br>WHERE id='jan001' | $t_3$ | -                                                          |
| COMMIT                                     |       | $t_4$                                                      |

- Výskyt fantomů
  - Pokud je povolena úroveň RR (a nižší) a v transakci se vyskytují dva stejné dotazy, pak může nastat výjimka souběhu zvaná výskyt fantomů.
  - V tomto případě systém vrátí pro stejný dotaz v transakci A v čase  $t_1$  resp.  $t_4$  různé výsledky: SŘBD neprovádí zamykání rozsahu záznamů z tabulky Student

| Transakce A                                           | Čas   | Transakce B                                           |
|-------------------------------------------------------|-------|-------------------------------------------------------|
| SELECT * from student<br>WHERE rocnik BETWEEN 1 AND 2 | $t_1$ | -                                                     |
| -                                                     | $t_2$ | INSERT INTO student<br>VALUES('mar006',<br>'Marek',2) |
|                                                       | $t_3$ | COMMIT                                                |
| SELECT * from student<br>WHERE rocnik BETWEEN 1 AND 2 | $t_4$ | -                                                     |
| COMMIT                                                | $t_5$ | -                                                     |

## Zotavení

- Pod pojmem zotavení rozumíme zotavení databáze z nějaké chyby. Jde o obnovu databáze s použitím systémových logů (UNDO log, REDO log) či stínových tabulek.
- výsledkem zotavení musí být korektní stav databáze
- Ne všechny DBS zotavení (a transakce) podporují, často především z výkonnostních důvodů, nicméně většina aplikací se bez podpory transakcí neobejde.
- K dosažení korektního stavu často využíváme nějaké redundantní informace, která je pro uživatele (za normálních okolností) skryta a využívají se pro rekonstrukci databáze v nekorektním stavu
- Základním problémem vzniklým při systémové chybě, je ztráta obsahu hlavní paměti, tedy ztráta obsahu vyrovnávací paměti (log nicméně aktualizace obsahuje).
- Během zotavení se po restartu systému provádí pro jednotlivé transakce tyto operace:
  - Přesný stav transakce přerušené chybou není znám a transakce musí být zrušena (angl. undo). Pro tuto akci budeme často používat označení UNDO.
  - Pokud byla transakce úspěšně ukončena (příkazem COMMIT), změny ovšem nebyly přeneseny z logu do databáze. V tomto případě musí být transakce přepracována (angl. redo). Pro tuto akci budeme často používat označení REDO.
- techniky zotavení
  - odloženou aktualizací (NO-UNDO/REDO)
    - neprovádí aktualizaci logu a databáze až do potvrzení transakce
    - po potvrzení transakce se zapíše do logů a potom do db
    - pokud transakce selže nemusí se provádět UNDO
    - REDO se provádí pokud se zapsalo do logu ale už ne do db
    - hrozí přetečení paměti při dlouhé transakci
  - okamžitou aktualizací (UNDO/NO-REDO)

- provádí aktualizace logu a databáze po každé aktualizaci transakce
  - Aktualizace jsou zapsány do logu a poté je aktualizována databáze
  - Pokud transakce selže před dosažením potvrzovacího bodu, pak je nutné provést UNDO - v logu je vše co se musí zrušit
  - nízký výkon - častý zápis do db ale nehrází přetečení paměti
- kombinovaná technika (UNDO/REDO)
  - Aktualizace jsou zapisovány do logu po COMMIT.
  - K aktualizaci databáze dochází v určitých časových intervalech - kontrolních bodech (check points).
  - při zotavení se vytvoří 2 seznamy UNDO a REDO
  - Do UNDO vlož všechny transakce, které nebyly úspěšně dokončeny před posledním kontrolním bodem.
  - průchod logu od posledního konkrétního bodu - Pokud je pro transakci T nalezen v logu záznam COMMIT, přesuň T ze seznamu UNDO do seznamu REDO.
  - Systém prochází log zpětně a ruší aktualizace transakcí ze seznamu UNDO.
  - Systém prochází logem dopředu a přepracovává aktualizace transakcí ze seznamu REDO.
- při chybě datového média se databáze obnovuje ze záložní kopie

- **Vykonávání dotazů v databázových systémech (fyzický návrh databáze, vykonávání dotazů, logické a fyzické operace).**

Fyzický návrh databáze je třetí fází návrhu databáze. V podstatě je tato fáze založena na převodu logického návrhu (tabulek, sloupců a integritních omezení) na fyzický návrh, který je použitelný v rámci zvoleného databázového systému. Jelikož jednotlivé databázové systémy mají svá specifika a omezení, může existovat více postupů v rámci fyzického návrhu databáze, reflektující daná specifika jednotlivých databázových systémů.

<https://docplayer.cz/1527446-12-blok-fyzicky-navrh-database.html>

Fyzická implementace definuje datové struktury pro základní logické objekty:

- tabulky,
- indexy,
- materializované pohledy,
- rozdělení dat (data partitioning).

Fyzická implementace tedy řeší uložení dat na nejnižší úrovni databáze.

Dostupných datových struktur je celá řada, implicitně nabízí SŘBD administrátorovi nějakou volbu.

Řeší se jak bude databáze na databázovém systému implementovaná aby byla co nejvýkonější.

Existuje celá řada typů tabulek a indexů vhodných pro různé operace.

Ve funkční analýze se zjišťují i často používané sql operace a podle nich se ladí implementace - neexistuje optimální univerzální fyzický návrh.

využívá se indexy, stránkování, ...

## implementace

- Tabulka typu halda (Heap table) – záznamy v tabulce nejsou uspořádány. Je to implicitní volba CREATE TABLE.
  - Složitost operací:
  - Není možné se spoléhat na uspořádání záznamů v tabulce ( $\Rightarrow$  neefektivní vyhledávání,  $O(n)$ ).
  - Tento typ tabulky je velmi efektivní z pohledu operace INSERT ( $O(1)$ ) a využití místa.
  - Před vykonáním operací DELETE a UPDATE musí DBS často provést vyhledání záznamu/záznamů.
- Shlukovaná tabulka (Clustered table) – záznamy jsou v datovém souboru seřazeny podle zvoleného klíče. Při každém vložení se seřazuje
  - efektivnější vyhledávání na hodnotu primárního klíče.
  - Zhoršený výkon zejména operace INSERT (data se musí 'zatřizovat').
  - Nebo může být realizováno např. B stromem nebo hashovací tabulkou

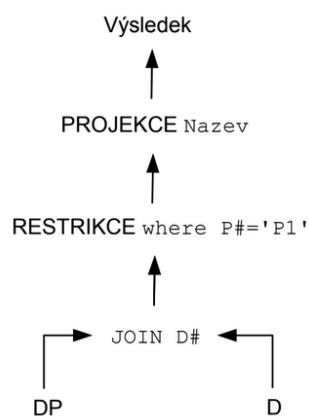
## Vykonávání dotazů ds2 12

### Výběr plánu

- Komponenta databázového systému zvaná query optimizer (query processor) vybere nejlepší plán (tedy plán dle kterého bude dotaz vykonán nejrychleji).
- Čas vykonávání dotazu můžeme zlepšit např. těmito technikami: parametrizované dotazy, hromadné operace, nastavení transakcí.
- Na úrovni databáze můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu případně dobu vykonávání operací  $\Rightarrow$  fyzický návrh databáze.
- Plán vykonání dotazu velmi ovlivňuje výkon a paměťovou náročnost - (dříve spojení nebo selekce dle podmínky? - selekce lepší)
- některé optimalizace řeší optimalizátor.
- Nicméně některé optimalizace (indexy, uložení) musí řešit vývojář.

Identifikujeme 3 fáze generování plánů dotazu:

- Převod dotazu do interní formy
  - Převod původního dotazu do zvolené interní formy
  - Eliminujeme syntaxi jazyka dotazu
  - Interní forma je nejčastěji nějaký druh dotazovacího stromu



- Převod do kanonické formy
  - V této fázi optimalizátor provádí celou řadu optimalizací.
  - Relační algebra umožňuje dotaz vyjádřit mnoha způsoby

- Převodem do kanonické formy tedy dochází k odstranění různých povrchních rozdílů a především nalezení efektivnějšího tvaru než nabízel původní dotaz.
  - Optimalizátor se snaží aplikovat různá transformační pravidla, která převádí výraz na výraz ekvivalentní
  - Dotaz tedy není ve skutečnosti vykonán přesně tak jak jej zadá uživatel!
- Vygenerování plánů dotazu a výběr nejlevnějšího plánu
  - V této fázi optimalizátor vytváří množinu dotazovacích plánů.
  - Každému plánu je přiřazena cena (I/O nebo CPU Cost), nejčastěji se jedná o sumu cen operací.
  - Cena procedury je závislá na aktuální mohutnosti vstupních relací, na mohutnosti mezivýsledků jednotlivých operací (odhad) a dalších statistik
  - Z množiny dotazovacích plánů pak optimalizátor vybírá ten nejlepší, tedy nejlevnější.
- Logické operace: DB2 - 12 od str.29
  - Selekcí ( odpovídá výběru řádků z tabulky na základě kritéria - podmínky)
  - Projekce ( odpovídá výběru sloupců tabulky)
  - Join (spojení)
  - Sort (třídění)
- Fyzické operace:
  - TABLE ACCESS (FULL) – sekvenční hledání v tabulce
    - Sekvenční průchod celou tabulkou – systém načte všechny stránky tabulky s maximální snahou o sekvenční čtení z disku a paměti.
    - nízká efektivita při vyhledávání dle podmínky
  - INDEX (UNIQUE SCAN a RANGE SCAN) – hledání v indexu
    - Unique scan - vhodné při vyhledávání jednoho klíče v indexu
    - Range scan - vhodné při vyhledávání rozsahu klíčů v indexu
    - index se vyplatí vytvářet při opakovém vyhledávání v tabulce - vede k zrychlení
    - problém pokud se přidávají záznamy musí se znova vytvořit index - časově náročné
    - lze vytvořit složený index obsahující více atributů
    - zrychluje logické operace pokud vyhledávají parametr, který je v indexu

## Index

- Index se vytváří obvykle pro klíče a cizí klíče.
- Index je vytvořen pokud:
  - bude používán k nalezení malého počtu záznamů v tabulce,
  - pokryje jeden nebo více dotazů.
- Atributy, které se často vyskytují v klauzuli where, jsou potenciálními kandidáty na index.
- Každý index znamená zvýšení počtu operací při změnách v databázi.
- Vytvoření nového indexu tedy musíme vždy pečlivě zvažovat.

## • Návrh a implementace datové vrstvy (objektově-relační mapování, DTO, DAO, efektivní implementace datové vrstvy).

### ORM

objektově relační mapování namapuje data z databáze do objektové struktury.

Tyto techniky se starají o konverzi mezi relační databází a objekty, se kterými se pracuje v objektově orientovaném jazyce. Díky těmto technikám se programátor k jazyku SQL vůbec nedostane, tabulky v databázi vidíme jako kolekce objektů, se kterými můžeme pracovat běžnými prostředky programovacího jazyka.

Jsme úplně odstíněni od toho, že pracujeme s relační databází. Použití ORM usnadňuje zejména provádění běžných databázových operací označovaných jako CRUD operace. Mezi tyto operace se řadí čtení (Read), zápis (Create), úprava (Update) a mazání dat (Delete). ORM nakonec provede operace jako SQL dotazy, ale uživatel se o to nemusí starat.

existují návrhové vzory, jakými lze ORM realizovat: Table Data Gateway, Row Data Gateway, Active Record, Data Mapper

Objektově relační mapování (ORM) je programovací technika zpřístupňující relační (či objektově-relační data) pro objektové prostředí.

- práce s objektovým modelem,
- rychlejší vytváření aplikací vs menší výkon aplikace
- přenositelnost mezi různými SŘBD vs. využívání specifických vlastností SŘBD (datové typy, funkce, bezpečnost atd.)
- Některé ORM (Hibernate, LINQ to SQL, Entity Framework) se navíc často pokouší vytvořit vrstvu mezi aplikací a databází, tak aby aplikace byla nezávislá na databázi.
- Existují nástroje třetích stran (- rychlá tvorba aplikace, snadná změna sřbd, oproti vlastní implementaci nižší výkon a kontrola nad sql) nebo lze vytvořit vlastní implementaci ( - plná kontrola nad sql příkazy ale problémy se změnou sřbd a čas na implementaci)

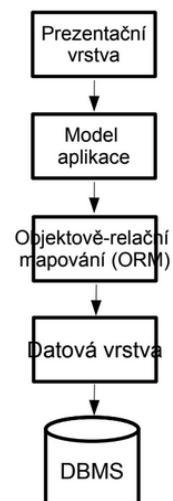
## DTO

- data transfer object
- objekt udržující data v aplikaci
- jedna instance reprezentuje jeden řádek databáze
- objekt do něhož se budou vkládat data z db? - třída do níž se mapují data z db
- obsahuje data, případně aplikační logiku (tj. metody které neřeší komunikaci s db)

```
public class User
{
 public int Id { get; set; }
 public String Login{ get; set; }
 public String Name{ get; set; }
 public String Surname{ get; set; }
 public String Address{ get; set; }
 public String Telephone{ get; set; }
 public int MaximumUnfinishedAuctions{ get; set; }
 public DateTime? LastVisit{ get; set; }
 public String Type{ get; set; }
}
```

## DAO

- Data access object
- jedna třída reprezentuje tabulku a její operace
- obsahuje logiku pro komunikaci s db, data získává jako instanci DTO a využívá je v CRUD metodách
- můžu si vybrat jakým způsobem implementuju DAO ve své aplikaci - výběr z návrhových vzorů pro datové zdroje (Table Data Gateway, Row Data Dateway, Active Record, Data Mapper)



DTO i DAO je součástí ORM

### Efektivní implementace datové vrstvy CO JE DATOVÁ VRSTVA pojed na internetu není

- Mezi námi a databází se nachází 2 vrstvy a to datová vrstva, která přistupuje k datům uloženým v nějakém trvalém úložišti, jako je například relační databáze a business vrstva, která určuje, jak lze data vytvářet, ukládat a měnit.
- Datová vrstva informačního systému odděluje aplikaci od databáze. Jde o třídy a funkce zajišťující komunikaci s databází.
- Úkol datové vrstvy je zajistit maximální výkon přístupu k datům
- Návrhové vzory na této vrstvě
  - Data Access Object / Data Mapper
  - Active Record (není zrovna typickým zástupcem vhodným pro třívrstvou arch.)
  - Row Data Gateway - instance = jeden řádek tabulky; obsahuje vyhledávací třídu, která nám vytváří instance. Může implementovat Table Data Gateway
  - Table Data Gateway - instance = tabulka; možno implementovat ve vyhledávací třída RDGateway
- Díky nákladné migraci relačních dat, vznikl objektově-relační datový model
- Existuje mnoho ORM řešení třetích stran pro přenos dat mezi db a aplikací
  - Hibernate, LINQ, Entity Framework
- 

DOPSAT (dnes 29.3.2022 nevím co napsat nemůžu najít žádné dobré info)

**Příklad otázky:** Jaký SQL dotaz může v transakci vracet neočekávané výsledky, pokud použijeme úroveň izolace Read Committed? Popište i relaci se kterou pracujete a napište konkrétní dotaz nad touto relací.

## **Architektury počítačů a operační systémy (APPS, OSY)**

- **Architektury počítačů (základní architektury počítačů, jejich popis, výhody a nevýhody, princip fungování počítače, způsoby adresování, hierarchické uspořádání pamětí).**

Architektura neurčuje jednoznačné definice, schémata nebo principy. Hovoří o tom že počítač se skládá z menších částí a teprve jejich vzájemné propojení vytvoří funkční celek.

Architekturu můžeme chápat ze 4 pohledů:

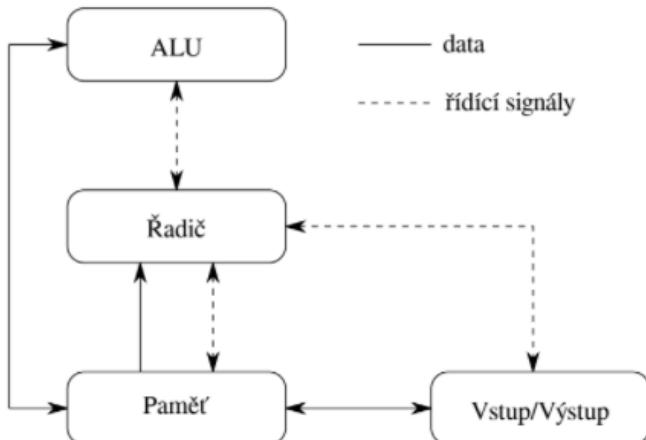
- struktura - popis jednotlivých částí a jejich propojení
- součinost - popis řízení dynamické komunikace mezi funkčními bloky
- realizace - popis vnitřní struktury bloků
- funkcionalita - výsledné chování počítače jako celku

Základní architektury (koncepce)

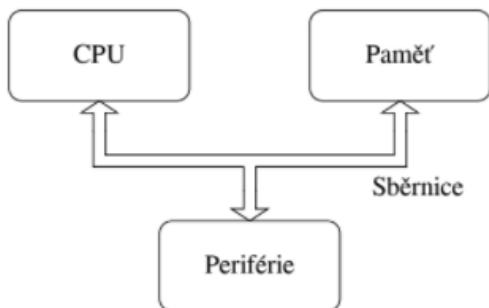
### **Von Neumannova**

určil kritéria a principy které musí počítač splňovat: (**principy fungování počítače**)

1. Počítač se skládá z paměti, řídící jednotky, aritmetické jednotky, vstupní a výstupní jednotky.
2. Struktura počítače je nezávislá na typu řešené úlohy, počítač se programuje obsahem paměti.
3. Následující krok počítače je závislý na kroku předchozím.
4. Instrukce a operandy (data) jsou v téže paměti. (rozdíl oproti Harvardské)
5. Paměť je rozdělena do buněk stejné velikosti, jejich pořadová čísla se využívají jako adresy
6. Program je tvořen posloupností instrukcí, ty se vykonávají jednotlivě v pořadí v jakém jsou zapsány v paměti.
7. Změna pořadí prováděných instrukcí se provede instrukcí podmíněného či nepodmíněného skoku.
8. Pro reprezentaci instrukcí čísel, adres a znaků, se používá dvojková číselná soustava.



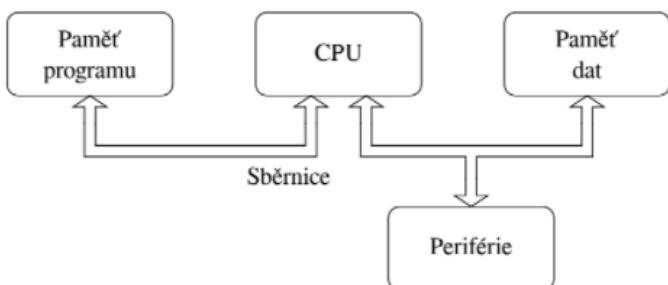
Obrázek 1: Základní schéma počítače podle von Neumanna



Obrázek 2: Počítač podle von Neumanna

### Harvardská architektura

Několik let po von Neumannovi, přišel tým odborníků z Harvardské univerzity s vlastní koncepcí počítače, která se od Neumannovy příliš nelišila, ale odstraňovala některé její nedostatky. V podstatě jde pouze o oddělení paměti pro data a program.



Obrázek 3: Harvardská architektura počítače

Nedostatek obou koncepcí je sekvenční vykonávání instrukcí, které sice umožňuje snadnou implementaci systému, ale nepovoluje paralelní zpracování. Paralelismus se musí simulovat až na úrovni operačního systému. Úzké místo systému je také ve sběrnicích, které nedovolují přistupovat současně do více míst paměti a současně umožňují přenášet data pouze jedním směrem.

## Porovnání vlastností

### von Neumann

- Výhody
  - rozdělení paměti pro kód a data určuje programátor
  - řídící jednotka procesoru přistupuje do paměti pro data i pro instrukce jednotným způsobem
  - jedna sběrnice - jednodušší výroba
- Nevýhody
  - společné uložení dat a kódu může mít při chybě za následek přepsání programu
  - jedna sběrnice tvoří úzké místo

### harvardská koncepce

- Výhody
  - oddělení paměti dat a programu přináší výhody - program nemůže přepsat sám sebe, paměti mohou být vyrobeny odlišnými technologiemi, každá paměť může mít jinou velikost nejmenší adresovací jednotky, dvě sběrnice umožňují jednoduchý paralelismus lze načítat instrukce a data současně
- Nevýhody
  - 2 sběrnice kladou vyšší nároky na řídící jednotku procesoru a vyšší náklady na výrobu
  - nevyužitou část paměti dat nelze použít pro program a obráceně

## adresování paměti

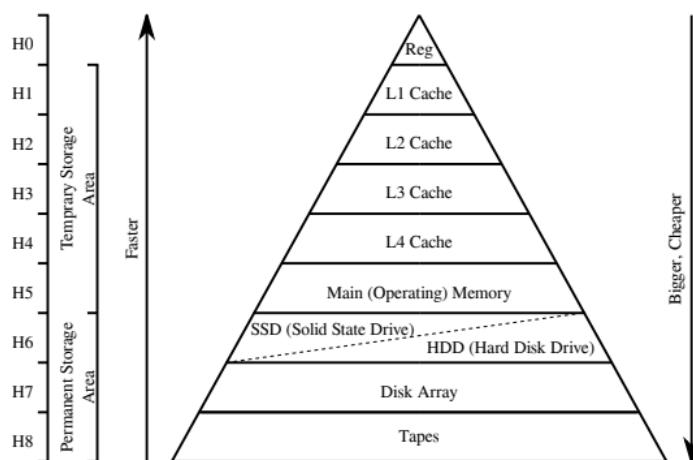
Paměť se skládá z paměťových buněk kde každá buňka nese logickou hodnotu 0/1. Paměťové buňky jsou umístěny ve čtvercové matici v jedné nebo více vrstvách. Výběr buňky musí být proveden ve 2 krocích pomocí řádkového a sloupcového dekodéru. Rozdělení adresování na 2 dekodéry přináší výhodu při adresování, protože z každého adresového dekodéru stačí polovina adresních vodičů (oproti tomu pokud by byl jen jeden).

Lze zredukovat počet adresních vodičů tak, že nejdříve bude dodána adresa řádku a na stejných vodičích následně i adresa sloupce. Pro výběr dekodéru, kterému je adresa určena musí být navíc přidány dva řídící signály RAS a CAS.

Adresy paměti jsou čteny jedna za druhou - multiplexing, stačí poloviční počet adresních vodičů. Pokud paměťový kontroler pošle adresu řádku aktivuje RAS signál a čip DRAM ví že dodaná adresa je adresa řádku. Kontrolér DRAM aktivuje adresový buffer k získání adresy a přesune ji do dekodéru řádku, který ji dekóduje. Poté paměťový kontroler pošle adresu sloupce spolu se signálem CAS a kontrolér DRAM pozná že se jedná o sloupec, a znova aktivuje adresový buffer. Adresový buffer přijme a přesune ji do dekodéru sloupce.

Takto se adresuje buňka a kontrolér DRAM určuje signály WE a READ jestli bude probíhá čtení nebo zápis.

## hierarchické uspořádání paměti v pc



Obrázek 12: Paměťová hierarchie

Každá technologie má své vlastnosti a výrobní cenu za jednotu kapacity - proto se využívá velké množství typů pamětí uspořádaných do více vrstev - Paměťová hierarchie

Čím je paměť rychlejší tím je dražší a má menší kapacitu a vesměs se jedná o dočasné paměti (energeticky závislé). Paměti pro trvalé ukládání dat jsou pomalejší ale mají větší kapacitu a jsou levnější

H0 - registry procesoru. Rychlosť odpovídá rychlosći CPU a pohybuje se v jednotkách ns, nebo desetinách ns. Kapacita se může pohybovat ve stovkách bytů na jádro, takže celkově i v jednotkách kB.

H1 - Cache L1, SRAM, rychlosť odpovídá vnitřním sběrnicím CPU, obvykle v nižších jednotkách ns. Kapacita desítky až stovky kB na jádro.

H2 - Cache L2, SRAM, rychlosť obvykle od jednotek do 10 ns. Kapacita stovky kB na jádro.

H3 - Cache L3, SRAM, rychlosť v nižších desítkách ns, kapacita v jednotkách až desítek MB.

H4 - Cache L4, SRAM, rychlosť v desítkách ns, kapacita desítka až stovky MB.

H5 - Hlavní paměť, někdy nazývaná operační paměť, DRAM, rychlosť v desítkách ns, kapacita jednotky až stovky GB,

H6 - SSD disky, Flash, rychlosť v desetinách či jednotkách ms, kapacita stovky GB až jednotky TB. Pevné disky, rychlosť v jednotkách až desítek ms, kapacita v jednotkách TB.

H7 - Disková pole, parametry odvozeny od pevných disků, kapacita od desítek TB až po jednotky PB.

H8 - Páskové jednotky, rychlosť dle použitého řešení od desítek minut až po hodiny.

Paměti můžeme rozdělit podle:

- Podle typu přístupu:

- RAM - random access memory
  - SAM - serial access memory
- Podle R/W:
  - RWM - read write memory
  - ROM - read only memory
  - WOM - write only memory
- Podle typu buněk:
  - DRAM - dynamic RAM
    - všechny buňky = kondenzátory. Ten se samovolně vybíjí a je potřeba ho neustále nabíjet (jinak po ~10ms se vybije)
    - informace je tedy uložena ve formě náboje (log 1 / log 0)
    - Buňky jsou uspořádány v matici
    - Prve se čte řádek, pak sloupec.
    - Varianty
      - FP(Fast Page)DRAM, kdy se při dalším čtení už neptá na řádek.
      - S(synchroní)DRAM, která využívá burst mode - zadá se ROW a COL a pak to čte přilehlé
  - SRAM - static RAM
    - informace uložena stavem klopného obvodu
    - Méně paměti, ale rychlejší -> cache paměti, zatímco DRAM pro klasické RAMky
    - paměťově buňky uspořádány do matice
  - PROM, EPROM, EEPROM, FLASH - programovatelné paměti.
    - DRAM a SRAM nedrží svůj obsah po odpojení napájení -> nejsou vhodné pro startovací proces PC -> protom ROM paměti
    - hlavní náplň těchto pamětí tedy je pamatovat si data i bez napájení
    - Existují nějaké pojistky, když jsou v původním stavu, tak vedou proud (log 0), když je (programátor) přepálí, tak proud nevedou (log 1)
    - Tím pádem po naprogramování do nich (ROM, PROM) už není možný další zápis.
    - EEPROM jsou pak paměti, které jdou přeprogramovat, informace se ukládá pomocí elektrického náboje, který je kvalitně izolován. Je jí možné vymazat UV zářením.
    - EEPROM pak jdou vymazat elektrickým impulzem. (Electrically Erasable Programmable ROM)
    - V dnešní době FLASH paměti, což jsou vylepšené EEPROMky, princip stejný.

• **RISC procesory (základní konstrukční vlastnosti, způsoby urychlování práce procesorů, zřetězené zpracování instrukcí, predikce skoků).**

dnes jsou 2 typy procesorů

CISC - počítač se složitým souborem instrukcí

RISC - počítač s redukovaným souborem instrukcí

Dnes se vyskytuje kombinace obou typů neexistuje žádný čistý risc nebo cisc procesor

Důvod vzniku RISC - ukázalo se že programátoři využívají instrukce velmi nerovnoměrně.  
Počet nejfrequentovanějších instrukcí je omezen na velmi úzkou část instrukčního souboru.  
Složité instrukce jsou využívány velmi málo. Nalezl se tedy optimální soubor instrukcí a vznikly procesory RISC.

### Vlastnosti architektury RISC

Nejtypičtější vlastností je malý instrukční soubor, ale je mnoho dalších vylepšení. Procesor vykonává jen ty nejnutnější instrukce, složitější komplikátor převádí na více jednoduchých instrukcí.

- v každém strojovém cyklu by měla být dokončena jedna instrukce (vykonání instrukce ale netrvá jeden strojový cyklus!)
- mikroprogramový řadič může být nahrazen rychlejším obvodovým řadičem
- používá zřetězené zpracování instrukcí
- celkový počet instrukcí a způsobů adresování je malý
- data jsou z hlavní paměti vybírána a následně ukládána výhradně jen pomocí instrukcí LOAD a STORE
- instrukce mají pevnou délku a jednotný formát (zjednoduší dekódování instrukcí)
- je použit vyšší počet registrů
- složitost se z technického vybavení přesouvá částečně do optimalizujícího komplikátoru

Risc přináší výhody pro uživatele i výrobce. Zkracuje se vývoj procesoru, snazší výroba, zřetězené zpracování instrukcí

nevýhody - nárůst délky programů tvořených omezeným počtem instrukcí (zpomalení se neprojevilo díky zřetězenému zpracování instrukcí)

### Zřetězené zpracování instrukcí

Vykonání instrukce se skládá z několika kroků (počet kroků se liší podle typu procesoru). U CISC musela instrukce projít všemi těmito kroky než se začala vykonávat další. U RISC se po provedení jednoho kroku zpracování hned začíná zpracovávat další instrukce a postupně se předává do dalších kroků zpracování. - princip výrobní linky.

Aby to fungovalo efektivně je potřeba aby každý krok zpracování pracoval stejně rychle a aby mezi jednotlivými kroky byla mezipaměť (registr) - toto předávání vede k jistému zpoždění ale celkový přínos je mnohem větší. Pro vykonávání několika instrukcí pak stačí méně strojových cyklů než u CISC

| Krok | Význam                        |
|------|-------------------------------|
| 1.   | <b>VI</b> Výběr Instrukce     |
| 2.   | <b>DE</b> Dekódování          |
| 3.   | <b>VA</b> Výpočet Adresy      |
| 4.   | <b>VO</b> Výběr Operandu      |
| 5.   | <b>PI</b> Provedení Instrukce |
| 6.   | <b>UV</b> Uložení Výsledku    |

|    | T <sub>1</sub> | T <sub>2</sub> | T <sub>3</sub> | T <sub>4</sub> | T <sub>5</sub> | T <sub>6</sub> | T <sub>7</sub> | T <sub>8</sub> | T <sub>9</sub> | T <sub>10</sub> | T <sub>11</sub> | T <sub>12</sub> | T <sub>13</sub> |
|----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| VI | I <sub>1</sub> |                |                |                |                |                | I <sub>2</sub> |                |                |                 |                 |                 | ...             |
| DE |                | I <sub>1</sub> |                |                |                |                |                | I <sub>2</sub> |                |                 |                 |                 |                 |
| VA |                |                | I <sub>1</sub> |                |                |                |                |                | I <sub>2</sub> |                 |                 |                 |                 |
| VO |                |                |                | I <sub>1</sub> |                |                |                |                |                | I <sub>2</sub>  |                 |                 |                 |
| PI |                |                |                |                | I <sub>1</sub> |                |                |                |                |                 | I <sub>2</sub>  |                 |                 |
| UV |                |                |                |                |                | I <sub>1</sub> |                |                |                |                 |                 | I <sub>2</sub>  |                 |

Tabulka 4: Postup provádění instrukcí procesorem CISC

|    | T <sub>1</sub> | T <sub>2</sub> | T <sub>3</sub> | T <sub>4</sub> | T <sub>5</sub> | T <sub>6</sub> | T <sub>7</sub> | T <sub>8</sub> | T <sub>9</sub> | T <sub>10</sub> | T <sub>11</sub> | T <sub>12</sub> |
|----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|
| VI | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub> | I <sub>6</sub> | I <sub>7</sub> | ...            |                |                 |                 |                 |
| DE |                | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub> | I <sub>6</sub> | I <sub>7</sub> |                |                 |                 |                 |
| VA |                |                | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub> | I <sub>6</sub> | I <sub>7</sub> |                 |                 |                 |
| VO |                |                |                | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub> | I <sub>6</sub> | I <sub>7</sub>  |                 |                 |
| PI |                |                |                |                | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub> | I <sub>6</sub>  | I <sub>7</sub>  |                 |
| UV |                |                |                |                |                | I <sub>1</sub> | I <sub>2</sub> | I <sub>3</sub> | I <sub>4</sub> | I <sub>5</sub>  | I <sub>6</sub>  | I <sub>7</sub>  |

Tabulka 5: Zřetězené provádění instrukcí procesorem RISC

## Problémy zřetězeného zpracování

Datové a strukturální hazardy - datové některá rozpracovaná instrukce potřebuje mít k dispozici data předchozí instrukce, která ještě není dokončena. strukturální - omezené prostředky procesoru chceme využívat využívaný prostředek

řešením je vložení prázdných (čekacích) instrukcí ?

Problémy plnění fronty instrukcí (problémy se skoky)

Pro optimální činnost zřetězeného zpracování je důležitá reakce na skokové instrukce.

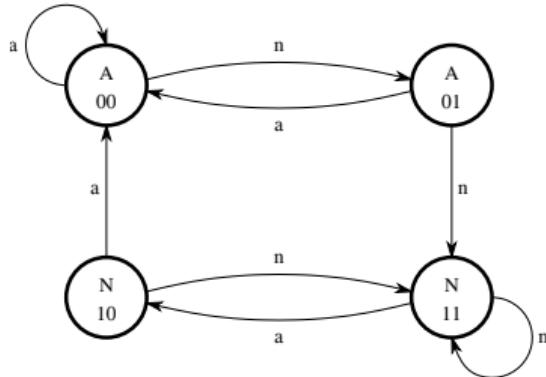
Nepodmíněné skoky nejsou problém. Problémy jsou s podmíněnými skoky kdy se rozhoduje které instrukce budou vykonány až za běhu programu. Je potřeba rozhodnout kterou "větev" instrukcí načítat ke zpracování než bude vyhodnocena podmínka. Pokud byly načítány správné instrukce, tak se použijou pokud ne tak se zahodí a fronta instrukcí se začne plnit novými instrukcemi. I tak to je rychlejší než čekat na vyhodnocení podmíny.

## Metody zlepšení plnění fronty instrukcí

Bit predikce skoku - Tato metoda že ve formátu instrukce se vyhradí jeden bit predikující zda se krok provede či nikoliv. Predikce může být statická nebo dynamická. Dynamická si při každém provedeném podmíněném skoku zaznamená jestli se skok provedl nebo ne. Dynamická metoda je výhodnější, ale vyžaduje nutnost bit predikce zapisovat což může být technicky komplikované. Jednobitová predikce je jednoduchá zlepšuje rozhodování hlavně při provádění

cyklu, ale způsobí že dojde k jednomu selhání vždy na začátku cyklu a k jednomu na konci. (na začátku je nastaveno že skočí, ale neskočí a na konci že neskočí ale skočí, při prvním neskočení si nastaví predikci že neskočí, při konci skočí a nastaví si predikci že skočí - proto při dalším cyklu udělá na začátku zase chybu)

Lepší chování nabízí dvoubitová predikce, ta sníží selhání u cyklů pouze na konci cyklu. Jedná se o 4 stavový automat s 2 stavům ano a ne, které predikují jestli dojde ke skoku nebo ne



Obrázek 3: Dvoubitová predikce jako stavový automat

Stav se změní pokud se skok 2krát po sobě provede nebo neprovede (u cyklu tedy na začátku ne zůstává ne, a na konci ano ale zůstane nastaveno na ne)

#### Zpoždění skokové instrukce

Frontu instrukcí je možno vyplnit instrukcemi než dojde k vykonání podmínky. Vyplnění prázdnými čekacími instrukcemi je neefektivní. Pokud by se podařilo před skokovou instrukcí najít několik instrukcí nesouvisejících přímo s podmíněným skokem mohly by se zařadit jako instrukce před dokončením podmínky.

#### Paměť skoků

jednobitová nebo dvoubitová predikce ale bity nejsou ukládány ve formátu instrukce ale v tabulce provedených podmíněných skoků která je realizována jako součást procesoru. Do tabulky se ukládají adresy posledních provedených skoků a k nim jednobitová nebo dvoubitová predikce

#### Zdvojená fronta zpracování

procesor má více front zpracování v každé z nich zpracovává jiné instrukce (došlo / nedošlo ke skoku). Až se vyhodnotí podmínka jedna z front se použije a druhá zahodí

- GPU, CUDA (důvody využívání GPU, popis technologie CUDA, postup výpočtu, základní pravidla programování, postup výpočtu, práce s pamětí).

GPU umožňuje masivní paralelismus - vykonávání mnoha instrukcí současně  
gpu nejsou univerzální, ale jsou specializované na masivní paralelismus

je třeba přenášet data do gpu a zpět do pc - sběrnice tvoří úzké místo  
GPU jsou vhodné pro výpočty vykreslování obsahu - paralelní výpočet barev pro všechny pixely?

nvidia 1993 1995 první grafický čip 2000 sjednocení s firmou 3dfx

2006/2007 - uvedení technologie cuda

snaha vyvinout graf. kartu tak, aby mohla řešit univerzální problémy, grafické početní, ..

první univerzální 2010 architektura fermi - umožňovala jakékoli gpu computing

CUDA je technologie firmy Nvidia.

- jedná se o hardwarovou a softwarovou architekturu, která umožňuje na GPU spouštět programy napsané v jazycích C/C++
- Díky technologie nemusí programátor znát technické detaily o grafické kartě, stačí mu znalost programování technologie CUDA a program funguje na všech GPU podporující tuhutu technologii.

cuda jádra(malé procesory) jsou umístěny v multiprocesorech čím výkonnější gpu tím více multiprocesorů. Multiprocessor se skládá z několika cuda jader. Různé typy gpu mají různý počet multiprocesorů. Lepší gpu jich mají více než horší stejné generace. Multiprocesory jsou spojeny pouze pomocí paměti, tudíž o sobě neví a jsou na sobě nezávislé.

Cuda jádra na jednom multiprocesoru jsou rozděleny na části tzv. warp. Každý warp má svůj dekodér instrukcí, tudíž je očekáváno, že ve všech cuda jádrech ve warpu budou současně vykonávány stejné instrukce. Jinak se sníží výkon warpu protože dekodér není dimenzován na dekódování více instrukcí současně.

pro gpu je vhodný delší sekvenční program bez smyček a podmíněných skoků gpu neumožňuje vykonávání instrukcí mimo pořadí (neumí optimalizaci vykonávání) je vhodné mít sekvenčně seřazená data (nemá rád bin stromy atd.) většina gpu čipu je určena pro výpočet - nemá moc pomocných obvodů, cache atd.



definují se funkce které vytváří mřížku a spouštějí kernel. Dále se definují kernel funkce které provádějí příslušnou operaci. Většinou pracují s daty které jsou pole a pracují s konkrétní hodnotou v poli podle identifikátoru vlákna. Pro každý prvek pole by mělo být vytvořeno vlákno. Je potřeba nastavit vhodnou velikost bloku a mřížky. na začátku kernelu testovat že id je v rozsahu jinak konec.

```
int x = blockDim.x * blockIdx.x + threadIdx.x; // sloupec
int y = blockDim.y * blockIdx.y + threadIdx.y; // radek
```

hlavičky funkcí

`__global__` - přeloženo pro gpu a hlavička v počítači

`__device__` - pouze pro gpu

`__host__` - pouze pro počítač

### [CUDA – Wikipedie \(wikipedia.org\)](#)

#### Postup výpočtu

Vytváří se program (kernel), který bude paralelně spuštěn na více jádrech. Vlákna jsou organizována do bloků. Vlákna ve stejném bloku mohou sdílet data a lze synchronizovat jejich běh. Počet vláken v bloku je omezen (max 4 x 4?)

Bloky jsou organizovány do mřížky. Blok lze v rámci mřížky identifikovat unikátním indexem, který je přístupný ve spuštěném kernelu. Každý blok vláken musí být schopen pracovat nezávisle na ostatních, aby byla umožněna škálovatelnost systému. (na gpu s více jádry běží současně více bloků)

typický průběh výpočtu

1. Vyhrazení paměti na GPU
2. Přesun dat z hlavní paměti RAM do paměti grafického akcelerátoru
3. Spuštění výpočtu na grafické kartě
4. Přesun výsledků z paměti grafické karty do hlavní RAM paměti

#### Práce s pamětí

Na grafické kartě je 6 druhů paměti, se kterými může programátor pracovat

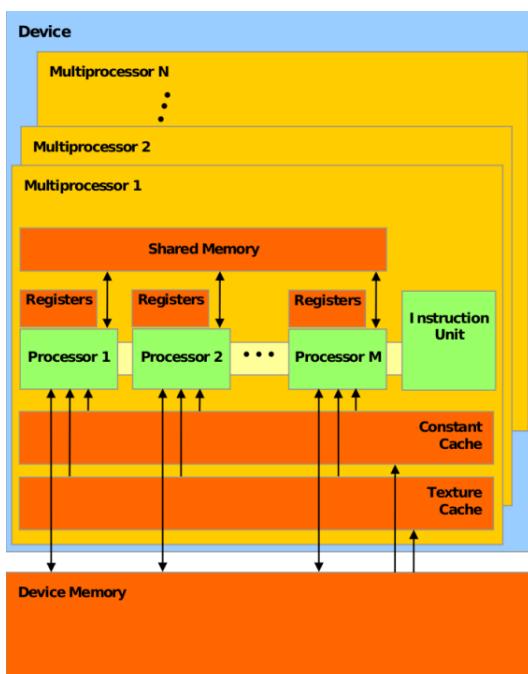
Pole registrů - je na stream procesorech, každé vlákno může přistupovat pouze ke svým registrům

Lokální paměť - paměť vlákna využívána když dojdou registry

Sdílená paměť - paměť na čipu gpu, mohou k ní přistupovat všechna vlákna v daném bloku

Globální paměť - sdílená mezi všemi streaming multiprocesory

Paměť konstant - globální paměť na čipu mikroprocesoru L1 cache (rychlá)



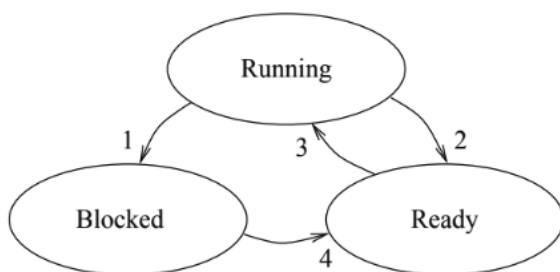
k cuda neměl poli vypracovaný dokument

- Procesy (vytváření procesů, implementace procesů v OS, tabulka procesů, stavy procesů, obsluha HW přerušení, multitasking, pseudoparalelismus, plánovací algoritmy).

Všechny moderní počítače musí provádět věci současně, což je realizováno přepínáním procesů. = pseudoparalelismus

Programy jsou organizovány jako řada sekvenčních procesů. Program jsou instrukce, které se mají vykonat. Proces je vykonání těchto instrukcí spolu se vstupními daty, s hodnotami instrukčního čítače, registrů a proměnných. Proces může vytvářet další procesy jako kopii sebe sama. V operačním systému je 1 proces init a z něj se vytvářejí další procesy - tvoří stromovou strukturu.

Každý proces má svůj adresní prostor. (proces nesdílí data)



Obrázek 2: Stavy procesu a přechody mezi nimi

- Running - bežící - právě využívá CPU,
- Ready - připravený - dočasně pozastavený během jiného programu,
- Blocked - zablokovaný - nemůže běžet, dokud nepřijde nějaká externí událost.

přechody

1. když proces zjistí že nemůže pokračovat (v některých systémech volání block)
2. plánovač rozhodl že proces běžel dlouho a chce jej přepnout
3. plánovač vybral proces, který spustí na cpu
4. nastane externí událost na kterou proces čekal

Procesy jsou v os ukládány do tabulky procesů

- záznam o každém procesu
- info - alokace paměti, status otevřených souborů, ..
- obsahuje vektor přerušení - adresy podprogramů obsluhy jednotlivých zařízení
- při přepnutí procesu se do tabulky musí uložit obsah registrů a ukazatel na zásobník (podprogram v assembleru)

Plánovač obsahuje i správu přerušení a meziprocesní komunikaci  
obsluha HW přerušení

1. HW uloží na zásobník čítač instrukcí
2. HW vezme nový čítač instrukcí z tabulky přerušení
3. podprogram v assembleru uloží registry
4. dále nastaví nový zásobník
5. spustí obsluhu přerušení v jazyce c

6. Plánovač označí čekající (blocked) proces jako připravený (ready).
7. Plánovač rozhodne, který proces bude následně spuštěn.
8. Obsluha v C se vrátí zpět do assembleru.
9. Podprogram v assembleru znovu spustí nový aktuální proces.

Operační systém musí podporovat multitasking tj. provádět několik procesů současně (rozdělovat mezi ně procesorový čas). Procesů je ale více než jader CPU (kdysi bylo 1 jádro). Proto implementuje tabulku procesů kde jsou procesy ukládány a střídá vykonávané procesy. Za určitý čas je spuštěno několik procesů ale souběžně běží pouze jeden. = pseudoparalelismus

(Procesy mohou vytvářet vlákna - sdílí adresní prostor procesu. Realizovány na straně OS nebo v uživatelském prostoru. Jsou vhodné při souběžném zpracovávání kdy potřebujeme sdílet paměť - náchylnost na souběh)

#### Plánování procesů

Když je připraveno více procesů ke spuštění, operační systém musí rozhodnout, který spustí jako první. Ta část operačního systému, která toto rozhodování provádí, se nazývá plánovač (scheduler) a použitý algoritmus rozhodování je plánovací algoritmus (scheduling algorithm). V dávkových systémech bylo jednoduché spustila se dávka a po dokončení se spustila další. V systémech sdíleného času to je složitější.

#### Kritéria plánovacího algoritmu

- férovost - rovnocenné šance na sdílení CPU
- efektivnost - držet procesor trvale využitý
- doba odezvy - minimalizace odezvy pro interaktivní uživatele (dostatečně časté střídání procesů)
- doba běhu - minimalizace času, který musí uživatel čekat na vykonání
- průchodnost - maximalizovat počet procesů vykonaných během časové jednotky

Některá kritéria si odporují - je potřeba najít nevhodnější kompromis pro určený typ využití

Počítač má elektronické hodiny, které pravidelně vyvolávají přerušení, které spustí plánovač a rozhodne jestli dojde k přepnutí procesu.

Preemptivní plánování - proces může být kdykoli pozastaven

Nepreemptivní - proces nesmí být pozastaven, spusť a dokonči

Preemptivní plánování přináší problémy se souběhem ale je jedinou možností jak mít interaktivní systém.

### Algoritmy plánování

#### Round robin

- každý proces má časový interval (kvantum), po který smí běžet
- po skončení kvanta je přepnuto na další proces, pokud skončí proces dříve nebo se zablokuje dojde také k přepnutí
- procesy jsou seřazeny postupně a hledá se ready proces a ten se spustí
- je potřeba nastavit vhodné kvantum - moc nízké vysoké nároky na režii a časté přepínání, moc vysoké ztráta interaktivity systému, ideálně 10ms
- priorita se dá řešit změnou velikosti kvanta, některé procesy mohou běžet déle než jiné

#### Prioritní plánování

- každý proces má nastavenou prioritu

- spouštěn je spustitelný proces s nejvyšší prioritou
- aby se předešlo tomu že proces s nejvyšší prioritou běží trvale plánovač každou časovou periodou sniže priority (když dosáhne nejnižší priority, tak se opět bude postupně zvyšovat na původní priority)
- lze seřadit do skupin podle priority a využívat round robin

Více front

- fronty podle priority
- po vyčerpání kvanta se přesune do fronty s nižší prioritou

Nejkratší dávka první

- je vhodné spustit nejdříve procesy, které běží nejkratší dobu, za kratší dobu bude dokončeno více procesů - nižší průměrná doba odezvy
- problém je seřadit dávky podle času, lze např. na základě předchozího chování, měření

Zaručené plánování

- máme n uživatelů a každý má zaručen  $1/n$  času
- je potřeba sledovat kdo kolik spotřeboval a podle toho přepínat

Plánování losováním

- Náhodný výběr procesu který bude spuštěn
- každý proces má počet losů a plánovač vždy vybere jeden los a vlastník je spuštěn
- proces který jich má více má větší šanci na spuštění
- výhody - při přidání procesu má proces šanci být spuštěn hned při nejbližším přepnutí, spolupracující procesy si mohou předávat losy

Real-Time plánování

- vhodné pro periodické události
- proces má limit do kdy musí být dokončen a podle toho je vybrán proces
- režie přepínání je v praxi příliš vysoká - speciální rt os

Dvouúrovňové plánování

- pokud se zaplní hlavní paměť některé procesy se začnou ukládat na disk - velké zpomalení
- toto plánování sníží zpomalení
- rozdělí na 2 skupiny na disku a hlavní paměti a nad nimi využívá nějaký standardní plánovací algoritmus
- střídají se procesy které jsou dlouho na disku s těmi co jsou dlouho v paměti

- **Meziprocesní komunikace (souběh, kritická sekce, vzájemné vyloučení, základní způsoby realizace vzájemného vyloučení, semafory, monitor, fronta zpráv, sdílená paměť).**

## Souběh

V operačních systémech mohou společně pracující procesy sdílet některé běžné paměti, kde všichni mohou číst i zapisovat. Sdílená paměť může být hlavní paměť, nebo sdílený soubor. Problém 2 nebo více procesů současně čtou nebo modifikují sdílená data.

## Kritická sekce

- je místo v kódu, kde může dojít k souběhu
- potřebujeme zabránit vzniku souběhu
- požadavky
  - Žádné dva procesy nesmí být současně uvnitř stejné kritické sekce.
  - Na řešení nesmí mít vliv počet a rychlosť CPU.
  - Žádny proces mimo kritickou sekci nesmí blokovat jiný proces.
  - Žádny proces nesmí zůstat čekat nekonečně dlouho na kritickou sekci.

### **Možnosti vzájemné vyloučení**

#### Zákaz přerušení

- nejjednodušší řešení, zákaz přerušení procesy, které jsou v kritické sekci
- povolení přerušení po opuštění kritické sekce
- zákaz přerušení zakáže i časovač pro přepínání procesů - nedojde k přepnutí procesu
- není vhodné pro uživatelské procesy, protože pokud nedojde z důvodu chyby k povolení přerušení přestane OS fungovat poběží pořád jeden proces
- využito v rámci jádra OS (nedělitelné instrukce)

#### Zamykací proměnné

- sdílená proměnná s hodnotou 0/1
- do kritické sekce lze vstoupit pokud je hodnota 0 a nastaví se na 1
- problém hodnota není uzamknuta proces může vstoupit ale než stihne nastavit hodnotu na 1 může být přepnuto a vstoupí druhý proces, 2 instrukce test a nastavení
- není vhodné řešení neřeší kritickou sekci pouze přesouvá problém

#### TSL instrukce

- test and set lock
- stejný princip jako zamykací proměnné, ale využívá hardwarovou instrukci pro testování a nastavení proměnné - nedělitelná instrukce
- instrukce TSL, pokud proměnná 0 vstoupí do kritické sekce a nastaví na 1, pokud není čeká dokud nebude 0
- problém je aktivní čekání dokud nebude moct vstoupit do kritické sekce

#### Uspání a probuzení

- když nelze vstoupit do kritické sekce uspí proces (Sleep) a po uvolnění je probuzen (Wakeup)
- zabrání aktivnímu čekání
- problém může být ztráta signálu, např. ve výrobce spotřebitel spotřebitel se chce uspat ale je přepnut proces, přepne se na výrobce ten chce probudit spotřebitele a uspí se, spotřebitel se uspí - ztráta signálu a uspané oba proces

#### Semafora

- uchovávají hodnotu, která identifikuje počet nezpracovaných přerušení, může nabývat hodnot 0 a větší
- operace down sníží hodnotu semaforu, pokud již je hodnota 0 uspí proces(vlákno) dokud nebude  $>= 1$ , potom sníží hodnotu a vstoupí do kritické sekce
- operace up zvýší hodnotu v semaforu
- up a down jsou nedělitelné instrukce
- lze jimi zamýkat kritickou sekci - na začátku down na konci up (binární semafor pouze hodnoty 0 a 1) nebo i řešit složitější IPC problémy - výrobce a spotřebitel, večeřící filozofové, spící holič

#### Monitory

- stejná funkčnost jako semafora, akorát se jedná o součást programovacího jazyka -

konstrukt programovacího jazyka pro označení kritické sekce (C# lock)

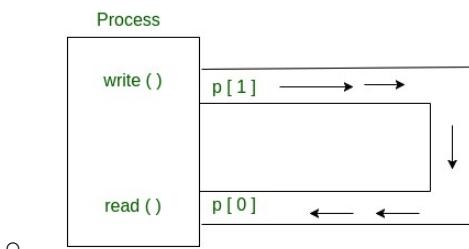
- u semaforů vede nevhodné použití k zablokování programu
- u monitorů řeší implementaci (použití semaforů) překladač - snížení rizika výskytu chyb

#### Předávání zpráv

- operace Send a Receive, které jsou implementovány jako systémová volání
- umožňuje odesílání zpráv i po síti
- send zašle zprávu do cíle
- receive vyzvedne zpráv, pokud není k dispozici zastaví se a čeká dokud něco nepřijde
- problém s nespolehlivostí zpráv(po síti) - ne vždy je doručeno - číslování zpráv, odesílání potvrzení doručení
- (lze takto řešit i výrobce spotřebitel - na začátku n prázdných zpráv od spotřebitele, výrobce jakmile obdrží zprávu odesílá data, výrobce jakmile obdrží zpracuje a odesílá prázdnou)

#### Sdílená paměť ??

- sdílení dat mezi procesy, meziprocesní komunikace
- shared memory (sdílená paměť)
  - systémové volání využívající semafory
  - může do ní přistupovat více procesů
  - narození od rour a front zpráv může takto komunikovat více procesů
- roury
  - má 2 konce zápis a čtení
  - meziprocesní komunikace, jedna strana posílá data druhá čte
  - jednosměrná komunikace
- - roura je "virtuální soubor" uložený v paměti
  - pokud proces začne číst dříve než je v rouře něco uloženo je uspán dokud tam něco není
- fronty zpráv
  - propojený seznam zpráv
  - procesy si otevřou stejnou frontu zpráv
  - proces mají systémová volání pro přidání zprávy do fronty a pro přečtení z fronty
  - zprávy ve frontě mají specifikovaný typ
- sockety
- signály
- sdílené soubory



- **Správa paměti (princip fungování virtuální pamětí, segmentace a její přínosy, správa/evidence volné paměti).**

Paměť je důležitý systémový prostředek, je jí omezené množství a je jí potřeba spravovat

Správa paměti bitmapami

- paměť rozdělena na alokační jednotky - části o stejné velikosti
- ke každé alokační jednotce přísluší 1 bit v bitmapě - 0 jednotka je volná, 1 obsazená
- čím menší je alokační jednotka tím větší je bitmapa - velikost bitmapy závisí na velikosti alokační jednotky
- jednoduchý způsob jak sledovat slova v paměti
- problém je náročné vyhledávání volných částí paměti - musí se procházet bitmapa a hledat dostatečně dlouhou sekvenci 0

Správa paměti propojenými seznamy

- uchovává se propojený seznam alokovaných a volných paměťových segmentů, kde segment je proces nebo mezera
- každý záznam specifikuje mezeru (díru) nebo proces a obsahuje počáteční adresu, délku a ukazatel na další záznam
- nelze mít více děr vedle sebe musí dojít k jejich spojení
- algoritmy pro alokaci paměti
  - první vhodný - první dostatečně velká díra
  - další vhodný - první dostatečně velká díra + pamatuje kde skončil a pokračuje tam
  - nejlepší vhodný - nejmenší dostatečně velká díra
  - nejhůře vyhovující - největší díra - zabrání vzniku velmi malých nepoužitelných děr
  - lze rozdělit na 2 seznamy - děr a procesů - rychlejší alokace pomalejší uvolnění
  - rychlý a vhodný - má seznamy děr podle velikostí, velmi rychlá alokace a pomalé uvolnění

Virtuální paměť

- nastal problém, že programy se staly natolik velkými, že se nemohly vejít do žádné dostupné díry (paměti)
- možné řešení bylo rozdělení programu na menší části (moduly), které se načítaly z disku po dokončení předchozího - náročné pro programátora, podobný princip začal řešit počítač > virtuální paměť
- vlastnosti virtuální paměti
  - celková velikost programů dat a zásobníků může překročit velikost dostupné paměti
  - O.S. uchovává v paměti jen ty části, které jsou využívány, zbytek je na disku
  - umožňuje delinearizaci paměti
  - umožňuje spustit program i když je v paměti jen částečně
- k tomuto využívá stránkování

Stránkování

- většina systémů virtuální paměti využívá techniku stránkování
- programy mají k dispozici virtuální adresový prostor, který může být větší než fyzický adresový prostor
- při použití virtuální paměti nejde adresa přímo na paměťovou sběrnici, ale do jednotky správy paměti (MMU), kde se mapují virtuální adresy na fyzické

- příchozí adresa se rozdělí na 2 části - číslo stránky + offset, číslo stránky se použije jako ukazatel do tabulky stránek a převede se na číslo rámce, který odpovídá stránce
  - následně se spojí s offsetem
  - získá fyzickou adresu v paměti, která se pošle na paměťovou sběrnici
- virtuální adresní prostor se dělí na jednotky - stránky
- odpovídající jednotky ve fyzické paměti se nazývají rámce stránek - rámce a stránky mají stejnou velikost
- Virtuální stránka musí mít informaci o přítomnosti nebo nepřítomnosti mapování stránky ve fyzické paměti (tj. jestli má rámec v paměti)
- pokud se pokusí přistoupit na stránku, která není namapována mmu oznámí výpadek stránky. O.S. vezme některý rámec, zapíše jej na disk a do uvolněného místa zapíše stránku
- tabulky stránek obsahují - číslo stránky, adresu rámce, byty R referenced, M modified, P protection, A absent (je načteno ano/ne), C caching
- tabulky stránek můžou být u větších tabulek velmi velké a potřebujeme aby byly rychlé
  - víceúrovňové tabulky stránek
    - není potřeba držet v paměti jednu velkou tabulkou
    - stačí držet v paměti primární ve které jsou informace kterou sekundární použít pro danou adresu (ta se dělí na více částí podle počtu úrovní + offset) popis pro 2 úrovně:
      - část adresy PT1 v primární tabulce vybere sekundární tabulkou
      - PT2 vybere v sekundární tabulce záznam stránky a získá adresu rámce
  - převrácené adresy stránek
    - informace pouze o tom co je načteno ve fyzické paměti
    - menší tabulka stránek ale pomalejší vyhledávání musí se projít vše
  - TBL - překlad s nahlédnutím do bufferu
    - často se přistupuje pouze k malému počtu záznamů v tabulce stránek
    - TBL je rychlý buffer, který obsahuje x posledních přístupů do paměti
    - nejdříve než nahlíží do tabulky stránek podívá se do rychlého TBL jestli tam adresa není.

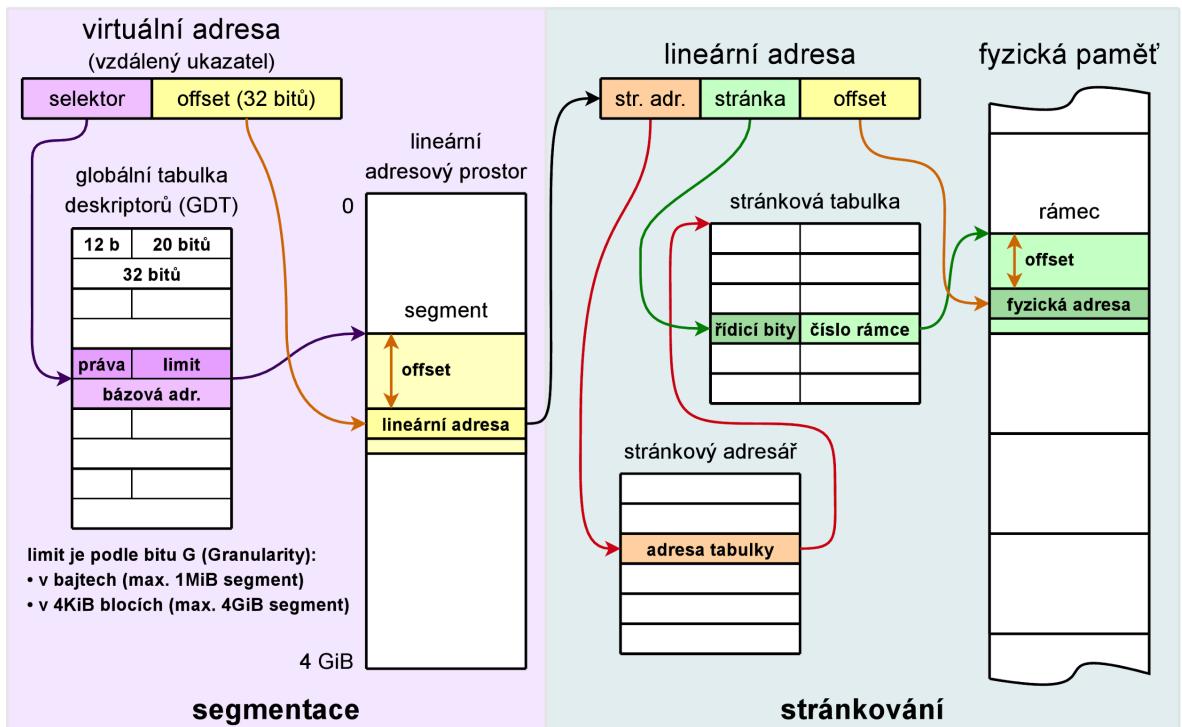
#### Algoritmy náhrady stránek

- když se objeví chyba stránky O.S. musí vybrat stránku, kterou vyjmeme z paměti
- je vhodné nahradit některou, která není často používána, aby se tam nemusela brzy znova vracet
- pokud byla původní stránka změněna musí se uložit na disk jinak není potřeba
- algoritmy:
- optimální algoritmus náhrady stránek
  - náhrada stránek podle počtu vykonaných instrukcí než budou znova potřeba
  - počet instrukcí nelze zjistit pouze logický koncept
- Náhrada dříve nepoužité stránky
  - u každé stránky se eviduje jestli k ní bylo přistupováno a byla modifikována
  - 4 kategorie
    - neodkazováno, nemodifikováno
    - neodkazováno, modifikováno - může vzniknout z kat3 nulováním R bitu
    - odkazováno, nemodifikováno
    - odkazováno, modifikováno

- je nejlepší nahradit stránku která má nejnižší kategorii
  - např. pokud není žádná stránka v kategorii neodkazováno, nemodifikováno, tak hledá jestli je nějaká v kategorii neodkazováno, modifikováno atd.
- FIFO
  - udržuje v paměti seznam stránek kde na začátku je nejstarší stránka na konci nejnovější
  - odstraní se vždy první
  - není ideální, nezohledňuje jestli je stránka využívána
- Náhrada stránky s druhou šancí
  - FIFO + kontrola R (referenced) bitu
  - pokud 1 bylo přistupováno dostane druhou šanci - reset R bitu a přesun na konec fronty
- Hodinový algoritmus náhrady stránky
  - stejně akorát se využívá kruhový seznam a ukazatel na prvek se inkrementuje není potřeba přesouvat prvky jako ve frontě
- Náhrada nejdéle nepoužívané stránky (LRU)
  - udržuje se seznam stránek seřazený podle toho kdy byla naposledy používána
  - odstraní se nejdéle nepoužívaná
  - je výpočetně náročné tento seznam udržovat - musí být aktualizováno při každém paměťovém odkazu
- Programová simulace LRU
  - sw řešení LRU
  - každá stránka čítač použití který se při přístupu inkrementuje - nezohledňuje že kdysi bylo používáno často a dlouho už ne - pořád vysoká hodnota čítače
  - zaznamenání historie do bitů čísla, v každém cyklu přidán bit 0/1 a bitový posun čísla o 1, mám historii přístupů do paměti

## Segmentace

- Může nastat problém, že proces potřebuje velké množství paměti a nevleze se do přiděleného prostoru, problém v posloupnosti adres - nemůže si vzít paměť jinde, adresový prostor musí být spojitý
- Proto vznikla segmentace - umožňuje mít více adresních prostorů (pro každý program 1)
- každý segment má lineární adresy 0 - maximum
- segmenty můžou mít rozdílnou velikost, která se může měnit během vykonání, kdy se může nezávisle na ostatních segmentech zvětšovat a zmenšovat, protože v adresním prostoru není nic co by tomu bránilo (zůstane spojitý)
- umožňuje oddělit adresní prostor pro data a kód, oddělení pro sdílení a ochranu
- segment je logická entita



- ze segmentu se překládá na lineární adresu, zde může být použito stránkování a lineární adresa je tedy virtuální a překládá se na fyzickou adresu
- adresní prostor segmentu je spojitý, ale převedené lineární adresy spojité být nemusí, může se skládat z více různě rozmištěných stránek

**Příklad otázky:** Vysvětlete princip fungování virtuální paměti a k čemu slouží segmentace. Co tyto technologie zlepšují či přináší proti základní koncepci fungování počítače dle von Neumanna?

## Počítačové sítě a komunikační technologie (POS, PB)

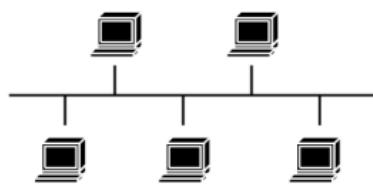
- Směrování a přepínání v počítačových sítích (topologie LAN sítí; aktivní prvky a jejich funkce – přepínač a jeho funkce, směrovač; směrovací protokoly – základní popis, třídy směrovacích protokolů, hierarchické směrování).

### LAN

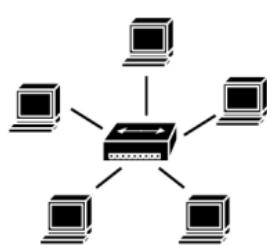
- local area network
- síť která pokrývá malé geografické území - domácnost, firmu, ..
- přenosové rychlosti jednotky Gb/s

#### topologie LAN

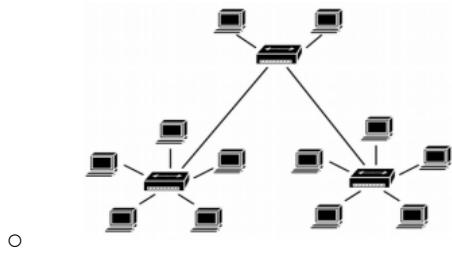
- sběrnice
  - pasivní médium
  - malé zpoždění signálu
  - signál je přijímán všemi připojenými zařízeními - limitovaná propustnost
  - slabá ochrana před selháním média



- hvězda
  - všechna zařízení jsou připojena k centrálnímu prvku (aktivní - switch, pasivní - hub)
  - odolné proti výpadku stanice, ale selhání centrálního prvku je kritické

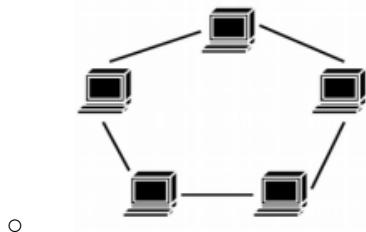


- strom
  - rozšířená hvězda
  - dnes nejpoužívanější
  - hierarchická struktura, musí tvořit stromovou strukturu, pokud existují redundantní spojení (tvořící cyklus) jsou zablokovány a topologie se pořád chová jako strom



- Kruh

- peer to peer spojení
- Data obíhají kolem kruhu a přes posuvné registry v každém uzlu
- Selhání kteréhokoliv uzlu znamená selhání sítě
- Paket je předán sousedovi, který jej může zkonzumovat nebo předat dál



Aktivní prvky

Hub

- centrální prvek v hvězdicové topologii
- vše co mu přijde odesílá na všechny ostatní připojené zařízení

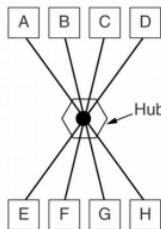
Switch

- centrální prvek v hvězdicové topologii
- vytváří si tabulku připojených zařízení, zná mac adresy připojených zařízení a podle nich přeposílá data
- pokud záznam v tabulce nemá odesílá všude a pokud se mu z nějakého portu vrátí odpověď tak si příchozí adresu uloží do své tabulky

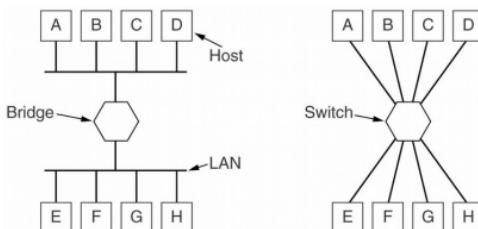
Bridge

- pracuje na 2 vrstvě OSI
- přemostění 2 lan sítí, může provádět změnu dat, konverzace podle typu připojeného média, kódování
- dnes byly nahrazeny switchy

## • Hub



## • Bridge/switch



## Router

- routování = Předávání paketů skok za skokem po nejkratší trase cestou přes síť k cíli
- Propojuje síť LAN na 3. vrstvě OSI RM.
- Router směruje příchozí pakety, hledá na který další router je má poslat, na základě hlavičky paketu a informací ze směrovací tabulky
- Aby směrovací tabulky nebyly příliš velké využívá se hierarchické směrování
- Pakety určené pro stanice připojené v síti routeru jsou odeslány přímo
- Směrovací tabulka může být nakonfigurována manuálně (statické směrování) nebo nebo vypočítána na základě informací vyměněných mezi sousedními routery (dynamické směrování)
- výměna informací mezi routery - směrovací protokol
- routery mohou obsahovat mnoho dalších funkcí - firewall, NAT, VPN, ..

## třídy směrovacích protokolů

### DVA - distance vector algorithms

- Směrovače neznají topologii sítě, ale pouze jejich rozhraní (adresy sousedů), přes která mají posílat pakety do jednotlivých sítí a vzdálenosti k těmto sítím (distanční vektory)
- na začátku směrovací tabulka obsahuje pouze přímo propojené sítě - staticky nakonfigurováno administrátorem
- tabulka je periodicky odesílána všem sousedům
- z došlých směrovacích tabulek sousedů (vzdáleností sousedů od jednotlivých sítí) a výběrem nejlepší cesty si směrovač postupně upravuje svou směrovací tabulku
- pokud trasa nebyla delší dobu sousedem inzerována, ze směrovací tabulky se odstraní
- metrikou je počet přeskoků na cestě mezi zdrojem a cílem - nezohledňuje parametry jednotlivých linek
- pomalá konvergence při změnách topologie - o změně se informuje až při příštím periodickém broadcastu směrovací tabulky
- příliš optimistické - směrovač se rychle učí dobré cesty, ale špatně zapomíná při výpadcích

- čekání na timeout cesty, která přestala být inzerována
- žádny směrovač nikdy nemá metriku horší než minimum z metrik sousedů + 1 - pomalé šíření špatných zpráv

#### RIP

- Routing information protocol
- starší - dnes se využívá v menších sítích
- jednoduchá implementace (nastavení)

#### LSA - link state algoritmus

- směrování na základě znalosti stavu jednotlivých linek sítě (funkčnost, cena)
- směrovače znají topologii celé sítě (graf) a ceny jednotlivých linek (ohodnocení hran). Tyto informace udržují v topologické databázi - všechny směrovače mají stejnou topologickou databázi
- každý směrovač počítá strom nejkratších cest ke všem ostatním směrovačům (a k nim připojeným sítím) pomocí Dijkstrova algoritmu - všechny směrovače počítají na základě stejných dat
- každý směrovač sleduje stav a funkčnost k němu připojených linek, při změně okamžitě šíří informaci o aktuálním stavu svého okolí všem ostatním směrovačům a ty si ji uloží do topologické databáze -> okamžitá reakce na změnu stavu linek

#### OSPF

- výpočet nejkratších cest a z toho vytvoří směrovací tabulku
- podporuje hierarchické směrování
- dnes jeden z nejpoužívanějších směrovacích protokolů

#### Hierarchické směrování

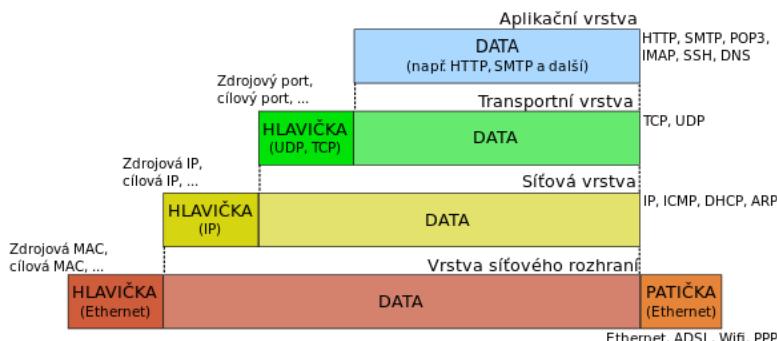
- rozdelení sítě do hierarchicky organizovaných celků
- směrovače v jednotlivých celcích znají jen topologii svého celku, cestu do vyššího celku a seznamy sítí v hierarchicky nižších celcích (nikoli jejich strukturu)
- Smyslem je omezení rozsahu směrovacích tabulek

Zde je popsáno směrování paketů tj. data jsou rozděleny na pakety a ty jsou směrovány ze zdroje do cíle. Existují i další (méně využívané) typy směrování vytváření okruhů - navázání spojení mezi zdrojem a cílem a přenášení dat po tomto spojení nebo Virtuální okruhy - virtuální okruh ale přenos je realizován pakety.

- **Protokolová rodina TCP/IP (TCP/IP model a jeho vztah k referenčnímu modelu ISO-OSI; síťové protokoly – IPv4 vs. IPv6; protokoly transportní vrstvy a princip fungování spolehlivého přenosového kanálu protokolu TCP).**

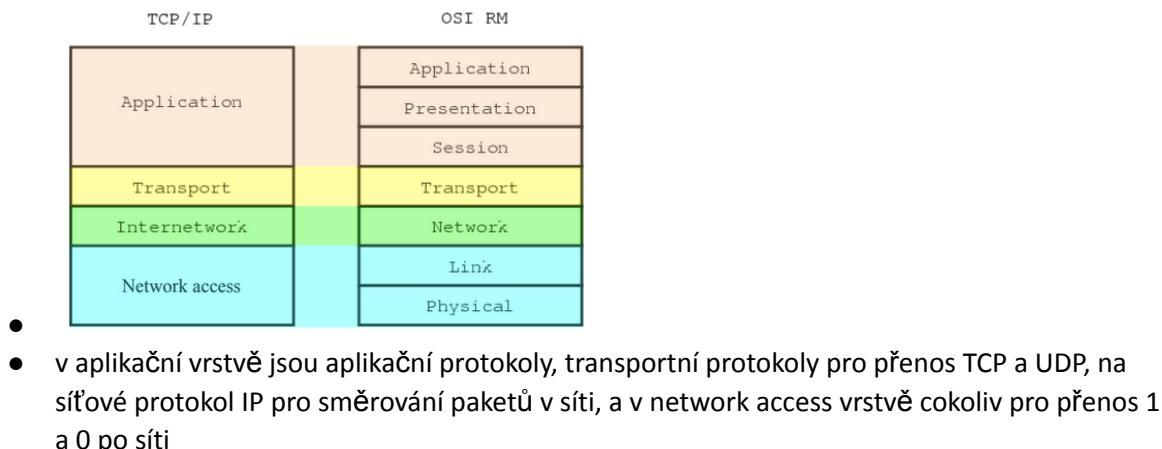
Cílem vrstvené architektury je dekompozice problému komunikace na menší snadněji řešitelné celky.

## ZAPOUZDŘENÍ DAT V SÍTI TCP/IP



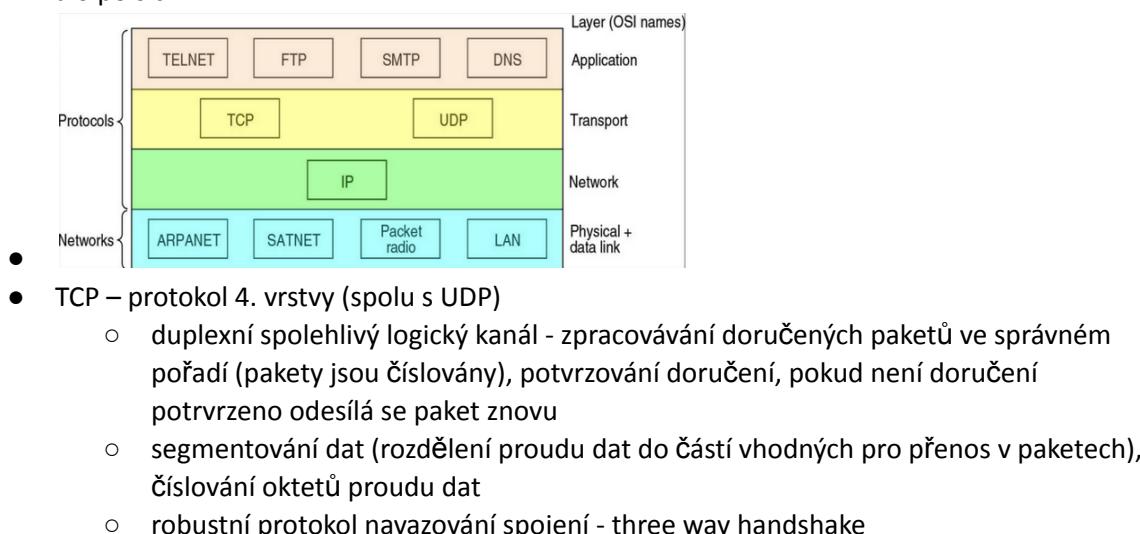
### TCP/IP

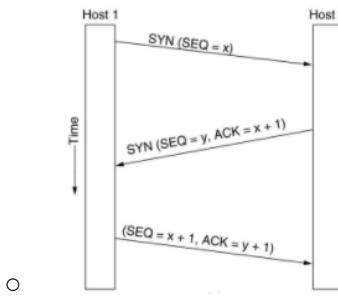
- standard pro komunikaci v Internetu
- (je ve stejné protokolové rodině jako http, smtp a další)
- oproti ISO OSI má pouze 4 vrstvy, zapouzdřuje stejnou funkčnost do méně vrstev



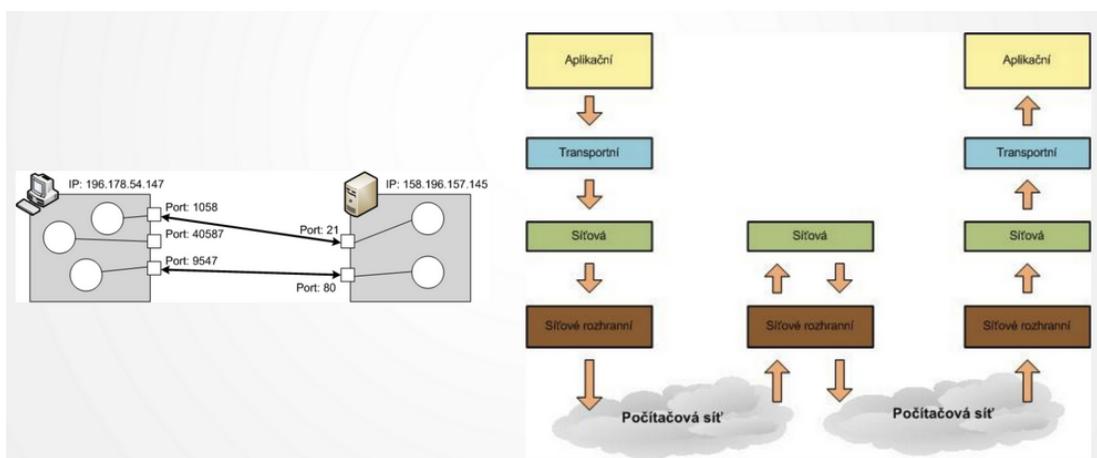
- v aplikaci vrstvě jsou aplikační protokoly, transportní protokoly pro přenos TCP a UDP, na

síťové protokol IP pro směrování paketů v síti, a v network access vrstvě cokoliv pro přenos 1 až 0 po síti





- 
- Uzavření spojení z obou stran
- Každá strana TCP spojení má přidruženo 16 bitové číslo portu, které specifikuje aplikaci. (známé aplikace mají pevně dané porty FTP 21 a 20, SMTP 25, DNS 53, HTTP 80)
- IP - protokol 3. vrstvy,
  - posílání nezávisle směrovaných paketů (bez navázání spojení)
  - ip identifikuje zařízení v síti
  - jakékoli zařízení, které pracuje na 3 a vyšší vrstvě musí mít ip adresu (směrovače pracují na 3 vrstvě, čtou ip adresu paketu a podle svých dat odesílá paket dále)
  -



rozdíl TCP a ISO-OSI

#### TCP/IP vs. OSI: What's the Difference

TCP/IP je starší než OSI model. Má méně vrstev, ale vrstvy řeší to co vrstvy OSI.

Model TCP/IP a model OSI jsou koncepcní modely používané k popisu veškeré síťové komunikace a samotný protokol TCP/IP je také důležitým protokolem používaným při všech operacích na internetu.

podívat na video ze sítí

#### Síťové protokoly

Slouží ke směrování paketů v síti

ipv4

- adresa 32b
- každé rozhraní prvku rozumějícího 3. vrstvě OSI RM připojené do sítě musí mít jednoznačnou IP adresu
- rozděleno na adresu sítě + adresu uzlu v rámci sítě

- k tomu se používá maska která určuje kolik bitů je adresa sítě a kolik uzlu
  - všechny stanice v síti mají společnou část ip adresy - směrovače nemusí ukládat názvy stanic, pouze adresy sítí
  - aby stanice napřímo mohly komunikovat musí mít stejnou adresu sítě (a různou uzlu)
- kdysi se využívaly třídy adres, které určovaly jaká část ip je pro síť a jaká pro uzel. Dnes se nevyužívá. Používají se beztřídní adresy, které mají specifikovanou masku určující délku prefixu.
- Ipv4 adres je omezené množství, jsou přidělovány oblastním správcem, dnes je nedostatek veřejných adres > využívá se NAT. Využívá se i podsíťování - vytváření sítí o co nejmenší nutné velikosti - určenou maskou. - Nutno pokud chceme více sítí a máme omezený počet adres.
- speciální adresy - první adresa sítě (192.168.1.0) a poslední broadcast (192.168.1.255)
- NAT - aby mohlo N strojů sdílet M adres, mají dynamicky vytvořené záznamy překladové tabulky časově omezenou platnost (timeout od posledního použití), při odstranění expirované položky se veřejná adresa vrátí zpět do poolu
- PAT - 1 veřejná a x neveřejných skrytých za touto adresou, rozlišeno podle portů
- hlavička obsahuje verzi, typ služby (přenosu?), kontrolní součet, velikost paketu (maximálně 65535 b), info o fragmentaci (kolik paketů přichází) - celá hlavička pouze v prvním paketu
- velikost hlavičky 20 - 64 Bajtů

## ipv6

- 128 bitů ( $3,4 * 10^{38}$  adres) - zapisuje se jako 8 hexadecimálních čtveřic oddělených :
- v zápisu je možno odstranit úvodní 0 z důvodu zkrácení
- díky velkému počtu adres není potřeba NAT, PAT, masky atd.
- broadcast (odeslání na všechny uzly na daném segmentu) byl nahrazen multicastem (odeslání na všechny stanice, které si o to řekly zaregistrovaly se do skupiny)
- novinka anycast - odeslání na nejbližší stanici s danou ip (můžu mít více uzlů se stejnou ip a odešle se na ten nejbližší (z hlediska směrování) - např. více serverů se stejnou ip, klienta obslouží ten nejbližší)
- hierarchické adresování - geograficky blízké sítě by měly mít blízké prefixy - to pomůže směrování, zmenšení směrovacích tabulek, rozsah se posílá někam dál jelikož ví že by zařízení s těmito ip by měly být poblíž sebe
- globální a linkové lokální adresy
- bezestavová autokonfigurace ip adresy bez dhcp serveru, nastavuje se samo na základě info z routeru nebo vygenerovat z vlastní mac adresy
- přechod na tuto verzi je pomalý - stále není příliš rozšířeno
- hlavička - verze, třída provozu, tok dat, délka dat, id další hlavičky (udp, tcp, icmp, ..), maximální počet přeskoků
- velikost hlavičky 40 Bajtů

## Protokoly transportní vrstvy

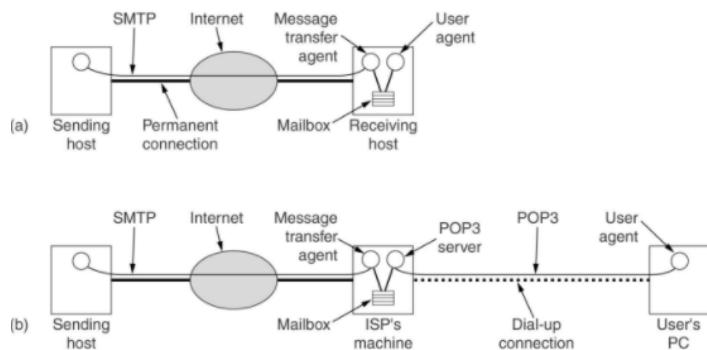
- Stará se o doručení dat k příslušnému aplikačnímu procesu na hostitelském počítači, umožňuje adresovat přímo aplikace pomocí čísel portů
- TCP - spolehlivý přenos dat - potvrzované
- UDP - nespolehlivý přenos dat - vyšší propustnost a kratší odezva - využívané např pro přenos médií kde je ztráta dat akceptovatelná

- Služby Internetu a jejich protokoly (elektronická pošta; protokol HTTP; DNS – typy záznamů, doménový strom, zóny, zabezpečení DNS; SSH).

Elektronická pošta

Mailbox - schránka pro ukládání pošty

Šifrování je možné zabalením do SSL



Protokol SMTP.

- Je orientován textově, neprovádí se šifrování a autentizace. Email lze podvrhnout.
- Šíření zpráv od poštovního klienta k poštovnímu serveru nebo mezi poštovními servery
- TCP/25 - nešifrováno neautorizováno
- na jednom spojení může následovat mnoho zpráv
- struktura - hello, mail from, rcpt to, data, turn, quit

protokoly pro přenos pošty mezi mailboxem a user agentem (user pc)

POP3 - post office protocol

- protokol pro vybírání obsahu poštovních schránek
- architektura klient-server
- TCP/110
- příkazy - user pass, list, retr, dele, rset, quit,..

IMAP - internet message access protocol

- dokonalejší obdoba POP3
- architektura klient-server, TCP/143
- předpokládá uchovávání zpráv na serveru (podpora "složek")
- vhodné zejména pro mobilní klienty
- vylepšené autentizační mechanismy
- cílem omezení dat přenášených na klienta
- možnost selektivního načítání zpráv a jejich částí
- možnost vyhledávání ve zprávách přímo na serveru
- (co provede klient u sebe projeví se i na serveru a opačně)

HTTP - hypertext transfer protocol

- model klient-server, požadavek - odpověď
- bezestavový protokol pro přenos dat mezi serverem a klientem, klient se dotáže a server

- odpoví
- TCP/80
- používá url k identifikaci zdroje
- využívá MIME - data se obalují hlavičkou mime (mime - multimedia mail extension, možnost strukturovat tělo zprávy a určení interpretace multimediálních dat (typ/podtyp - text/html), definuje způsob kódování binárních dat a informuje o struktuře zprávy)
- podporuje autorizaci přístupu a relokaci stránek
- metody http
  - GET - získání dokumentu specifikovaného zadanou o zadanou cestou/URL
  - HEAD - získání hlavičky dokumentu
  - POST - posílá data (webový formulář) na server
  - PUT - uloží dokument do složky na serveru
- odpovědi http
  - 1xx informace
  - 2xx úspěch
  - 3xx přesměrování
  - 4xx chyba na straně klienta
  - 5xx chyba na straně serveru

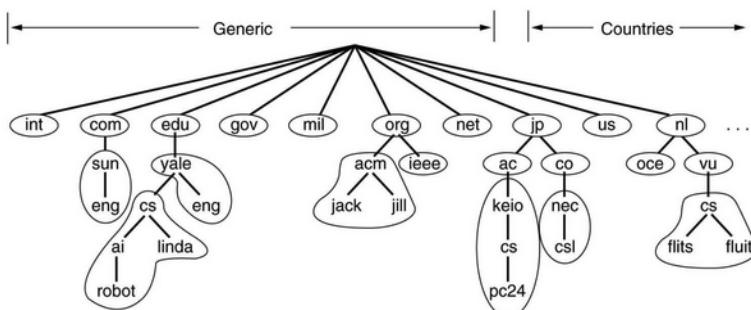
dopsat něco?

#### DNS - domain name server

- jmenná služba používaná v internetu
- zajišťuje mapování doménových jmen na ip adresy
- využívají distribuovanou databázi, DNS servery (name/jmenné servery)

#### Doménová jména

- hierarchická (stromová) struktura jmenného prostoru
- každý uzel lze identifikovat doménovým jménem, doména je skupina jmen se společnou pravou stranou, kořenem stromu je doména .
- doménové jméno je tvořeno spojením jména uzlu stromu se všemi jmény uzelů na cestě ke kořeni oddělovačem tečka
  - délka části mezi tečkami max 63 znaků
  - celková délka max 256



- strom je rozdělen do zón = část stromu, která je uložena na jednom DNS serveru
- server je autoritativní pro domény v jím spravované zóně

#### Vyhledávání v DNS databází

- Provádí SW klienta (resolver - provádí komunikaci s DNS) nebo rekursivní DNS server, po komponentech jména - začíná se od root serverů

- při dotazu na jméno, které není pod správou - odmítnutí dotazu, rekurzivní vyhledání a neautoritativní odpověď

#### Typy záznamů

- univerzální formát
  - doménové jméno
  - typ záznamu
  - data proměnné délky
  - time to live

| Type  | Meaning              | Value                                     |
|-------|----------------------|-------------------------------------------|
| SOA   | Start of Authority   | Parameters for this zone                  |
| A     | IP address of a host | 32-Bit integer                            |
| MX    | Mail exchange        | Priority, domain willing to accept e-mail |
| NS    | Name Server          | Name of a server for this domain          |
| CNAME | Canonical name       | Domain name                               |
| PTR   | Pointer              | Alias for an IP address                   |
| HINFO | Host description     | CPU and OS in ASCII                       |
| TXT   | Text                 | Uninterpreted ASCII text                  |

- 

#### Zabezpečení DNS

- odpovědi dns mohou být podvržené
- falešné mapování doménových jmen na ip a falešné mx a srv záznamy
- generování falešných odpovědí
- obrana DNSSec
  - autentifikuje dns servery - certifikáty
  - brání podvržení odpovědí
  - každý name server generuje veřejný a privátní klíč pro podepisování odpovědí jeho domény, každý příjemce ověřuje příchozí zprávu veřejným klíčem

#### SSH - secure shell

- obdoba telnetu (emulátor terminálu přes síť) ale provoz je šifrován
  - využití asymetrické kryptografie
  - klíče generovány automaticky Diffie-Hellmanovým algoritmem
  - podpora autentizace serveru
- postaveno na šifrovací spojově orientované službě vrstvy SSL (Secure socket layer) tcp/22
- lze využít i dalším účelům (přenos souborů)

- **Základy kryptografie (blokové a proudové šifry včetně jejich režimů; symetrická a asymetrická kryptografie a jejich využití; hashovací funkce; infrastruktura veřejného klíče – základní pilíře, certifikační autorita, certifikát veřejného klíče).**

Cílem kryptografie je utajení informace. Cílem šifrování je (po zadání klíče) změnit čitelnou zprávu tak aby se stala nečitelnou. Pro zčitelnění zprávy je potřeba provést opačný proces a zprávu dešifrovat.

Čitelná zpráva - plaintext, otevřený text

Šifrovaná zpráva - cipher text

operace převodu - šifrování a opačný proces dešifrování

plaintext se zašifruje šifrovacím algoritmem řízeně klíčem. Výstup je šifrovaná zpráva. Ta je dešifrována dešifrovacím algoritmem (obrácený postup šifrovacího). Musí mu být zadán klíč

(symetrický - stejný, asymetrický - druhý než kterým bylo šifrováno). Výstupem je opět plaintext. Kryptografie musí spočívat v utajení klíče (který řídí šifrování) ne v utajení algoritmu.

### Symetrická kryptografie

- kryptografie s tajným klíčem, používá se stejný klíč pro šifrování i dešifrování
- náročné distribuovat klíč tak aby zůstal utajen
- používá se pro zajištění utajení dat - relativně výkonné

2 typy:

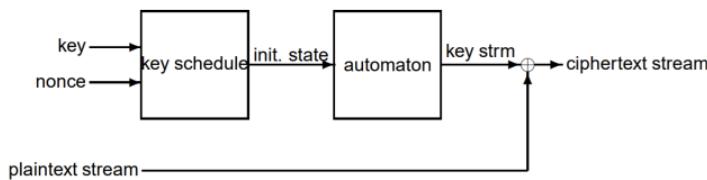
### Blokové šifry

- symetrická šifra pracující s bloky pevně stanovené délky
- při šifrování je každý blok zakódován pomocí šifrovacího algoritmu (který je řízený šifrovacím klíčem)
- Převádí n-bitová data (bloky) na jiná. Převod je řízen klíčem, proto lze dešifrovat.
- Mění pouze data v blocích pořadí bloků zůstává stejné?
- zástupci
- AES - využíváno pro wifi zabezpečení WPA2
  - využívá substituci - nahrazení Bytů (tj. Bajtů) za jiné (dle tabulky substitucí)
  - záměna Bytů (Bajtů) v řádku (rotace doleva) o různý počet bytů pro každý řádek
  - záměna sloupců - změna hodnot sloupců přenásobení maticí
  - přidání podklíče
  - tyto kroky se opakují 9 krát a po 10 se provedou bez záměny sloupců > výstupem je šifrovaný text
  - [AES Rijndael Cipher explained as a Flash animation - YouTube](#)
- ECB - electronic codebook mode
  - substituce kdy je blok otevřeného textu šifrován do bloku šifrového
  - každý blok je šifrován stejným způsobem, stejná data budou stejná i po zašifrování
  - nejjednodušší a nejrychlejší blokový režim, vhodné pro krátké zprávy
  - vhodné pro poruchová spojení - ztráta jednoho bloku neovlivní šifrování ostatních bloků, každý blok je šifrován nezávisle
  - snadná kryptoanalýza, opakovaný blok je šifrován stejně - lze budovat kódovou knihu odposlechem bez znalosti klíče
  - stereotypní začátky a konce zpráv
- CBC - cipher block chaining mode
  - Přidává k algoritmu mechanismus zpětné vazby.
  - Každý blok šifrového textu je použit pro zašifrování dalšího bloku otevřeného textu
  - Každý blok ŠT je závislý na všech předešlých blocích otevřeného textu.
  - Opakovaný blok je šifrován odlišně, pouze když je odlišný některý z předchozích bloků
  - Pro šifrování prvního bloku používá inicializační vektor (pseudonáhodná data)
  - Snadná softwarová implementace.
  - Šifrování je neparalelizovatelné, dešifrování ano - ztrátou některého bloku, následující bloky potom nelze dešifrovat.
  - Vhodný pro šifrování souborů.

### Proudové šifry

- otevřený text se zpracovává bit po bitu nebo bajt po bajtu
- rychlejší a pro implementaci stačí jednodušší hw narození od blokových šifer

- ale jsou náchylnější ke kryptoanalytickým útokům (pokud jsou nevhodně implementovány - počáteční stav nesmí být použit dvakrát)
- zástupci
- RC4
  - generuje pseudonáhodný proud bajtů (generování je řízeno klíčem) a používá spojení těchto bajtů spolu s plain textem operací xor
  - dešifrování stejným způsobem



Asymetrická kryptografie (kryptografie s veřejným klíčem) používá dvojici klíčů které jsou matematicky provázané.

- Asymetrická kryptografie vznikla jako reakce na obtížnou správu klíčů (např. distribuce klíčů) symetrické kryptografie.
- Používá se pro utajení, autentizaci, integritu a nepopiratelnost
- Dvojice klíčů - Soukromý klíč a veřejný klíč. Každý odesílatel i příjemce vlastní tuto dvojici klíčů a její použití se liší podle toho, zda chceme zprávu šifrovat nebo podepisovat.
- Klíče jsou matematicky provázané (generují se takové dvojice)
- Když je zpráva zašifrována jedním z nich může být dešifrována pouze tím druhým.
- Při odeslání zašifrujeme veřejným klíčem příjemce a je zaručeno že zprávu si bude moci přečíst pouze příjemce a to po dešifrování svým privátním klíčem.
- Další použití je digitální podpis. (příjemce je ubezpečen že autorem je označená osoba a s dokumentem nebylo manipulováno) Odesílatel pošle zprávu spojenou s haší zprávy. Tato hash je zašifrována(podepsána) odesíatelovým soukromým klíčem a zřetězena s odesílanou zprávou

Příjemce obdrží zprávu, rozloží ji na zprávu a podepsanou hash. Podepsanou hash dešifruje veřejným klíčem odesilatele. Vytvoří hash z příchozí zprávy a obě hashe porovná. Pokud jsou shodné je jistota kdo je odesilatelem a že se zprávou nebylo manipulováno.

- je nutné aby byl privátní klíč utajen!!!
- algoritmy
- RSA – šifrování, digitální podpis, výměna klíčů,
- Diffie-Hellman - výměna klíčů,
- DSS - digitální podpis,
- ElGamal - šifrování, digitální podpis, výměna klíčů,
- ECC (kryptografie na bázi eliptických křivek) - šifrování, digitální podpis, výměna klíčů.

### hashovací funkce

- Kryptografické hašovací funkce (KHF) jsou důležitými nástroji pro kryptografické aplikace, např. pro digitální podpis,
- Zajišťují integritu, dále ve schématech pro autentizaci a nepopiratelnost. Použití:
  - uložení přihlašovacích hesel - nepřímo pomocí hašovacích hodnot,
  - kontrola integrity dat (bez klíče),
  - klíčované hašovací autentizační kódy zpráv (HMAC),
  - digitální podpisy - výhodné, že haš má pevnou délku,

- kontrola správnosti kryptografických klíčů při dešifrování
- Jsou to speciální tzv. „jednocestné hašovací funkce odolné kolizím“ - zahashují zprávu tj. převedou ji do zašifrované podoby pevné délky a zpráva nelze převést do původního tvaru
  - hash má pevnou délku
  - můžou nastat kolize - pro dvě zprávy stejná hash
  - u odolných proti kolizím musí být náročné nalézt dvě zprávy se stejnou hashí
- Jednocestná hashovací funkce musí splňovat
  - Pro daný vstup  $M$  a danou funkci  $H$  je snadné vypočítat  $H(M)$ , ale pro danou  $H(M)$  je nesnadné vypočítat  $M$ .
  - Argument (zpráva)  $M$  může mít libovolnou délku.
  - Výsledek  $H(M)$  má pevnou délku.
  - Funkci  $H(M)$  lze relativně snadno realizovat jak hardwarově, tak softwarově.
  - Funkce  $H(M)$  je jednocestná.
- nastávají situace že pro různé vstupy je výstupem stejná hash - omezený počet hashí

### Infrastruktura veřejných klíčů

- Je potřeba důvěryhodně distribuovat veřejné klíče > certifikáty
- PKI (public key infrastructure) – soustava technických a organizačních opatření spojených s vydáváním, správou, používáním a odvoláváním platnosti kryptografických klíčů a certifikátů.
- PKI založeno na
  - bezpečnostní politika - definuje pravidla pro provoz celé infrastruktury PKI,
  - procedury - definice postupů pro generování, distribuci a používání klíčů,
  - produkty - HW/SW komponenty pro generování, skladování a používání klíčů,
  - autority - prosazují plnění bezp. politiky s pomocí procedur a produktů.
- Součásti PKI
  - certifikační autorita - poskytovatel certifikační služby, vydavatel certifikátu,
  - registrační autorita - registruje žadatele o vydání certifikátu a prověruje jejich identitu,
  - adresářová služba - prostředek pro uchovávání a distribuci platných klíčů a seznam odvolaných certifikátů.
- Certifikát
  - Certifikát (certificate) je elektronický dokument sloužící k identifikaci jednotlivce, serveru společnosti nebo jiné entity a spojuje tuto entitu s veřejným klíčem
  - certifikát spojuje jméno držitele páru soukromého a veřejného klíče s tímto veřejným klíčem a potvrzuje tak identitu entity,
  - poskytuje záruku, že identita spojená s vlastníkem daného veřejného klíče není podvržená
  - certifikát obsahuje
 

|                        |
|------------------------|
| veřejný klíč vlastníka |
| sériové číslo          |
| doba platnosti         |
| název vlastníka        |
| název CA               |
| digitální podpis CA    |
- Certifikační autorita

- Organizace, která se zabývá vydáváním certifikátů se nazývá certifikační autorita (certificate authority, CA). Potvrzuje identitu entit, jímž vydává certifikáty. Metody použité k potvrzování identity se různí a závisí na zásadách jednotlivých CA, CA je musí zveřejnit.
- CA provádí - registrace uživatelů certifikátů, vydávání certifikátů k veřejným klíčům, odvolání platnosti certifikátů, vytváření veřejného seznamu certifikátů, správu klíčů po dobu jejich platnosti, ..

- **Tvorba bezpečných aplikací (obecné principy; nejčastější zranitelnosti, jejich kategorizace a popis; bezpečnost databází a webových aplikací a typické útoky na ně).**

Bezpečnost je stav systému, kdy je možnost úspěšné dosud nezjištěné krádeže, falšování, narušení informací a služeb udržována na nízké nebo přípustné úrovni.

Pod pojmem bezpečnost ICT obvykle rozumíme ochranu odpovídajících systémů a informací, které jsou v nich uchovávány, zpracovávány a přenášeny.

Pro bezpečnost a správné chování IS by měly být dodržovány následující zásahy:

- autorizace
- spolehlivost
  - konzistence zamýšleného a výsledného chování IS
- autenticita
- nepopiratelnost odpovědnosti
- prokazatelnost odpovědnosti
- důvěryhodnost
  - je-li entita autorizována, můžeme důvěřovat tomu, že je tím, za co se vydává

Dále by měl IS splňovat pravidla CIA:

- Confidentiality - důvěryhodnost - k aktivům mají přístup jen autorizované subjekty
- Integrity - aktiva smí modifikovat jen autorizované subjekty
- Availability - aktiva jsou autorizovaným subjektům dostupná kdykoli je potřebují

Zranitelné místo (vulnerability) je slabina systému využitelná ke způsobení škod nebo ztrát útokem.

Zranitelnost snižuje informační bezpečnost. Je náročné vytvářet sw neobsahující zranitelnosti.

(terminologie

Hrozba (threat) – potenciální možnost (soubor okolností) využití zranitelného místa k útoku.

Existence hrozby představuje riziko tj. pravděpodobnost využití zranitelného místa.

exploit - program nebo sekvence příkazů, která využívá prog. nebo konfigurační chybu resp. zranitelnost k provedení neočekávané činnosti nebo k vyvolání neočekávaného chování, vedoucí většinou k získání prospěchu

Rootkit – po průniku do systému se zde útočník zabydlí, nahradí některé části (např. systémová

volání), aby zůstal skryt před uživateli.

**Autentizace** - ověření identity uživatele, uživatel je ten za koho se vydává

**Autorizace** - dává oprávnění pro určitou činnost

Autorizace - přidělení přístupových práv uživateli, který úspěšně absolvoval proces autentizace, respektive nepřidělení těchto práv uživateli, který autentizačním požadavkům nevyhověl.

Autorizace (authorization) entity pro jistou činnost rozumíme určení, že je z hlediska této činnosti důvěryhodná. Udělení autorizace si vynucuje, aby se pracovalo s autentickými entitami.

Autenticita (authentication) - původ informací je ověřitelný. Autentizací se rozumí proces ověřování pravosti identity entity (tj. uživatele, procesu, systémů, informačních struktur apod.).

)

zranitelnosti chyby při psaní bezpečných aplikací - hlavně u webových aplikací

Injekce

- Využití uživatelského vstupu pro upravení nebo rozšíření příkazu
- Nesprávná neutralizace zvláštních elementů použitých v "
- Předávaný řetězec může obsahovat znaky pro ukončení řetězce a předání nezamýšlených parametrů k dalšímu zpracování
- nejčastější zneužití - dotazy do databáze, parametry příkazového řádku při volání příkazu os
- řešení ověřování vstupů - sanitování vstupů, nahrazování zvláštních znaků

Buffer overflow

- Kopírování bufferů bez kontroly délky vstupu
- Jedna z typických chyb u prog. jazyků, které to umožňují
- Může vést k přepsání dat na zásobníku včetně ukazatele na návratovou adresu, proměnné významné pro další chod programu, nebo dokonce kódu programu.
- řešení
  - datové stránky paměti oddělené od kódu nelze přepsat kód,
  - Address Space Layout Randomization - adresní prostor programu je pokaždé jiná
  - ověřování délky vstupu

Cross-site Scripting (XSS)

- chybná neutralizace vstupu během generování www stránky, nežádoucí vložení javascriptu do stránky
- lokální (DOM based) - chybná neutralizace v parametru předaném lokálnímu interpretu v odkazu
- neperzistentní - v parametru, použitém při generování stránky na serveru je předám skript místo hodnoty, typicky v rámci odkazu. Většinou výsledek chybného escapování vstupu
- perzistentní - server si nekorektní vstup uloží (např. do databáze) a pak takto pozměněný obsah vrací komukoliv (např. diskuzní fórum s plným HTML editorem včetně <script>),
- obrana - escapování vstupů, validace vstupu, nemožnost uložení zvláštních sekvencí do db

Chyba v autentizaci a správě sezení (typické problémy):

- chybí potřeba autentizace pro kritickou funkci - např hlavní stránka chce přihlášení, ale zadáme-li natvrdo URL, funkce se provede i bez něj
- použití napevno zadaných přihlašovacích údajů - v kódu stránky, jako součást ladění nebo vzdáleného přístupu (úmyslný backdoor)
- Session ID je součástí URL
- Slabá kontrola při obnovení ztraceného hesla

- Chybné omezení nadměrného počtu pokusů o autentizaci
- Chybí rotace SID - sid nikdy nevyprší
- ukládání a zasílání citlivých informací v nezašifrované formě

#### Problémy s autorizací

- chybné ověření přístupu ke zdrojům
- Chybějící autorizace - uživatel se dostane k funkci, kterou by vůbec neměl mít k dispozici, ale byla mu poskytnuta kvůli chybějící správě oprávnění
- Chybná autorizace - obejití autorizačních mechanismů - např přepsání cookie se seznamem oprávnění uživatele

#### Problémy s důvěrností dat

- Nezabezpečené úložiště kryptografických informací a nedostatečná ochrana dat na transportní vrstvě
- Chybí šifrování citlivých dat - posílání nebo ukládání citlivých údajů v nezašifrované a nehashované podobě
- použití jednocestného hashování bez solení - nepřidá se řetězec k heslu před hashováním - lze využít tabulky hashí pro rozpoznání hesla
- použití slabého kryptografického mechanismu - slabý hash pro uložení hesla, slabý kryptografický algoritmus (malá délka klíče)

#### Nebezpečné přímé reference

- problémy se soubory a URL
- neomezené nahrání nebezpečného typu souboru např php
- procházení cestou, opuštění vyhrazeného adresáře (../../ u relativní cesty)
- zahrnutí funkcionality z nedůvěryhodného umístění - import (include, require) cizí knihovny např z cizího serveru
- zahrnutí externího souboru místo lokálního např. zobrazení cizího webu v iframe

#### Riskantní rozhodnutí

- závislost na nedůvěryhodných vstupech při bezpečnostním rozhodnutí
- např hodnoty cookie jsou nedůvěryhodné uživatel je může měnit (nevhodné pro autentizaci) - řešení kontrola stavu aplikace - uživatel nemůže měnit stav
- použití potenciálně nebezpečné funkce - knihovny se zranitelnostmi, rizikové funkce api

#### Cross-Site Request Forgery CSRF

- Podvržení uživatelského požadavku v prohlížeči pomocí automatizovaného mechanismu
- Některým automatizovaným mechanismem (onLoad, zneužití tagu <img>, XMLHttpRequest, URL generované skriptem, apod.)
- řešení - Kontrola Referer, zda požadavek pochází ze správné stránky, Použití unikátního uživatelského tokenu při odesílání formuláře

#### Open redirect

- Nevalidované přesměrování, přesměrování URL na nedůvěryhodnou adresu
- Typicky pokud je jedním z parametrů cílové URL. Dojde k přesměrování mimo stránky

#### Problémy s kódem programu

- stažení kódu bez kontroly oprávnění
- code injection - kód generován na základě uživatelských dat. V případě neošetření vstupů se podaří ovlivnit běh programu
- špatný výpočet velikosti bufferu
- přetečení celočíselné hodnoty

#### Problémy s daty

- Neřízený formátovací řetězec - podvrhnutí formátovacího řetězce pro zpracování vstupu a výstupu např v printf

- Nesprávná validace vstupů

Chyby při provádění programu

- Nemusí vést přímo ke zranitelnosti, ale způsobují problematické situace
- Existence postranního kanálu - chybová hlášení která může útočník číst a získávat info co se děje
- Chybějící informování o chybách
- Race condition – souběžné provádění nad sdíleným prostředkem bez synchronizace
- Špatná správa systémových zdrojů

typické útoky na webové aplikace a databáze - sql a php injection, xss, csrf, brute force - slovníkové útoky, dos

Jak vyvíjet bezpečné aplikace ... ? znát typické bezpečnostní chyby a snažit se jím vyvarovat.

Kontrola vstupů od uživatele

ještě něco dopsat?

- **Bezpečnost počítačových sítí (útoky na infrastrukturu, servery a aplikační protokoly, a jejich detekce; paketové filtry, stavový firewall; virtuální privátní sítě).**

zranitelnosti protokolů internetu

DoS - denial of service

- Cílem narušení fungování služby
- Cílem útočníka vyčerpání systémových prostředků síťového prvku/serveru/spoje, včetně pokusu o jeho zhroucení - zpomalení odezvy, vyčerpání hw prostředků
- Zpravidla generován provoz z podvržené zdrojové adresy (nedokončený tcp handshake)
- ddos - distributed dos, zdroj útoku není jeden

Spoofing

- Útočník vystupuje pod identitou někoho jiného, aby dodal útoku důvěryhodnost, obešel zabezpečení systému, poškodil subjekt pod jehož identitou vystupuje, získal údaje prostřednictvím sociálního inženýrství

Man in the middle

- útočník odposlouchává provoz v obou směrech
- přeruší přímý kanál a každé ze stran vystupuje v roli jejich partnera při komunikaci
- přeposílá data a zaznamenává komunikaci
- může modifikovat obsah přenášených dat

Podvržení odpovědi

- varianta spoofingu nebo man in the middle
- místo modifikace přenášených dat lze odeslat vlastní odpověď (odpověď z webu nahradím vlastní)

Replay útok

- použití zachycených dat k nějaké operaci, nemusím znát vnitřní reprezentaci dat
- např. zachytím komunikaci uživatele pro přihlášení ke službě a můžu ji sám použít.
- obranou jsou náhodná přihlašovací data např tokeny

Pokusy o získání autentizačních (přihlašovacích) informací

- Útok, použitý k odhalení hesla - oblíbená hesla, hrubou silou, slovníkový útok, rainbow

tables - pokud máme hash hesla a je použita kryptografická metoda bez „solení“, je možné hesla předpočítat a vyhledat rychle řetězec v předvypočtených tabulkách

útoky na 1-2 vrstvě

Fyzická vrstva

- odposlech dat
- DoS - přestřížení kabelu, zahlcení pásma
- MitM - bezkontaktní platby, nfc

Spojová

- MAC spoofing
  - skutečná mac adresa je nahrazena jinou nastavenou v os - obejití jednoduchých filtrů povolujících přístup pouze pro dané mac adresy
  - maskování identity útočníka
  - instalace sw svázaného s konkrétním hw
  - součást útoku, snažícího se přeplnit CAM tabulku přepínače - MAC flooding
- útoky na protokol ARP
  - snaha vytvoření neplatného mapování ip adresy na mac adresu
  - využití pro podvrženou mac adresu výchozí brány a následný MitM útok

útoky na 3-4 vrstvě

podvržení adresy

DoS útoky:

SMURF útok

- Založen na úvaze, že ICMP echo reply je pouštěno zpět bezestavovým firewallem - útočník zasílá velké množství icmp paketů s falešnou zdrojovou IP a zařízení odesílají odpověď na tuto zdrojovou adresu
- Využíval zesílení útoku použitím broadcastu na síti jako cílové adresy
- v dnešní době detekováno a filtrováno

SYN flooding

- zahlcení serveru požadavky o TCP spojení, které nebudou nikdy realizovány
- Často opět využit IP spoofing nebo botnet(síť napadených počítačů)
- U distribuovaného útoku hůře filtrovatelné

LAND útok

- Speciální paket s SRC = DST - zacyklení komunikace zařízení posílá požadavky na sebe samotného
- většinou je ošetřeno a lze blokovat na firewallu

napadení probíhajícího TCP spojení

- Predikce sekvenčního čísla u TCP - narušení komunikace, např. potvrzování dat která ještě nebyla doručena
- násilné ukončení spojení
- replay útoky pro potvrzení
- SHREW - zahlcování spoje v intervalech

Útoky na směrovací protokoly

- lze zjistit informace o topologii sítě
- narušení směrovacích tabulek distribuováním falešných směrovacích informací

## Útoky na aplikační protokoly

### na DHCP

- útočník může v síti spustit vlastní dhcp server - může blokovat odpovědi původního serveru nebo rychlejší odpověď
- dnes pouze povolené dhcp servery - blokování na jiných portech přímo na switchi

### na DNS

- přesměrování dns dotazů na server útočníka
- dns spoofing - podvržená odpověď správného dns serveru
- řešení podepisování zpráv od dns klíčem

### na HTTP servery

- přesměrování požadavku na transparentní http proxy
- podvržení http hlaviček
- Low rate útoky
  - low bandwidth dos - nejdou cestou zahlcení šířky pásma ale vyčerpáním paměti/prostředků
  - slowloris - pomalé zasílání http hlaviček v požadavku tak, aby server neukončil předčasně spojení
  - slow http post - v hlavičce post je specifikována délka dat, data jsou posílána extrémně pomalu
  - slow read - chování jako uživatel na pomalé lince, extrémně pomalé čtení odpovědi ze serveru

### na elektronickou poštu SMTP

- lze podvrhnout hlavičku zprávy - odesilatele

### na SQL servery

- dos - překročení počtu povolených spojení mezi skripty na http a databází
- session hijacking - převzetí session mezi databází a sql serverem - útočník může posílat vlastní příkazy
- sql injection - využití neošetřených uživatelských vstupu pro úpravu dotazu

Firewall slouží k řízení a zabezpečování síťového provozu mezi sítěmi s různou úrovní důvěryhodnosti a zabezpečení. Definuje pravidla pro komunikaci mezi sítěmi, které od sebe odděluje.

### paketové filtry

- první typ firewallu
- sleduje síťové adresy a porty paketu, aby zjistil, zda má být tento paket povolen nebo zablokován. (nastaveno konfigurací co má být povoleno)
- lze obejít spoofingem

### stavový firewall

- zaznamenává všechna připojení procházející skrze síť - dle paketů pro začátek a konec spojení (ty propustí aby se navázalo spojení)
- kontroluje pakety a testuje jestli je součástí stávajícího spojení nebo není součástí spojení. (asi propouští pouze ty co mají spojení)

### vpn

- prostředek pro bezpečné propojení několika počítačů prostřednictvím nedůvěryhodné počítačové sítě
- Lze snadno dosáhnou stavu kdy spojené počítače budou mezi sebou komunikovat, jako

- kdyby byly propojeny v rámci jediné privátní sítě.
- navazování spojení ověřováno pomocí digitálních certifikátů
- pomocí vpn se lze vzdáleně připojit do lokální sítě

**Příklad otázky:** Proč se dnes používá pro komunikaci s webovými servery převážně HTTPS? Jakému typu útoku má bránit, jakou bezpečnostní funkci má primárně zajistit, a jaký význam pro komunikaci se HTTPS servery má certifikační autorita?

## Počítačová grafika (ZPG, URO)

- **Metody a nástroje pro realizaci grafických uživatelských rozhraní (kognitivní schopnosti člověka, mentální modely, základní pravidla designu, barevné prostory, volba barev a prezentace textu).**

kognitivní schopnosti člověka (schopnosti zpracovávat informace)

- vstup informace (vizuálně, sluchem, hmatem), vyhodnocení informace, stanovení reakce, provedení reakce
- v gui člověk vnímá informace vizuálně
- člověk má senzorickou paměť (paměť vjemů), krátkodobou paměť a dlouhodobou
- gui by nemělo přetěžovat paměť uživatele - malý počet položek čehokoli, co by mělo být při obsluze programu člověkem krátkodobě zaregistrováno
- při konstrukci gui je vhodné jej vytvářet tak jako něco co už uživatel zná a nebude se muset učit nové gui (typické ikony atd)

Gestalt teorie

- Popisuje lidské vnímání celků složených z částí.
- člověk obraz rozděluje na objekty a pozadí. K rychlé interpretaci obsahu přispěje jednoduché pozadí
- člověk si i složité tvary pamatuje jako jednoduché - v gui používat jednoduché tvary (obdélník, kružnice)
- Některé tvary mají význam např šipky
- Idealizovaný tvar - čím je tvar jednodušší je snáze vnimatelný
- blízkost objektů symbolizuje jejich vazbu > vytváření skupin aby byly snadno rozpoznatelné a oddělitelné
- symetrie - symetrie člověku usnadňuje vnímání a zapamatování.

UI Golden rules - základní pravidla designu

- konzistence - podobné posloupnosti akcí v podobných situacích, konzistence terminologie, podobný vzhled oken, ..
- Informativní zpětná vazba - na akci uživatele reagovat zpětnou vazbou informující co se děje nebo stalo
- prevence chyb a řešení chybových situací - nedovolit uživateli udělat chybu (např ve formulářích), když chyba vznikne informovat jak ji odstranit
- undu - možnost vrátit se o krok zpět
- plná kontrola nad produktem - zkušený uživatel chce mít plnou kontrolu, pokud nemá

- má pocit nespokojenosti nad neočekávaným chováním
- produkt také vhodný pro zkušené uživatele - gui zkušené uživatele zdržuje - klávesové zkratky, příkazový řádek
- organizace akcí do uzavřených celků - rozdělit větší akce na menší celky (formuláře, kroky úkonů)
- nepřetěžovat krátkodobou paměť - přiměřený počet položek v menu, tlačítek, nedopustit chaos na obrazovce

## Mentální modely

- člověk ve věcech hledá pořádek a řád
- snaží se vytvořit mentální model (mapu) celého gui
- návrhář a uživatel by si měli vytvořit podobný mentální model - pokud má každý jiný něco je zle
- např. formulář - ví jaký typ info se vyplňuje v jaké sekci, v gui ví kde najde info o firmě, nabídku, ..
- když si vytvoří mentální model - pocit jistoty a ovládnutí produktu, pocit zvládnutí nových situací, pokud nevytvoří - nejistota zda zvládne řešit nové úkoly
- Při návrhu GUI směřujeme úsilí k tomu, aby si uživatel udělal správný mentální model produktu, okna, stránky co nejrychleji.
- autor by se měl rozhodnout jaký mentální model by si měl uživatel vytvořit a odpovídajícím designem mu jej vnutit
- chyby
  - neutřízený obsah oken a stránek - není žádný model
  - obsah je utřízen ale uživatel nechápe klíč utřízení - nepřesvědčivý model
  - obsah oken a stránek je rozumný, ale nevhodný vzhled ztěžuje uživateli pochopení obsahu a vytvoření modelu - špatná reprezentace modelu
  - vytvoření modelu ztěžují nadbytečné grafické prvky

## Barevné prostory ?? v prezentacích o tom nic není

[https://cs.wikipedia.org/wiki/Barevn%C3%BD\\_prostor](https://cs.wikipedia.org/wiki/Barevn%C3%BD_prostor)

- Barevný prostor je předem definovaná množina barev, kterou je schopno určitě zařízení snímat, zobrazit nebo reprodukovat.
- Barevný prostor je ve většině případů založen na barevném modelu (RGB, CMYK), ale na rozdíl od barevného modelu má barevný prostor standardizované odstíny základních barev.
- sRGB, adobe rgb, ...

## Volba barev a prezentace textu

- volba barev je obtížná je vhodné používat co nejméně, poučit se se od jiných řešení nebo kombinací barev v přírodě
- je vhodné volit málo syté barvy - vypadá decentně a harmonicky
- nebo jedna barva a různé odstíny
- volit barvy podle prezentovaného obsahu (červené logo firmy - červené barvy atd)
- k dominantní barvě můžeme zvolit velmi blízkou barvu

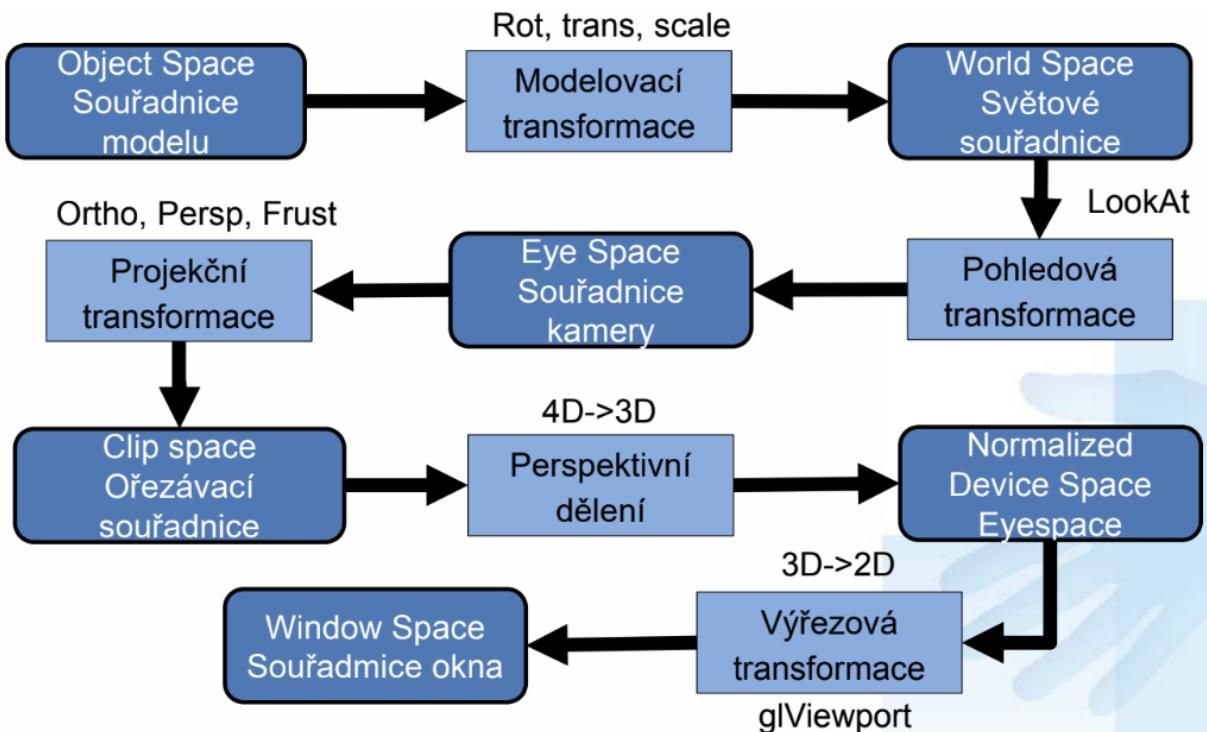
- je mnoho zrakově handicapovaných - gui by mělo být navrženo tak aby šlo používat i když nevidím barvy

text

- V podstatě platí zásady pro „normální“ tisk. Jen je třeba vzít v úvahu některé specifické okolnosti elektronické prezentace.
- taky rozdělovat text do menších bloků přitahující pozornost - členění na odstavce, přiměřená délka odstavců
- dnes většinou navigační sloupec vlevo a text vpravo
- konzistence vzhledu mezi jednotlivými stránkami gui
- Klasické knižní fonty (Times New Roman) mají patku, která vede oko po řádku (důležité u dlouhých řádků). Znaky jsou relativně malé a tenké, což vede k vysokému využití plochy. Při použití na obrazovce mohou ale být špatně čitelné (závisí na velikosti znaků a barvě pozadí).
- Pro prezentace na obrazovce není patka často zapotřebí, protože řádky jsou krátké. Žádoucí je větší tloušťka a větší vzdálenost znaků.
- rady
  - Používejte jen jeden, nejvýše dva fonty.
  - Používejte nejvýše tři velikosti znaků.
  - Varianty italic a bold používejte velmi omezeně – na obrazovce jsou hůře čitelné.
  - nepodtrhávat text (stačí že jsou odkazy podtržené)
  - Nepište dlouhé texty velkými písmeny.
  - Volte vhodnou barvu písma a vhodné pozadí.
- Pozadí je zapotřebí volit s největší opatrností. Ve všech uvedených ukázkách pozadí zhoršuje čitelnost. I zdánlivě nevinné pozadí může v případě delších textů způsobovat únavu. Člověk je nejvíce zvyklý na pozadí bílé.

- **Standardní zobrazovací řetězec (realizace jednotlivých kroků řetězce, modelovací a zobrazovací transformace, Phongův osvětlovací model, řešení viditelnosti, identifikace těles, stručná charakteristika standardu OpenGL a jazyka GLSL).**

popis převodu object space do windows space v rámci vykreslovacího řetězce (obecného)



- Vstupem do vykreslovacího řetězce je model (jeho vrcholy) v lokálním souřadném systému.
- Následně se na jednotlivé vrcholy modelu aplikují transformace. Tím dojde k převodu do globálního souřadného systému. (tj. přesné umístění modelu ve scéně)
  - zobrazovací transformace:
  - posun
    - $A' = A + b$
  - rotace
    - rotace kolem osy z
    - $x' = x * \cos a - y * \sin a$   
 $y' = x * \sin a + y * \cos a$   
 $z' = z$
  - změna měřítka
    - $x' = S_x * x$   
 $y' = S_y * y$   
 $z' = S_z * z$
- Z důvodu zefektivnění výpočtu se všechny transformace realizují jako násobení matic.
- Kvůli posunu je potřeba rozšířit matici o jednu dimenzi, využívají se tzv. homogenní souřadnice - nová homogenní souřadnice se většinou nastaví na hodnotu 1, všechny ostatní souřadnice se totiž násobí touto souřadnicí
- Díky tomu se operace omezí na násobení matic
- posun

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- rotace

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- zde rotace kolem osy z
- rotace kolem jiných os - cos a sin není v řádku a sloupci osy podle které se má provádět rotace

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- změna měřítka

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- Tako se veškeré transformace omezí na násobení matic
- Matice transformací se poté násobí s vektorem (pozicí) vrcholu(násobí se ještě s dalšími maticemi - výstupem vektor)

- Pohledová transformace - přenásobí se pohledovou maticí tj. maticí kamery - přenásobením se dostane do eye space souřadnic
  - ta je definována pozicí kamery, cílem/bodem kam se dívá (vypočteno sférickým souřadným systémem) a right vektorem (vektor směr vpravo od pozice kam se dívá)

**LookAt(eye, look, up);**

$$v=|(look-eye)|$$

$$r=|(v \times up)|$$

$$u=|(r \times v)|$$

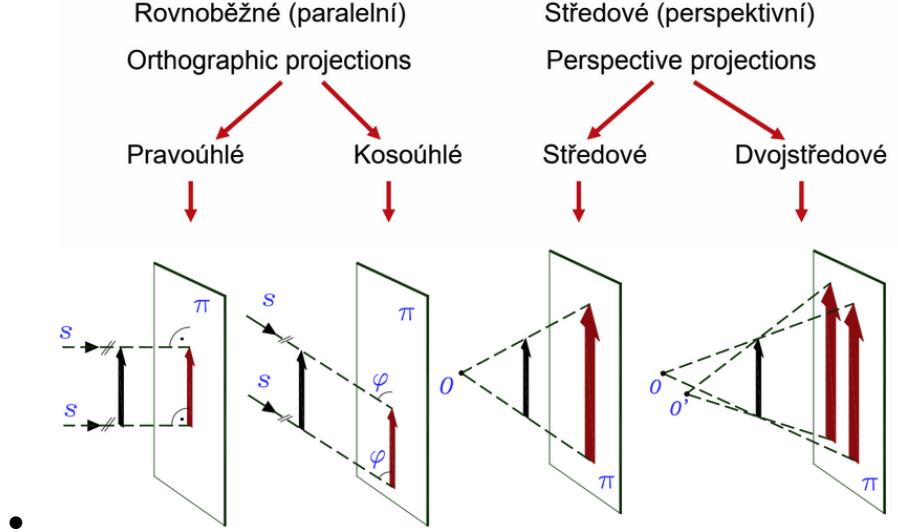
$$M = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -v_x & -v_y & -v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Výsledná *LookAt* matice  $L$  je dána jako

- $L = MT$

- tímto přenásobením se definuje kam se dívá
  - nerotuje se kamera ale všechny objekty před kamerou - pohledovou maticí se totiž násobí všechny vrcholy objektů  
  - Následně se vrcholy násobí projekční maticí tj. maticí určující typ promítání - po tomto přenásobení se projekční maticí > projection space



- typy promítání
    - rovnoběžné - pravoúhlé, kosoúhlé (nárys, bokorys, půdorys, mongeovo promítání)
    - středové (perspektivní) - středové, dvojstředové
  - v 3D grafice se používá středové
  - Následuje perspektivní dělení převod z homogenního souřadného systému do kartézského (4D do 3D, dělá se dělením všech hodnot poslední homogenní souřadnicí)
  - Nakonec převod do windows space 3D do 2D (rasterizace?)

Výpočty týkající se zobrazování probíhají v shaderech. Shader je program určený pro řízení konkrétní části vykreslovacího řetězce GPU.

## Vertex shader

- výpočet polohy vrcholů (tj. transformace vrcholů, normál, světla)
  - Mění polohu vrcholu, ale nemůže ani přidávat ani rušit jednotlivé vrcholy.
  - Nemá informaci o tělese (nezná sousední vrcholy)
  - Vypočtené hodnoty se z vertex shaderů interpolují do fragmentů

## Fragment shader

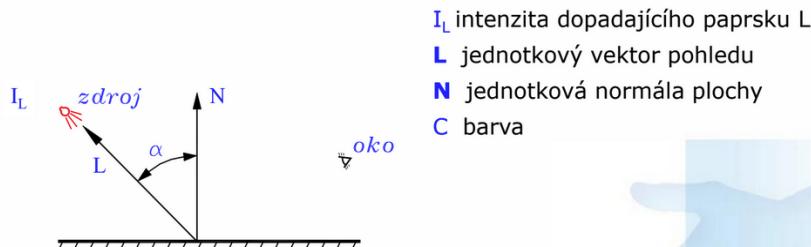
- výpočet barvy pro jednotlivé fragmenty (fragment se poté může podílet na barvě pixelu, v pixelu může být více fragmentů)
  - Výpočet osvětlení, mlhy atd.
  - Nemůže změnit souřadnice

Geometry shader - změna geometrie v určitém čase

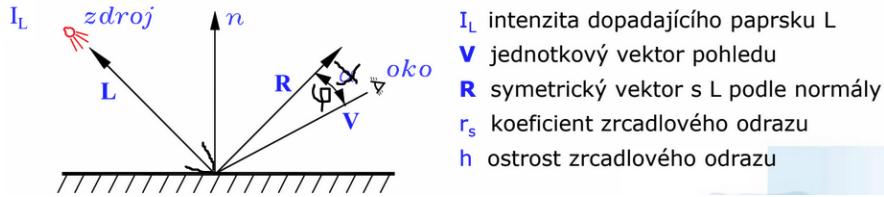
Tessellation shader - zjemňování trojúhelníkové sítě

Phongův osvětlovací model - jeden z hlavních typů fragment shaderu

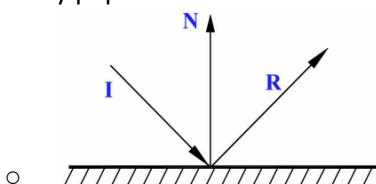
- Osvětlovací model nám umožňuje vytvořit iluzi o vlastnostech materiálů zobrazovaného tělesa. K tomu nám pomáhají různé typy fragment shaderů. Phongův osvětlovací model přidává kromě barvy tělesa také další složky:
- Ambientní, difúzní a zrcadlová složka
- Vytváří na povrchu odlesky
- $I_v = I_a + I_d + I_s$
- ambientní + difuzní + spekulární(zrcadlová)
- ambientní - konstantní hodnota aby těleso nebylo úplně černé
- difuzní - nastavení jasu podle úhlu mezi světlem a normálou fragmentu



- spekulární - zrcadlový odlesk tj. úhel mezi odrazem světla a kamerou



- $I_v = I_A r_a + \sum_{i=0}^m I_{Li} (r_d \cos \alpha_i + r_s \cos^h \varphi_i)$
- $r$  - koeficienty pro jednotlivé složky (např. pro útlum s vzdáleností od světla)
- suma - výpočet pro všechny světla ve scéně - proto indexy i identifikují světlo
- $\cos \alpha$  - úhel mezi světlem a normálou fragmentu - úhel lze vypočít skalárním součinem
- $\cos \varphi$  - úhel odrazu světla a kamery - mocněno koeficientem  $h$  - určující ostrost zrcadlového odrazu, čím větší tím je odlesk koncentrovanější a ostřejší
- odražený paprsek reflect =  $I - 2.0 * \text{dot}(N, I) * N$ .



### Lokální zobrazovací metody

- Pouze přímé osvětlení.
- Objekty se navzájem neovlivňují z hlediska osvětlení (vidíme jen to, co je osvětlené).
- Počítáme pouze s jedním odrazem paprsku od povrchu objektu. Každý objekt ve scéně má individuálně (lokálně) vyhodnocen osvětlovací model nezávisle na ostatních objektech.
- Jediný případ, kdy se pozice objektů bere v potaz, je při vypočtu viditelnosti

### Globální - vícenásobné odrazy světla, ray tracing

#### Řešení viditelnosti

- Může být více překrývajících se objektů (za sebou) a je potřeba určit který bude zobrazen tj. který je blíže kamere

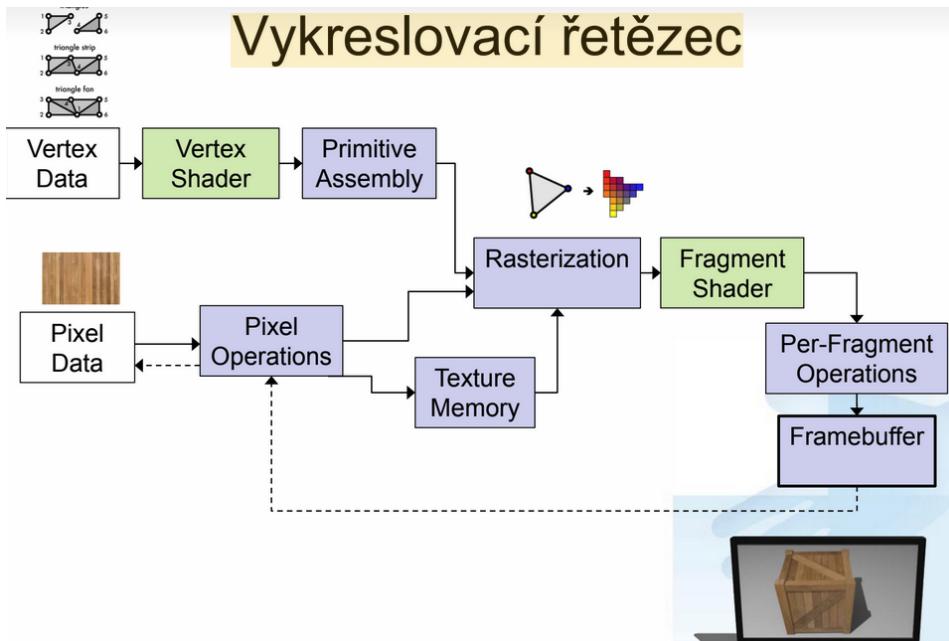
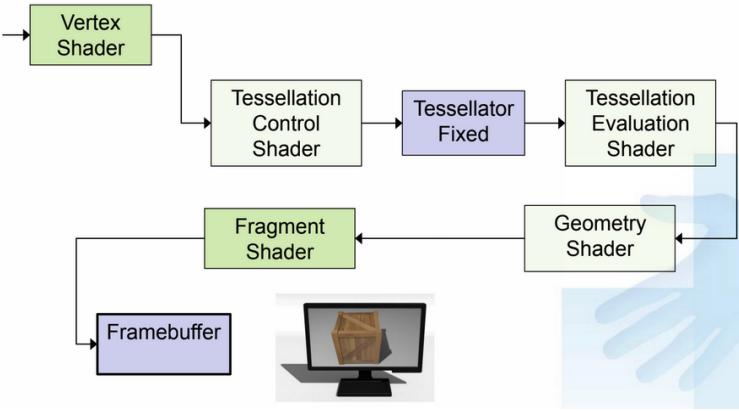
- Malířův algoritmus
  - postupné nanášení barev
  - je nutno seřadit objekty od nejvzdálenějšího k nejbližšímu a v tomto pořadí je vykreslovat
  - problém pokud se objekty překrývají, nelze určit který je vzdálenější, nutno rozdělit na více menších objektů
  - problém je dopředu nevíme jak bude výpočetně náročné
- Plovoucí horizont ?? asi nezmiňovat
  - vhodné pro funkci 2 proměnných
  - horní a dolní horizont mezi něž se vykresluje
- Z-buffer
  - paměť hloubky
  - vyplní barvu pixelu (do paměti) a uloží hodnotu hloubky fragmentu tj. vzdálenost od kamery
  - pokud na stejný pixel bude zapisovat další hodnotu, zapíše ji pouze je-li hodnota z (vzdálenosti) nižší než uložen v bufferu (případně přepíše pixel a uloží novou hloubku)
  - největší vzdálenost 1, nejmenší 0, když se začíná je vše nastaveno na 1
  - výpočetní náročnost roste lineárně s počtem ploch
  - pokud se objekty nepřekrývají je efektivnější malíř jinak z buffer

#### Identifikace těles - Stencil buffer

- Při vykreslování jednotlivých fragmentů do frame bufferu si budu značit id tělesa, abych pak poznal o které těleso se vlastně jedná.
- Při případném kliknutí na konkrétní pixel dokážu určit na jaké těleso těleso bylo kliknuto. (mám seznam s id těles)
- případně lze i dopočít globální souřadnice (případně i lokální) kde bylo kliknuto zpětným průchodem vykreslovacího řetězce (glm funkce unproject)

#### OpenGL a GLSL

- Grafická knihovna určena pro tvorbu 2D a 3D aplikací
- OpenGL a DirectX - nejrozšířenější hardwarově podporované knihovny (přímo výrobci GK), další knihovny jsou většinou postaveny nad těmito technologiemi
- starší verze měly fixní vykreslovací řetězec, nešlo jej ovlivňovat
- novější verze přidávají možnosti shaderů tj. počítačový program určený pro řízení konkrétní části vykreslovacího řetězce
- Pro tvorbu shaderů slouží specializované programovací jazyky tzv. shader jazyky, v OpenGL je jazyk GLSL
- základní typy shaderů - vertex, fragment, geometry, tessellation
- (Ne)unifikované shadery – část čípu vyhrazena pouze pro konkrétní shader (Pevně daný počet vertex a fragment shaderů)
- Unifikované shadery – možnost měnit dynamicky typ shaderu. (lze počítat jak libovolné shader úlohy (vertex, fragment, geometrický, teselační).
  - podle aktuální potřeby lze využívat všechny shadery
  - Unifikované shadery však nemohou být tak efektivní jako shadery určeny pouze pro konkrétní typ úloh.
- zjednodušené schéma **vykreslovacího řetězce OpenGL** (po vertex muze tessall., geometry)



dodatek k opengl - <http://lucie.zolta.cz/index.php/pocitacova-grafika/22-volitelna/61-open-gl>

OpenGL je grafická knihovna pro zobrazování 2D a 3D objektů vyvinutá v 90. letech. Dnes je o všeobecně uznávaný standard podporován výrobci grafických karet. Standard OpenGL definuje množinu funkcí, které se volají z programu. Pokud nejsou některé z těchto funkcí podporovány na technické úrovni, je podpora realizována programově, což zajišťuje široké využití i při zachování technické nezávislosti programu.

OpenGL je jednoduchý, nepodporuje objektový orientované programování. Přesto nabízí široké možnosti a urychluje práci s grafikou. Kromě vykreslování základních typů objektů, umožňuje OpenGL transformace. A to **transformace zobrazovací** a **transformace modelovací**. Při těchto transformacích se pracuje s transformačními maticemi.

- **Geometrické modelování (affinní a projektivní prostory, popis těles a možnosti jejich reprezentace, základní křivky používané v počítačové grafice, jejich vyjádření, vlastnosti a použití, Fergusonova kubika, Bézierova křivka).**

<http://lucie.zolta.cz/index.php/pocitacova-grafika/22-volitelna/53-affinni-prostor>

<http://lucie.zolta.cz/index.php/pocitacova-grafika/22-volitelna/54-projektivni-prostor>

**Afinní prostor** je prostor s body. Dále obsahuje přidruženým vektorový prostor (souřadný systém) pomocí kterého je možné jednotlivé body prostoru zaměřit. Součástí affinního prostoru je také zobrazení, které přiřadí dvojici bodů vektor. - prostor pro práci s vektory

**Euklidovský prostor** je affinní prostor v kterém je zaveden skalární součin a norma coby odmocnina skalárního součinu. To umožňuje měřit délku vektorů a úhly mezi nimi.

Affinní transformace je zobrazení bodů jednoho affinního prostoru do jiného.

Kartézská soustava souřadnic - Souřadné osy vzájemně kolmé, protínají se v 1 bodě, jednotka se obvykle volí na všech osách stejně velká, souřadnice polohy bodu je možno dostat jako kolmé průměty polohy bodu k jednotlivým osám.

**Projektivní prostor** vzniká z potřeby vyrovnat se s nevlastními body (body v nekonečnu) se kterými se špatně počítá. Projektivní prostor využívá **homogenních souřadnic**.

Homogenní souřadnice zavádíme proto abychom mohli reprezentovat body v prostoru o jednu dimenzi větším. To usnadňuje nejrůznější výpočty - omezeno jen na násobení matic.

Transformace je díky homogenním souřadnicím jednodušší, protože k ní stačí pouze transformační matice. Na rozdíl od affinní transformace, která ještě měla zvlášť vektor posunu. Rozšiřuje se o 1 dimenzi - všechny hodnoty se násobí novou souřadnicí (volí se hodnota jedna). Při převodu zpět do affinního(kartézského) prostoru se musí podělit poslední homegenní souřadnicí.

Bod v affinním prostoru (kartezsky souřadný systém)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

Bod v projektivním prostoru s homogenním souřadným systém

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & b_x \\ a_{10} & a_{11} & b_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Příklady transformační matic

Posunutí

$$\begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix}$$

Otočení

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

Změna měřítka

$$\begin{bmatrix} z & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$z>1$  zvětšení,  $z<1$  změšení

Zkosení:

$$\begin{bmatrix} sh & 0 & 0 \\ 0 & sh & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

V počítačové grafice se pro provádění transformací převádí na homogenní souřadnice. Před vykreslením obrazu je však třeba se vrátit do affinního prostoru a přepočítat homogenní souřadnice na kartézské.

Reprezentace těles

**polygonální síť** – nejrozšířenější způsob zápisu 3D objektů. Každý polygonálně definovaný prostorový objekt se skládá z uspořádané skupiny spojených rovinných n-úhelníků, neboli polygonů. Ty jsou tvořeny minimálně trojicí bodů, neboť nejjednodušším plošným útvarem je trojúhelník. Častěji se ovšem využívají obdélníky. DirectX či OpenGL, vyžadují polygonové modely, neboť pouze ty mohou být akcelerovány současným široce dostupným hardwarem a tím pádem prezentovány v dostatečné rychlosti.

[https://wikisofia.cz/wiki/Z%C3%A1klady\\_reprezentace\\_trojrozm%C4%9Brn%C3%A9ho\\_prostoru\\_v\\_p%C4%8D%C3%ADta%C4%8D](https://wikisofia.cz/wiki/Z%C3%A1klady_reprezentace_trojrozm%C4%9Brn%C3%A9ho_prostoru_v_p%C4%8D%C3%ADta%C4%8D)

těleso se skládá z trojúhelníků nebo z čtyřúhelníků.

další možností že je popsáno vrcholy, které se vykreslují

nebo

<http://lucie.zolta.cz/index.php/pocitacova-grafika/22-volitelna/63-reprezentace-telesa-hranici> ale to jsme se určitě takto neučili

Křivky používané v PG

Křivka je trajektorie bodu při spojitém pohybu. Je to nekonečná množina bodů závislá na jediném parametru. Křivky můžou být rovinné, prostorové, interpolační (procházejí danými body), approximační (blíží se k daným bodům).

<http://lucie.zolta.cz/index.php/pocitacova-grafika/22-volitelna/56-krivky>

Parametrické křivky:

- V trojrozměrném euklidovském prostoru každému číslu  $t$  odpovídá na křivce příslušný bod  $P(t)=[x(t),y(t),z(t)]$ .
- Polohový vektor  $p$  bodu  $P$  je vektor daný počátkem souřadné soustavy a souřadnicemi příslušného bodu  $P$ .

Pro práci s křivkami v počítačové grafice potřebujeme aby měly tyto vlastnosti:

- lokalita změn - změna polohy jednoho řídícího bodu křivky se projeví pouze na omezeném úseku křivky.
- křivka nemusí procházet krajními body řídícího polygonu
- affinní transformace celé křivky bude mít stejný výsledek jako transformace každého jejího bodu zvlášť
- křivka leží v konvexní obálce řídícího polynomu

Polynomiální křivky

Vzniknou proložením řídících bodů polynomem obecně  $n$ -tého, v praxi však 3. stupně. V případě že je třeba křivku o více než jedné vlnovce, řeší se to pak interpolací po částech, kdy výslednou křivku poskládáme s více polynomů třetího řádu. Podle toho zda křivka body prochází, nebo se jím jen blíží, můžeme rozdělit polynomiální křivky na:

- **Interpolační polynom** - prochází zadanými body. Interpolační křivku vypočteme tak, že zadané body dosadíme do obecných rovnic polynomů, vznikne soustava rovnic, kterou vypočteme konstanty a z obecné rovnice máme konkrétní.  
I když se omezíme na polynom 3 stupně, může se výsledná funkce rozkmitat. To řeší **Hermitův polynom**, který kromě samotných řídících bodů udává i derivaci v tomto body (směrnici tečny).
- **Aproximační polynom** - se přibližuje k daným bodům. Někdy totiž není možné body proložit funkcí, neboť body jsou nad sebou. V takovém případě chceme aby se funkce co nejvíce blížila daným bodům. K tomu se využívá **metoda nejmenších čtverců**

Typy křivek:

### Fergusonova křivka

- James C. Ferguson – rok 1964 generování křivek řízené dvěma body a vektory v nich (pracoval v The Boeing Company).
- Hermitovou interpolaci aplikovanou na složky vektorového vyjádření křivky pro jednotkovou změnu parametru na obloucích získáme tzv. Fergusonovy křivky.

$$\text{Ve vztahu obecně: } \mathbf{R}(t) = F_0(t) \cdot \mathbf{G} + F_1(t) \cdot \mathbf{H} + F_2(t) \cdot \mathbf{g} + F_3(t) \cdot \mathbf{h}$$

pro  $t \in <0, 1>$ .

$\mathbf{G}, \mathbf{H}$  ... polohové vektory bodů;

$\mathbf{g}, \mathbf{h}$  ... vektory tečen v bodech, ve kterých je Fergusonova křivka jednoznačně určena

Pro  $F_i, i = 0, 1, 2, 3$  platí:  $F_0(t) = 2t^3 - 3t^2 + 1$

$$F_1(t) = -2t^3 + 3t^2$$

$$F_2(t) = t^3 - 2t^2 + t$$

$$F_3(t) = t^3 - t^2$$

$$t \in <0, 1>.$$

Pro  $t = 0 \dots \mathbf{R}(t) = \mathbf{G}$ ; pro  $t = 1 \dots \mathbf{R}(t) = \mathbf{H}$

- Pro  $\mathbf{g} = \mathbf{h}$  je výsledek approximován úsečkou

$$\mathbf{R}(t) = F_0(t) \cdot \mathbf{G} + F_1(t) \cdot \mathbf{H} + F_2(t) \cdot \mathbf{g} + F_3(t) \cdot \mathbf{h} \quad \text{pro } t \in <0, 1>.$$

$\mathbf{G}, \mathbf{H}$  ... polohové vektory bodů;

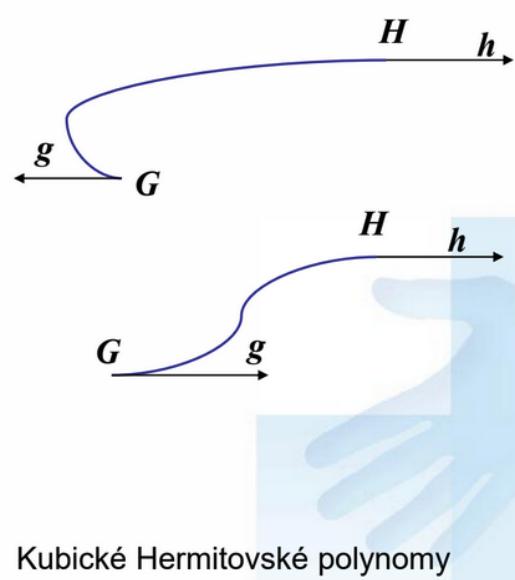
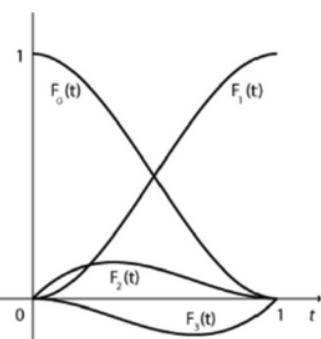
$\mathbf{g}, \mathbf{h}$  ... vektory tečen v bodech G a H

$$F_0(t) = 2t^3 - 3t^2 + 1$$

$$F_1(t) = -2t^3 + 3t^2$$

$$F_2(t) = t^3 - 2t^2 + t$$

$$F_3(t) = t^3 - t^2$$



Kubické Hermitovské polynomy

- Výsledný tvar Fergusonovy kubiky lze ovlivnit třemi způsoby:

- polohou řídících bodů V0 a V1
- směrem tečných vektorů v0 a v1

- velikostí tečných vektorů  $v_0$  a  $v_1$
- Velikost vektorů  $v_0$  a  $v_1$  významně ovlivňuje výsledný tvar křivky. Čím je délka tečných vektorů větší, tím více se křivka přimyká k příslušnému tečnému vektoru.

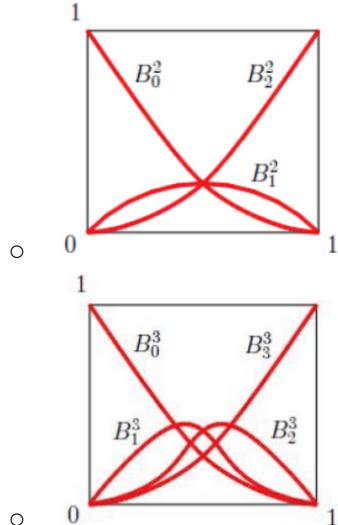


### Beziérova křivka

- Pierre Étienne Bézier (1910 - 1999) Renault - vymyslel pan Bezier, který pracoval v Renaultu, pro modelování aut
  - Beziérovu křivku vymysleli nezávisle na sobě dva francouzští konstruktéři aut. Beziér z Renaultu ji analyticky popsal v roce 1933 a o mnoho let později ji přes geometrickou konstrukci znova objevil Casteljau matematik Citroenu.
- máme zadáno  $n$  řídících bodů  $P$  ( $n \geq 1$ )

$$P(t) = \sum_{i=0}^n P_i B_i^n(t), \text{ kde } t \in \langle 0,1 \rangle$$

- $B$  - pro výpočet bázových funkcí se využívá Bernsteinových polynomů
  - součet bernsteinových polynomů n tého stupně je vždy 1



$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}, \text{ kde } \binom{n}{0} = 1 \text{ a } 0^0 = 1$$

-Němec říkal že vzorec bernstein není nutno umět

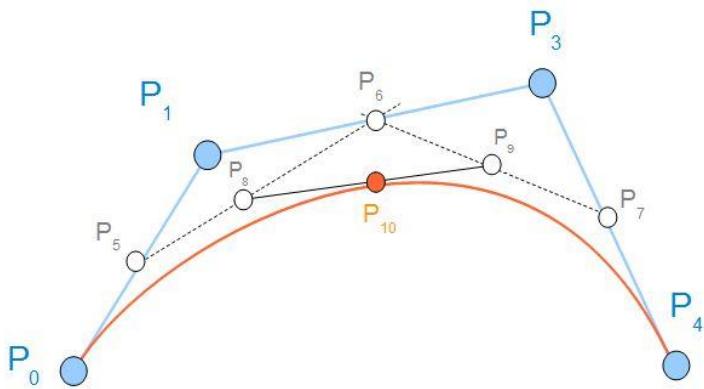
- křivka jde od prvního bodu do posledního, prvním a posledním bodem prochází, k ostatním bodům se blíží tj. Bézierova křivka prochází počátečním a koncovým bodem řídícího polygonu.
- parametr  $t$  určuje pozici na křivce je v rozsahu 0 - 1

- V praxi se používá hlavně **Beziréová kubika** zadána čtyřmi body  $P_0 - P_3$  a při více řídících bodech se počítá po částech. Rovnice Bezierovy kubiky má tvar:

- $P(t) = P_0B_0^3(t) + P_1B_1^3(t) + P_2B_2^3(t) + P_3B_3^3(t)$ ,
- kde  $B$ :
- $B_0^3(t) = (1 - t)^3$
- $B_1^3(t) = 3t(1 - t)^2$
- $B_2^3(t) = 3t^2(1 - t)$
- $B_3^3(t) = t^3$

### Konstrukce Bezierovy křivky

Pro geometrickou konstrukci Beziérové křivky zvolíme poměr  $t$  v kterém dělíme lomenou řídící čáru, jak je vidět na obrázku vpravo, kde je  $t=0,5$ .



Takto jsme vykreslili první bod křivky  $P_{10}$ . Konstrukcí bodu  $P_{10}$  jsme získali nové řídící body, které použijeme pro získání dalších bodů křivky. Další dva body křivky tedy získáme stejným způsobem za použití řídících bodů  $\{P_0, P_5, P_8, P_{10}\}$  a  $\{P_{10}, P_9, P_7, P_4\}$ . Tímto rekurzivním způsobem postupně vykreslíme celou křivku.

Výhoda této konstrukce je, že můžeme ovlivnit hustotu vykreslování dle potřeby. Například v oblasti velkého zakřivení.

Tj. v tomto případě mám bezierovu kubiku (tj. 4 řídící body) parametr  $t$  určuje v jakém poměru budu rozdělovat úsečky mezi body a do nich umisťovat nové body. Umístím body na úsečky v zadaném poměru (o jeden méně), vzniknou nové úsečky (o jednu méně) a takto opakuji dokud nebudu mít 1 finální bod, který je bodem beziérové křivky se zadaným parametrem.

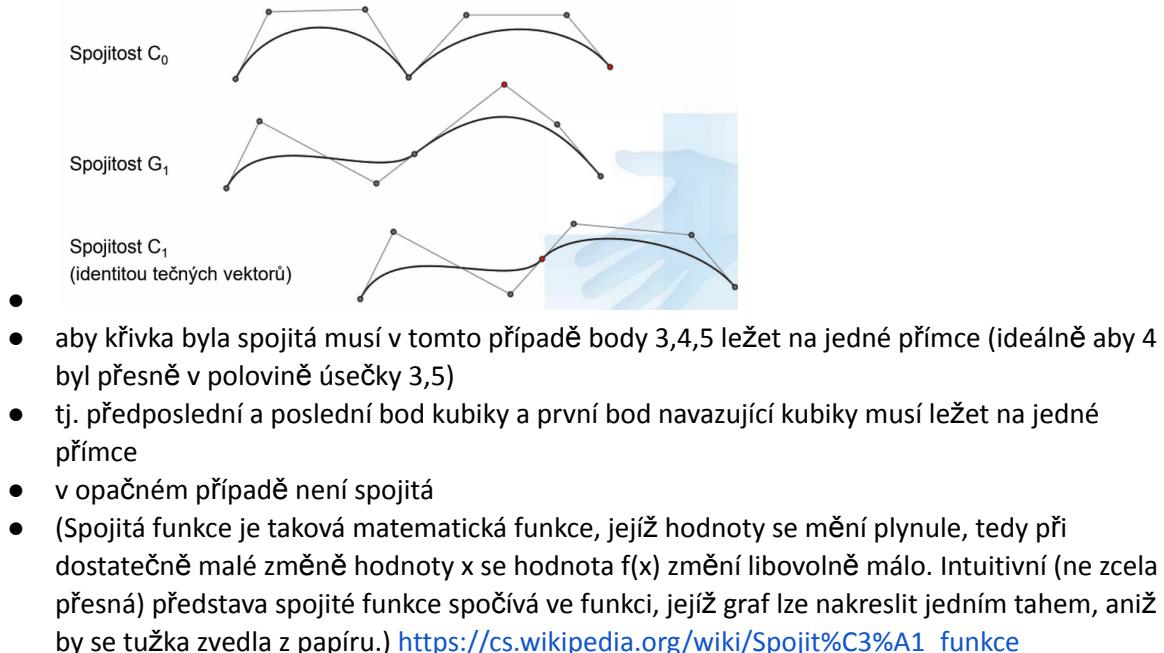
animace [https://homel.vsb.cz/~vod03/vyuka/MAMA/prednasky/prednaska05\\_animace.html](https://homel.vsb.cz/~vod03/vyuka/MAMA/prednasky/prednaska05_animace.html)

Zvyšování stupně beziérových křivek vede ke zvýšení složitosti výpočtu polynomů, proto se zavedl **Bézierův spline**.

- Beziérův spline je spline křivkou, u které se pro výpočet jejich jednotlivých částí používá Bezierova křivka (lineární Bézierova spline křivka, kvadratická Bézierova spline křivka, kubická

Bézierova spline křivka, apod.).

- Je lepší mít křivky menšího stupně, které na sebe navazují - vykreslují se po obloucích
- (v programu lze realizovat mám více řídících bodů a podle hodnoty parametrů vybírám, které se použijí tj 0-1 1-2 2-3 atd. a hodnotu t omezím na rozsah 0-1)
- Takto vzniklé křivky mohou být nespojité



- aby křivka byla spojité musí v tomto případě body 3,4,5 ležet na jedné přímce (ideálně aby 4 byl přesně v polovině úsečky 3,5)
- tj. předposlední a poslední bod kubiky a první bod navazující kubiky musí ležet na jedné přímce
- v opačném případě není spojité
- (Spojité funkce je taková matematická funkce, jejíž hodnoty se mění plynule, tedy při dostatečně malé změně hodnoty x se hodnota  $f(x)$  změní libovolně málo. Intuitivní (ne zcela přesná) představa spojité funkce spočívá ve funkci, jejíž graf lze nakreslit jedním tahem, aniž by se tužka zvedla z papíru.) [https://cs.wikipedia.org/wiki/Spojite%C3%A1\\_funkce](https://cs.wikipedia.org/wiki/Spojite%C3%A1_funkce)

pomocí neracionálního beziera nelze přesně popsat kružnici!! můžeme udělat křivku podobnou

kružnici ale není to kružnice

rationální bezier křivka dovoluje bodum nastavovat vahy a potom lze vykreslit - info navíc

**Příklad otázky:** Popište možnosti reprezentace těles a způsob jejich vykreslení pomocí standardního zobrazovacího řetězce v kontextu grafického rozhraní OpenGL.

Trojúhelník je minimální jednotkou polygonu. Tento tvar pomáhá udržovat počet výpočtů potřebných pro každý detail na co nejnižší úrovni.

## B. Předmět Teoretické základy informatiky

### Algoritmy (ALG I, ALG II, ZSU, UTI)

info v algoritmy 2 - přednášky dvorský - All Slides

- **Strategie algoritmického řešení problémů (strategie řešení hrubou silou, úplným prohledáváním, sníž a vyřeš, rozděl a panuj, transformuj a vyřeš, záměna paměťové a časové složitosti, dynamické programování, hladové algoritmy).**

strategie řešení hrubou silou

- Strategie řešení problémů hrubou silou je založena na přímočarém přístupu k řešení problému, kdy algoritmus řešení vychází přímo ze zadání problému a pojmu obsažených v zadání.
  - je způsob řešení problému či úlohy, při kterém se systematicky prochází celý prostor možných řešení problému.
  - Jeho výhodou je nalezení opravdu nejlepšího řešení či případný důkaz o nemožnosti řešení problému.
  - Jeho nevýhodou bývá velká složitost hledání, a tedy časová, případně paměťová náročnost algoritmu.
  - Velmi často čas potřebný k nalezení řešení roste exponenciálně či dokonce s faktoriálem, takže i pro velmi malé prostory možných řešení je tato metoda v praxi nepoužitelná.
- 
- obecná strategie – je obtížné najít problém, kde by „nezabrala“,
  - obecně sice nevede k efektivním algoritmům, ale pro některé problémy, např. násobení matic, pattern matching, jsou algoritmy založené na této strategii použitelné i pro větší vstupy,
  - přijatelná strategie v případě, kdy se nevyplatí zabývat se sofistikovanějším algoritmem – ad hoc řešení problému,
  - vždy použitelná strategie pro řešení problémů s malou velikostí vstupu a význam i jako měřítko, kterým můžeme poměrovat efektivnější algoritmy řešící stejný problém.
  - např SelectSort - setřízení pole o velikosti n - n průchodů a pokaždé se najde další nejmenší hodnota a umístí se na i tou pozici

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 89 | 45 | 68 | 90 | 29 | 34 | 17 |
| 17 | 45 | 68 | 90 | 29 | 34 | 89 |
| 17 | 29 | 38 | 90 | 45 | 34 | 89 |
| 17 | 29 | 34 | 90 | 45 | 68 | 89 |
| 17 | 29 | 34 | 45 | 90 | 68 | 89 |
| 17 | 29 | 34 | 45 | 68 | 90 | 89 |
| ○  | 17 | 29 | 34 | 45 | 68 | 89 |
|    |    |    |    |    |    | 90 |

```

1 for i ← 0 to n – 2 do
2 min ← i;
3 for j ← i + 1 to n – 1 do
4 if A[j] < A[min] then
5 min ← j;
6 end
7 end
8 Swap (A[i], A[min]);
9 end

```

- bubble sort - záměna sousedních hodnot dokud nejsou ve správném pořadí, složitost  $O(n^2)$

|    |   |    |    |    |    |    |    |    |    |    |
|----|---|----|----|----|----|----|----|----|----|----|
| 89 | ↔ | 45 | 68 | 90 | 29 | 34 | 17 |    |    |    |
| 45 |   | 89 | ↔  | 68 | 90 | 29 | 34 | 17 |    |    |
| 45 |   | 68 | 89 | ↔  | 90 | 29 | 34 | 17 |    |    |
| 45 |   | 68 | 89 | 90 | ↔  | 29 | 34 | 17 |    |    |
| 45 |   | 68 | 89 | 29 | 90 | ↔  | 34 | 17 |    |    |
| 45 |   | 68 | 89 | 29 | 34 | 90 | ↔  | 17 |    |    |
| 45 |   | 68 | 89 | 29 | 34 | 17 | 90 |    |    |    |
| 45 | ↔ | 68 | ↔  | 89 | ↔  | 29 | 34 | 17 | 90 |    |
| 45 |   | 68 |    | 29 |    | 89 | ↔  | 34 | 17 | 90 |
| 45 |   | 68 |    | 29 |    | 34 | 89 | ↔  | 17 | 90 |
| 45 |   | 68 |    | 29 |    | 34 | 17 | 89 | 90 |    |

```

1 for i ← 0 to n – 2 do
2 for j ← 0 to n – i – 2 do
3 if A[j] > A[j + 1] then
4 Swap (A[j], A[j + 1]);
5 end
6 end
7 end

```

- vyhledávání, a další
- hrubou silou lze řešit cokoliv, ale často to je neefektivní

úplným prohledáváním - levitin str 115

- speciální případ strategie hrubou silou
- mnoho problémů vyžaduje nalezení prvku speciálních vlastností v doméně
- typicky se jedná o problémy, které zahrnují kombinatorické problémy - permutace, kombinace a podmnožiny
- Vyčerpávající vyhledávání je jednoduše hrubý přístup ke kombinatorickým problémům. Navrhuje vygenerovat každý prvek problémové domény, vybrat z nich ty, které splňují všechna omezení, a pak najít požadovanou prvek (např. takový, který optimalizuje nějakou objektivní funkci).
- najde mnoho řešení a vybere z nich to nejlepší/to které splňuje všechny podmínky
- Ačkoli je myšlenka poměrně přímočará, implementace vyžaduje algoritmus pro generování kombinatorických objektů
- Typické problémy: obchodní cestující, knapsack, assignment problem
  - nalezení nejkratší trasy mezi všemi zadanými body
  - přidělení hodnot (váh) objektům a výběr objektů aby součet hodnot byl nižší než zadaná hodnota, název dle batohu a předměty s váhou
  - optimalizace přiřazení úkolů agentům - každý agent má nějakou cenu vykonání určitého úkolu a každý může mít pouze jeden úkol
- u těchto algoritmů je problém nalézt časově efektivní algoritmus, musí se prozkoumat všechny možnosti a vybrat ta nejlepší

sniž a vyřeš <https://www.geeksforgeeks.org/decrease-and-conquer/>

- založeno na využití vztahu mezi řešením problému a řešením jeho menší části
- jakmile je takový vztah vytvořen lze jej využít shora dolů nebo zdola nahoru
- shora dolů vede k rekurzivní implementaci
- zdola nahoru je obvykle implementována iterativně, přičemž se začíná řešením nejmenší části problému
- 3 hlavní varianty
  - snižování o konstantu
    - V této variantě se velikost instance při každé iteraci algoritmu zmenšuje o stejnou konstantu. Obvykle je tato konstanta rovna jedné
    - např. Insertion sort
 

```
void insertionSort(int arr[], int n)
{
 int i, key, j;
 for (i = 1; i < n; i++){
 key = arr[i];
 j = i - 1;
 while (j >= 0 && arr[j] > key)
 {
 arr[j + 1] = arr[j];
 j = j - 1;
 }
 arr[j + 1] = key;
 }
}
```

      - označení klíče, přesouvá se směrem dopředu dokud je jeho levý soused větší než klíč (vzestupně)
    - DFS - prohledávání do hloubky, navštíví nejdříve prvního souseda přezkoumávaného vrcholu pokud jej ještě nenavštívil a až potom se vrací zpět a navštěvuje další vrcholy
    - BFS - prohledávání do šířky, navštíví nejdříve všechny sousedy přezkoumávaného vrcholu a poté jede do prvního navštíveného
  - snižování o konstantní násobek
    - Tato technika navrhuje zmenšení instance problému o stejný konstantní násobek při každé iteraci algoritmu. Ve většině aplikací je tento konstantní faktor roven dvěma.
    - Algoritmy snižování o konstantní faktor jsou velmi efektivní, (zejména pokud je tento faktor větší než 2)
    - např. Binary search - vyhledávání v setříděném poli
 

```
// pole, Levé ohrazení, pravé ohrazení, hledaná hodnota
int binarySearch(int arr[], int l, int r, int x)
{
 if (r >= l) {
 int mid = l + (r - 1) / 2;

 if (arr[mid] == x)
 return mid;

 // menší bude se prohledávat levá část pole
 if (x < arr[mid])
 return binarySearch(arr, l, mid - 1, x);
 else
 // jinak pravá
 return binarySearch(arr, mid + 1, r, x);
 }
 // nenašel
 return -1;
}
```

      -
    - snížení o proměnnou velikost
      - V této variantě se vzor zmenšování velikosti v jednotlivých iteracích

algoritmu mění.

- Protože v problému hledání největšího společného dělitele dvou čísel je sice hodnota druhého argumentu na pravé straně vždy menší než na straně levé, ale nezmenšuje se ani o konstantu, ani o konstantní faktor.
- např. výpočet mediánu, interpolation search,
- euklidův algoritmus - největší společný dělitel

$$\begin{array}{r} 106 / 16 = 6, \text{ remainder } 10 \\ | \\ 16 / 10 = 1, \text{ remainder } 6 \\ | \\ 10 / 6 = 1, \text{ remainder } 4 \\ | \\ 6 / 4 = 1, \text{ remainder } 2 \\ | \\ 4 / 2 = 2, \text{ remainder } 0 \\ | \\ \text{GCD} \end{array}$$

- ```
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}
```

- Princip snížení nebo redukce instance problému na menší instanci téhož problému. Problém bude vyřešen vyřešením menšího problému. Rozšířením menší instance získáme řešení původního problému
- Základní myšlenka techniky zmenšování a dobývání je založena na využití vztahu mezi řešením dané instance problému a řešením její menší instance. Tento přístup je také znám jako přírůstkový nebo induktivní přístup.

rozděl a panuj

<https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/?ref=gcse>

- Principem je rozdělení problému na několik podproblémů. Ty se rekuzivně vyřeší. Pokud jsou dílčí problémy dostatečně malé lze je řešit jednoduše. Nakonec se spojí řešení dílčích problémů do řešení původního problému
- většinou $O(n \log n)$
- quicksort
 - vybírá se pivot, podle něhož se pole rozdělí na 2 části menší a větší než pivot

```
procedure quicksort(List values)
if values.size <= 1 then
    return values

pivot = náhodný prvek z values

Rozděl seznam values do 3 seznamů
seznam1 = { prvky menší než pivot }
seznam2 = { pivot }
seznam3 = { prvky větší než pivot }

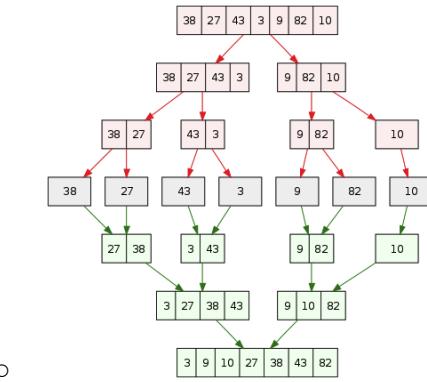
return quicksort(seznam1) + seznam2 + quicksort(seznam3) - pseudokód (vzestupně)
```
 - pokud je prvek menší než pivot zvýší se pomocný index a zamění se prvek s prvkem na pomocném indexu
 - nakonec se prohodí pivot s prvkem na pomocném indexu + 1 - tím se pole rozdělí na 2 části menší než pivot a větší než pivot
 - rekuzivně se 2 krát volá na části před pivotem a za pivotem, opakuje se dokud není velikost části 1
 - <https://www.youtube.com/watch?v=PgBzjCcFvc>

```

void quicksort(int array[], int left, int right){
    if(left < right){
        int boundary = left;
        // pokud je na indexu i menší než pivot tj. left
        // tak se prohodi s prvkem na pomocnem indexu boundary
        for(int i = left + 1; i < right; i++){
            if(array[i] < array[left]){
                swap(array[i], array[boundary]);
            }
        }
        swap(array[left], array[boundary]);
        // mam rozdeleno na menší než pivot a vetsi
        quicksort(array, left, boundary);
        quicksort(array, pivot + 1, right);
    }
}

```

- merge sort - <https://www.geeksforgeeks.org/divide-and-conquer/>
<https://www.geeksforgeeks.org/merge-sort/>

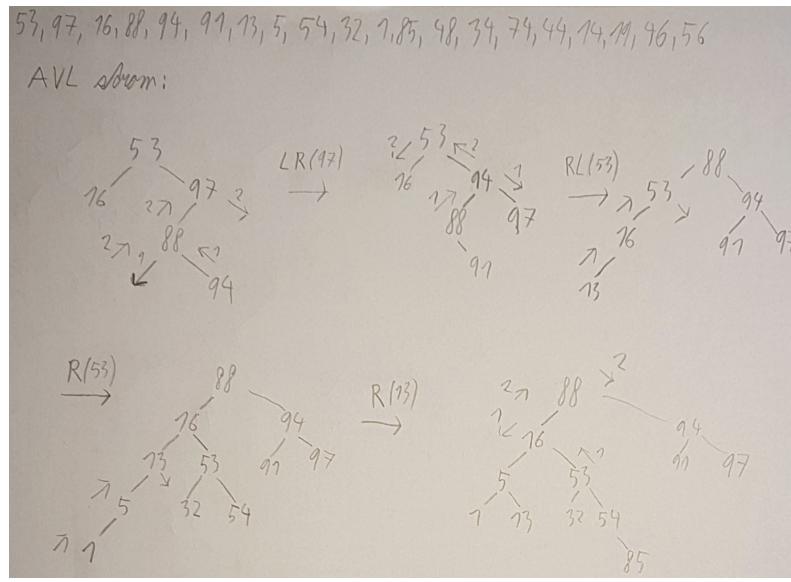


- rozdělováno na půlky dokud není pouze 1 rekurzivním voláním s předáním poloviny pole
- poté se spojuje a sestavuje se pole porovnáváním prvků v levé a pravé větvi dokud jsou ve větvích prvky a dříve se vloží menší prvek
 - nejdříve levá větev a potom pravá - dle volání rekurze
- nejbližší páry bodů a další

Transformuj a vyřeš levitin str 202

- upravení problému dat tak aby byl vhodný k řešení a až potom se řeší
- 3 varianty
 - Transformace na jednodušší nebo vhodnější instanci téhož problému
 - Transformace na jinou reprezentaci téže instance
 - Transformace na instanci jiného problému, pro který je algoritmus k dispozici
- předtřízení
 - myšlenka spočívá v tom že na setřízeném poli/prvcích je mnoho algoritmů pro vyhledávání efektivnější
 - například nalezení prvku (např půlením intervalu a určování ve které půlce se bude hledat dle hodnoty středu jestli menší větší), nalezení stejných hodnot (jsou vedle sebe)
- Gaussova eliminační metoda
 - řešení soustavy lineárních rovnic
 - LU-rozklad matic
 - Inverzní matic
 - Determinant matic
- Vyhledávací stromy

- vyvažování stromů tak aby byly větve podobně dlouhé - usnadňuje vyhledávání
- je nutné vyvažovat při sestavování vyhledávacích stromů aby v nich šlo efektivně vyhledávat
- AVL stromy - rotace vrcholů



Záměna paměťové a časové složitosti levitin str 42

- časová složitost - jak rychle algoritmus poběží
 - paměťová složitost - kolik paměti bude algoritmus potřebovat
 - dříve byly oba zdroje čas i paměť omezené
 - dnes je paměti dostatek, proto není problém když algoritmus potřebuje více paměti
 - složitost se určuje k velikosti vstupu tj. N a podle složitosti se určují třídy složitosti O
 - v některých případech lze zrychlit vykonávání algoritmu na úkor množství potřebné paměti - tj. snížení časové složitosti na úkor paměťové (nebo i opačně)
- <https://cs.khanacademy.org/computing/computer-science/cryptography/comp-number-theory/v/time-space-tradeoff>

dynamické programování

- Dynamické programování je metoda pro efektivní řešení určitých optimalizačních úloh.
- Lze jej použít pro řešení úloh, které lze rozdělit na podúlohy, jejichž optimální řešení lze použít při řešení původní úlohy.
- Princip dynamického programování spočívá v rekurzivním dělení úlohy na menší části, které se řeší ve vhodném pořadí, jejich výsledky se zaznamenávají a jsou použity pro řešení složitějších podúloh včetně původní úlohy.
- Dynamické programování je technika pro řešení problémů s překrývajícími se dílčími problémy. Tyto podproblémy obvykle vznikají rekurencí vztahující se k řešení daného problému k řešení jeho menších podproblémů. Namísto opakovaného řešení překrývajících se podproblémů se při dynamickém programování navrhoje řešit každý z menších podproblémů pouze jednou a výsledky zaznamenat do tabulky, z níž lze poté získat řešení původního problému.
- fibonacciiho posloupnost

- ```
function fib(n)
 if n == 0 return 0
 if n == 1 return 1
 return fib(n - 1) + fib(n - 2)
```

## hladové algoritmy

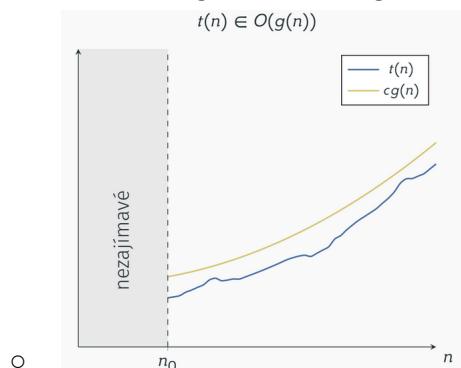
- je jakýkoliv algoritmus, který se při řešení problému řídí heuristikou, že v každé fázi provede lokálně optimální volbu. V mnoha problémech hladová strategie nevede k optimálnímu řešení, ale může přinést lokálně optimální řešení, které se blíží globálně optimálnímu řešení v rozumném čase.
- např. pro problém obchodního cestujícího (který má vysokou výpočetní složitost) je následující heuristika: "V každém kroku cesty navštívte nejbližší nenavštívené město". Tato heuristika nemá v úmyslu najít nejlepší řešení, ale končí v rozumném počtu kroků; nalezení optimálního řešení takto složitého problému obvykle vyžaduje nepřiměřeně mnoho kroků.
- nesnaží se hledat nutně správné řešení ale snaží se rychle najít to nevhodnější
- najde globálně optimální řešení
- lze vhodně využít pro - cestujícího, batoh, hledání kostry grafu, ...

## • Složitost algoritmů (asymptotická notace, korektnost algoritmů a její analýza).

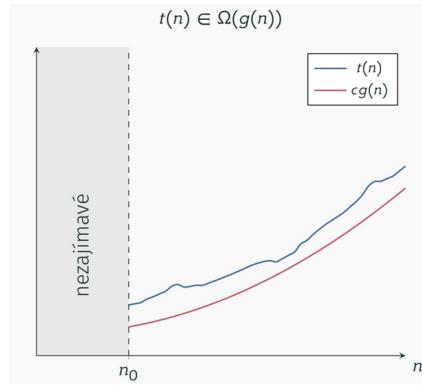
Výpočet efektivity algoritmů. Závisí na velikosti vstupu. Může se řešit časová a paměťová složitost.

Asymptotická notace dvorsky str 220/587

- Asymptotická složitost určuje časovou náročnost algoritmu na základě počtu vstupních dat
- O-notace
  - Říkáme, že funkce  $t(n)$  patří do  $O(g(n))$ , jestliže existuje kladná nenulová reálná konstanta  $c$  a přirozené číslo  $n_0 > 1$  takové, že  $t(n) \leq cg(n)$  pro všechna  $n \geq n_0$
  - tzv. od určitého  $n$  je hodnota funkce náročnosti  $t$  pro všechna  $n > n_0$  menší rovná než funkce  $g$ , náročnost algoritmu je menší než  $g(n)$  - náročnost alg roste pomaleji

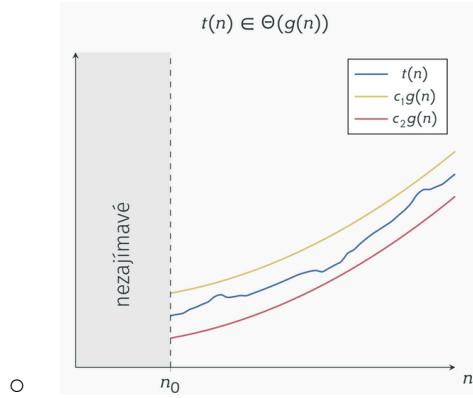


- Formálně jsou definičním oborem i oborem hodnot funkcí  $t(n)$  i  $g(n)$  přirozená čísla  $\Rightarrow$  graf by měl být složen pouze z bodů, nikoliv křivek.
- $\Omega$ -notace(Omega)
  - Říkáme, že funkce  $t(n)$  patří do  $\Omega(g(n))$ , jestliže existuje kladná nenulová reálná konstanta  $c$  a přirozené číslo  $n_0 > 1$  takové, že  $t(n) \geq cg(n)$  pro všechna  $n \geq n_0$
  - náročnost algoritmu je větší než  $g(n)$  - náročnost alg roste rychleji



- **Θ-notace (theta)**

- Říkáme, že funkce  $t(n)$  patří do  $\Theta(g(n))$ , jestliže existují kladné nenulové reálné konstanty  $c_1, c_2$  a přirozené číslo  $n_0 > 1$  takové, že  $c_1g(n) \leq t(n) \leq c_2g(n)$  pro všechna  $n \geq n_0$ .
- náročnost algoritmu roste podobně rychle - tj. je v rozmezí mezi konstantami



- theta notace je průnik O a omega notace

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

Lower bound      avg bound      upper bound

- Základní vlastnosti:

- 1.  $f(n) \in O(f(n))$
- 2.  $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- 3.  $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$
- tranzitivita, srovnej s  $a \leq b \wedge b \leq c \Rightarrow a \leq c$

### Využití limit k výpočtům

Rychlosť růstu funkcí lze snadněji počítat pomocí limit:

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ roste pomaleji než } g(n) \\ c & t(n) \text{ roste stejně rychle jako } g(n) \\ \infty & t(n) \text{ roste rychleji než } g(n) \end{cases}$$

Je zřejmé, že:

$$t(n) \in O(g(n)) \Leftrightarrow t(n) \text{ roste pomaleji nebo stejně rychle než } g(n)$$

$$t(n) \in \Omega(g(n)) \Leftrightarrow t(n) \text{ roste stejně rychle nebo rychleji než } g(n)$$

$$t(n) \in \Theta(g(n)) \Leftrightarrow t(n) \text{ roste stejně rychle jako } g(n)$$

- Přestože teoreticky existuje nekonečně mnoho tříd složitosti, složitost většiny algoritmů padne do několika málo tříd.

| Třída      | Jméno          | Poznámka                                                                                         |
|------------|----------------|--------------------------------------------------------------------------------------------------|
| 1          | konstantní     | složitost nezávisí na velikosti vstupu; jen velmi málo algoritmů                                 |
| $\log n$   | logaritmická   | typicky algoritmy redukující velikost vstupu konstantním faktorem; vyhledávání půlením intervalu |
| $n$        | lineární       | algoritmy zpracovávající seznam o $n$ prvcích; např. sekvenční vyhledávání                       |
| $n \log n$ | „ $n \log n$ “ | „rozděl a panuj“ algoritmy; průměrné složitosti QuickSortu, MergeSortu                           |

- 

| Třída | Jméno         | Poznámka                                                                                               |
|-------|---------------|--------------------------------------------------------------------------------------------------------|
| $n^2$ | kvadratická   | obecně algoritmy se dvěma vnořenými cykly; elementární metody třídění, sčítání matic typu $n \times n$ |
| $n^3$ | kubická       | obecně algoritmy se třemi vnořenými cykly; násobení matic typu $n \times n$                            |
| $2^n$ | exponenciální | typicky generování všech podmnožin $n$ prvkové množiny                                                 |
| $n!$  | faktoriál     | typicky generování všech permutací $n$ prvkové množiny                                                 |

- 

Korektnost algoritmů a její analýza dvorsky 95/857 levitin str 13

u algoritmů se zkoumá

Správnost (korektnost)

- Algoritmus považujeme za správný, pokud pro každý správný vstup poskytne v konečném čase správný výsledek. Pro nesprávný vstup není chování správného algoritmu definováno.
- Obvyklou metodou důkazu je matematická indukce.
- Pro korektní důkaz správnosti algoritmu nestačí prokázat správnost pro některou instanci problému, správnost musíme umět prokázat pro všechny instance problému, a naopak
- jako důkaz nesprávnosti algoritmu stačí najít jedinou instanci problému, abychom mohli algoritmus prohlásit za chybný.
- běžná technika pro ověření správnosti je matematická indukce, protože iterace algoritmu poskytuje posloupnost kroků potřebných k důkazu
- případně to že je algoritmus nesprávný lze dokázat i na jednom vstupu při kterém je výstup nesprávný a algoritmus selže

Časová složitost

- jak rychle algoritmus pracuje
- rychlosť neměříme časovými jednotkami, ale množstvím provedených instrukcí algoritmu

#### Prostorová složitost

- jak mnoho paměti potřebuje
- měříme v bytech a násobcích

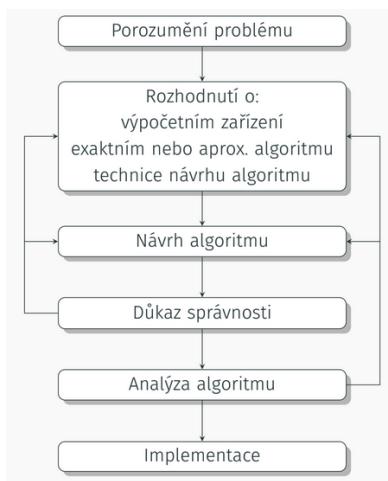
#### Jednoduchost

- nelze exaktně definovat
- subjektivní záležitost - náročnost pochopení, složitost

#### Obecnost

- obecnost navrženého algoritmického řešení – řešit problém velice obecně a nebo brát v úvahu možná zjednodušení v konkrétním případě?
  - řešit obecný problém může být jednodušší než konkrétní např. NSD, na stejném úrovni např. hledání mediánu nebo výrazně náročnější např. kvadratická rovnice vs obecná
- obecnost instance problému – návrh algoritmu by měl zpracovat všechny rozumně očekávatelné, přirozené, instance problému.
  - např. návrh řešení kvadratické rovnice i pro komplexní čísla atd.

#### Proces návrhu algoritmu



- **Základní datové struktury (pole, seznam, zásobník, fronta, graf, vyhledávací strom, hašovací tabulka, halda).**

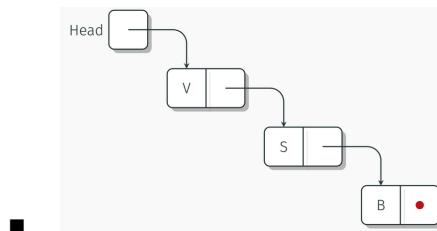
Datovou strukturu můžeme definovat jako způsob organizace vzájemně souvisejících dat.

Jakou použít datovou strukturu silně závisí na řešeném problému.

nejdůležitější datové struktury

- Pole
  - Konečná posloupnost n hodnot uložených ve spojitém úseku paměti.
  - Přístup pomocí indexu, náhodný přístup s konstantní časovou složitostí.
  - index je nezáporné číslo s hodnotami 0 až n-1
  - přímý přístup na pozici v paměti (odkaz na první hodnotu + index tj. posun adresy o velikost položky \* i)
- Seznam

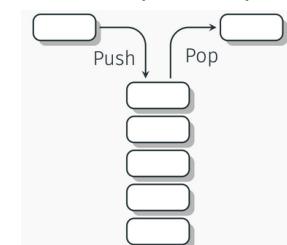
- lineární datová struktura, operace nejsou striktně určeny, existuje mnoho variant.
- v nejjednodušším případě jen nutný odkaz na první prvek seznamu, tzv. hlavu seznamu.
- varianty:
- jednosměrný seznam (singly linked list) – nejjednoduší varianta, odkaz pouze na následníka,
  - složen z položek, které obsahují data a odkaz na další položku
  - každá položka odkazuje na následovníka, lze k nim přistupovat sekvenčně
  - přímý přístup na základě indexu je složitý - je nutný průchod cyklem, pohyb seznamem dozadu je taky problém
  - seznam ukončen prázdným odkazem null



- obousměrný seznam (doubly linked list) – položka obsahuje odkaz na předchůdce i následníka,
  - položky obsahují odkaz na následující i na předchozí prvek seznamu
  - lze snadno procházet i směrem dozadu
- kruhový seznam (circular list) – hlava a ocas seznamu splývají.
  - může být jednosměrný nebo obousměrný
  - jediná změna je odkaz že poslední odkazuje na první a v případě obousměrného i první na poslední

- Zásobník

- princip last-in, first-out tj LIFO
- prvek, který byl vložen poslední, je jako první ze zásobníku vyzvednut
- prvky vkládáme na tzv. vrchol zásobníku (stack pointer).
- prvně vložený prvek se nazývá dno zásobníku (stack bottom)
- základní operace
  - Push – vložení prvků na vrchol zásobníku
  - Pop – vyjmutí prvků z vrcholu zásobníku
  - IsEmpty – test prázdnosti zásobníku
  - Top – vrátí prvek z vrcholu zásobníku bez jeho vyjmutí



- chyby - podtečení (stack underflow) - pop na prázdném zásobníku, přetečení (stack overflow) - není kam uložit další prvek
- využití zásobníku
  - volání funkcí (metod)
  - v hodnocování aritmetických výrazů

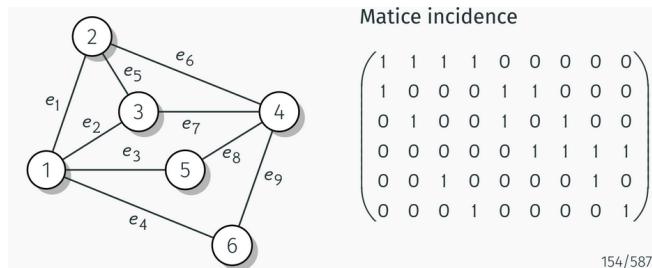
- zásobníkově orientované jazyky, například PostScript, PDF
  - testování parity závorek, HTML/XML značek
- Fronta
  - princip first-in, first-out, FIFO
  - prvek, který byl vložen první, je také jako první z fronty vyzvednut
  - první prvek se nazývá hlava fronty (head),
  - poslední prvek se nazývá ocas fronty (tail).
  - základní operace
    - Enqueue – vložení prvku na konec fronty
    - Dequeue – vyjmutí prvku ze začátku fronty
    - Peek – vrátí prvek ze začátku fronty bez jeho vyjmutí
    - IsEmpty – test zda je fronta prázdná
    - operace u fronty i zásobníku mají časovou složitost  $O(1)$  - časová složitost nezávisí na počtu uložených prvků
  - chybové stavy podtečení a přetečení
  - využití
    - tisková fronta u sdílené tiskárny
    - plánovač v operačním systému (více běžících procesů na jednoprocесорovém počítači  $\Rightarrow$  procesy se musí střídat)
    - obsluha uživatelů na serverech obecně
- Prioritní fronta
  - řešení úlohy „Vymí z množiny největší prvek a zpracuj ho.“
  - na rozdíl od obyčejné fronty se k prvkům váže ještě prioritita,
  - prvek s vyšší prioritou předbíhá ty s nižší prioritou a odchází z fronty dříve
  - implementace
    - pomocí pole nebo setříděného pole,
    - efektivněji pomocí datové struktury zvané halda (heap).

## Graf

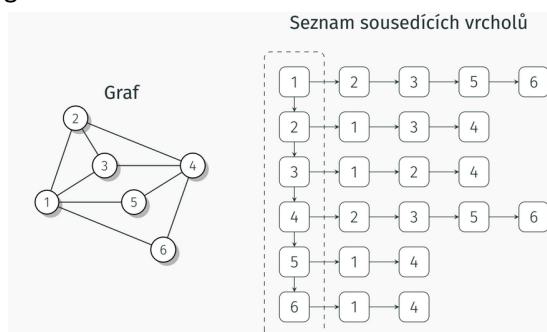
- Neorientovaný graf
  - Neorientovaným grafem nazýváme dvojici  $G = (V, E)$ , kde  $V$  je konečná neprázdná množina vrcholů,  $E$  je množina jednoprvkových nebo dvouprvkových podmnožin  $V$ . Prvky množiny  $E$  se nazývají hrany grafu.
  - Příklad
    - $G = (V, E)$
    - $V = \{1, 2, 3, 4, 5, 6\}$
    - $E = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$
  - O hraně e říkáme, že spojuje vrcholy u a v.
  - Vrcholům u a v říkáme krajní vrcholy hrany e.
  - vrcholy u a v jsou incidentní (nebo, že incidují) s hranou e. A obdobně, že hrana e je incidentní s vrcholy u a v.
  - Protože hrana e spojuje vrcholy u a v říkáme, o nich že jsou to sousední (sousedící) vrcholy.
  - Hranu spojující vrchol se sebou samým nazýváme smyčkou.
  - Stupněm vrcholu v neorientovaném grafu nazýváme počet hran s vrcholem incidentních, tj. počet hran vrcholu
  - Součet stupňů vrcholů libovolného neorientovaného grafu  $G = (V, E)$  je roven

dvojnásobku počtu jeho hran.

- úplný graf je ten který má každý vrchol propojen s každým dalším tj. každý vrchol má  $n - 1$  hran, kde  $n$  je celkový počet vrcholů.
- Orientovaný graf
  - hrany mají určen směr, který je závislý na pořadí hran v  $E$
- Reprezentace grafu
  - maticí incidence
    - Počet řádků matice odpovídá počtu vrcholů, počet sloupců odpovídá počtu hran.
    - Pokud je vrchol incidentní s hranou, je na dané pozici jednička, jinak nula.



- maticí sousednosti
  - Čtvercová matice, kde řád matice odpovídá počtu vrcholů v grafu.
  - Pokud jsou dva vrcholy spojeny hranou, je na dané pozici jednička, jinak nula.
- Matice sousednosti
 
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$
- seznam sousedních vrcholů
  - pro každý vrchol seznam vrcholů se kterými je spojen hranou
  - vhodné pro řídké grafy (s malým počtem hran), snadnější změny struktury grafu - změna vrcholů a hran



- maticová reprezentace vhodnější pro husté grafy ale vložení a mazání vrcholů je komplikované
- Vážené grafy

- každé hraně je přiřazeno číslo - váha/cena hrany
  - může symbolizovat - délka cesty, kapacita datové linky
  - mohou být orientované i neorientované
- sled - posloupnost vrcholů v grafu
- cesta - sled v němž se neopakuje žádný vrchol
- uzavřený sled - sled který má alespoň jednu hranu a počáteční a koncový vrchol splývají
- cyklus = uzavřená cesta - je uzavřený sled v němž se neopakují vrcholy ani hrany
- graf je souvislý pokud mezi každými 2 vrcholy existuje cesta
  
- volný strom
  - Souvislý, acyklický, neorientovaný graf
  - strom má  $n-1$  hran a mezi 2 vrcholy existuje právě jedna cesta
  - tj. neobsahuje cykly
- les
  - Acyklický graf, který není spojitý
  - tj. více stromů
- ve stromech se určuje hloubka vrcholu a výška stromu
  - Délka cesty od kořene stromu k vrcholu  $x$  se nazývá hloubka vrcholu  $x$  ve stromu  $T$ .
  - Největší hloubka libovolného vrcholu se nazývá výška stromu  $T$ .
- seřazený strom - je v něm určeno pořadí potomků

#### binární strom

- uzel má levého a pravého potomka (nebo kořen má levý a pravý podstrom), lze je využít pro vyhledávání (na základě podmínky s kořenem se určí jestli bude prohledávat pravý nebo levý podstrom)
- vyhledávací pravidlo Nechť  $y$  je uzel v binárním stromu. Potom pro každý uzel  $x$  v levém podstromu uzlu  $y$  a každý uzel  $z$  v pravém podstromu uzlu  $y$  platí  $x.key < y.key < z.key$ .
- Binární strom, ve kterém pro všechny jeho uzly platí toto pravidlo nazýváme **binární vyhledávací strom**
  - Navigační pravidlo tedy určuje, jak mají být data v binárním vyhledávacím stromu rozmištěna.
  - Znalosti rozmištění dat ve stromu využijeme při jejich vyhledávání.
  - Algoritmy pro vložení a vyjmutí ze stromu jsou navázány na algoritmus vyhledávání.
  - Binární vyhledávací strom je tedy už od počátku budován s ohledem na toto pravidlo.
  - Hledání hodnoty a zahájíme v kořeni stromu  $r$ . Potom mohou nastat tyto možnosti:
    - Strom s kořenem  $r$  je prázdný, potom tento strom nemůže obsahovat uzel s klíčem a a hledání končí neúspěchem.
    - V opačném případě srovnáme klíč a s klíčem kořene  $r$ . V případě, že
      - $a = r.key$  strom obsahuje uzel s klíčem a a hledání končí úspěšně;
      - $a < r.key$  všechny uzly s klíči menšími než  $r.key$  jsou levém podstromu, pokračujeme rekurzivně v levém podstromu;
      - $a > r.key$  všechny uzly s klíči většími než  $r.key$  jsou pravém podstromu, pokračujeme rekurzivně v pravém podstromu.
    - Vložení klíče musí korespondovat s vyhledávacím algoritmem.
    - Nejdříve se musíme pokusit vkládaný klíč ve stromu najít.

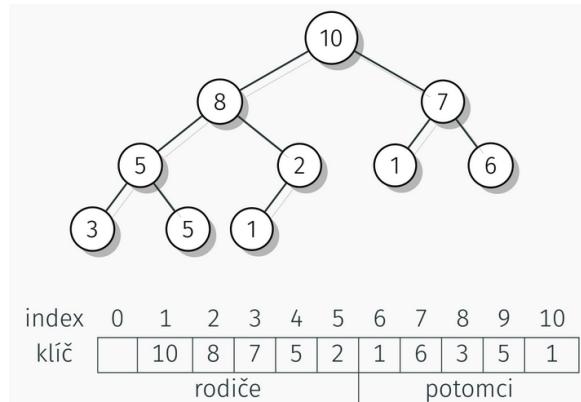
- Pokud jej nenajdeme, tak místo, kde jsme hledání neúspěšně zakončili odpovídá místu ve stromu, kde by tento klíč měl být.

### Hašovací tabulka

- Hašovací tabulka (popřípadě hashovací tabulka nebo hašovací tabulka) je vyhledávací datová struktura, která asocuje hašovací klíče s odpovídajícími hodnotami. Hodnota klíče je spočtena z obsahu položky pomocí nějaké hašovací funkce.
- využití pro rychlé vyhledávání v poli
- klíč hodnoty kterou budeme ukládat do pole předáme hašovací fci která vypočte z index a uloží se na ten index, umožní poté snadné nalezení hodnoty podle klíče
- obecně vznikají kolize, klíčů je více než je velikost tabulky, pro různé klíče může dojít k vypočtení stejné haše. Lze řešit zřetězením záznamů (pole seznamů) nebo otevřená adresace (uložení na nejbližší následující volnou pozici)
- při vyhledávání dle klíče je tedy potřeba testovat jestli je na pozici opravdu uložena hledaná hodnota případně ji nalézt
- hašování je v porovnání s dalšími vyhledávacími metodami např. s vyhledávacími stromy rychlejší, neposkytuje záruku pro nejhorší případ
- pro vyhledání je potřeba znát klíč a neposkytuje intervalové dotazy tj. vyhledání klíčů v intervalu a sekvenční průchod

### Halda

- částečně setříděná datová struktura, zvláště vhodná pro implementaci prioritní fronty.
- Prioritní fronta – datová struktura chápaná jako multimnožina, kde prvky jsou řazeny podle priority a podporující operace
  - nalezení prvku s nejvyšší prioritou,
  - odebrání prvku s nejvyšší prioritou a
  - vložení nového prvku do fronty.
- halda je využívána například v:
  - plánování úloh v OS
  - grafových algoritmech např. Primův, Dijkstrův atd.
  - třídění haldou – HeapSort
- Haldu definujeme jako binární strom s jedním klíčem v každém uzlu, který splňuje následující dvě vlastnosti:
  - kompletnost, tj. všechna patra stromu jsou zaplněna, s výjimkou posledního. V posledním patře může zprava chybět několik listů a
  - rodičovská dominance, tj. klíč v každém uzlu je vždy větší nebo roven než klíče ve všech jeho potomcích. V listech je libovolný klíč vždy brán jako větší než klíče v neexistujících potomcích.
- vlastnosti hald
  - Klíče na každé cestě z kořene do listu tvoří nerostoucí posloupnost. Jinak mezi klíči nejsou žádné vztahy, např. menší klíče v levém podstromu než v pravém atd.
  - Pro n klíčů existuje pouze jeden úplný binární strom. Jeho výška je  $\log_2 n$
  - Největší klíč je vždy v kořeni haldy.
- reprezentace haldy v poli
  - zaznamenávání od kořene zleva doprava po úrovních



- 0 se většinou nevyužívá
- konstrukce haldy
  - zdola nahoru
    - úprava existujícího stromu
    - začíná se vpravo dole a zaměňují se hodnoty potomků a uzel < potomek
    - Konstrukce haldy s n prvky vyžaduje, v nejhorším případě, méně než  $2n$  porovnání.
  - shora dolů
    - Opakované vkládání nového klíče do již existující haldy.
    - Nový klíč vložíme na konec haldy.
    - Nový klíč porovnáme s rodičem a případně nový klíč přesuneme o patro výš.
    - Takto postupujeme dokud nenarazíme na většího rodiče nebo dojdeme do kořene haldy.
    - Výška haldy s n prvky je  $\approx \log_2 n$ , tudíž složitost vložení klíče do haldy je  $O(\log n)$ .
    - Konstrukce shora dolů je tedy složitější než zdola nahoru.
  - odstranění z haldy se provádí záměnou prvku s posledním tj. nejvíce vpravo dole a provedením konstrukce

## • Explorativní analýza dat (data a jejich vlastnosti – numerické, kategoriální a jiné atributy, statistické vlastnosti dat, vztahy mezi atributy).

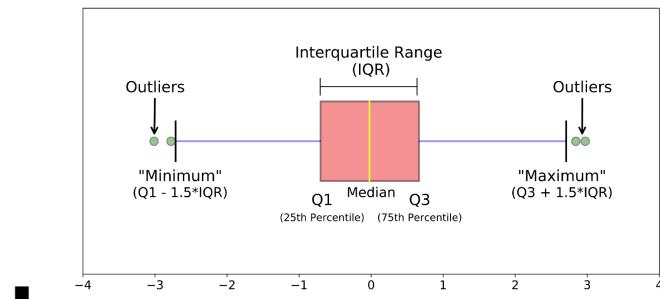
Explorativní analýza je souhrn metod používaných pro průzkum dat a hledání hypotéz, které stojí za to testovat.

Cílem explorační analýzy je nalézt zajímavá data. Vědět co mě na datech zajímá a nalézt způsob jak to zobrazit.

### Data a jejich vlastnosti

- obrázky - klasifikace (co je na obrazku), detekce (kde to je), segmentace (výřez objektu),..
- text - určení citového zabarvení, spam, .....
- získávání informací z časových řad - měření

- Datová matici uchovává řádky záznamů a sloupce vlastností
- může obsahovat data různých datových typů
- numerické hodnoty
  - celočíselná, reálná nebo komplexní čísla
  - celočíselná jsou zpracovávány jako reálná nebo kategoriální
  - Sloupce reálných čísel mají základní limity - minimum a maximum.
  - Dalšími důležitými vlastnostmi jsou průměr, medián a kvartil.
  - Nejdůležitějším rysem je rozložení reálných hodnot. Rozložení znázorňuje, které hodnoty se vyskytují a jak často.
  - Přehled funkce lze převzít z histogramu nebo boxplotu.
  - boxplot
    - rychlé nalezení mediánu, kvartilů



- histogram
  - znázornění distribuce dat, tj. znázornění množství dat v určitých rozsazích
- kategorické data
  - Kategoriální data představují soubor možností, které může funkce nabývat.
  - Data mohou být číselná nebo textová.
  - Často třída nebo značka pro data
  - zpracování je realizováno
    - binarizací
    - pořadové kódování - problém s vzdáleností některé jsou dále od 0 než jiné
    - one-hot kódování - kódovací vektor o délce počtu kategorií obsahující 0 a 1 na pozici kterou kóduje - stejná vzdálenost od 0 všech kategorií
    - algoritmické kódování (cyklická funkce)
- Textová data
  - mají podobu jednoho slova nebo otevřeného textu
  - Jednoslovny a krátký text může představovat kategoriální hodnotu
  - Sloupce s otevřeným textem se obtížně zpracovávají.
  - Obvykle se zpracovávají samostatně jako textová data: - normalizované, tokenizované, kódované (embedding), ...
- Grafy
  - Skládá se z uzlů a hran (a hodnot).
  - Zobrazuje strukturu nebo topologii nějakých informací nebo vztahů.
  - Může zobrazovat mnoho vlastností dat

#### statistické vlastnosti dat

- Populace vs. vzorek
- Populace je soubor všech objektů ve skupině.

- Vzorek je podmnožina populace.
- Náhodnost vzorku je největší otázkou. - ve vzorku by měla být rovnoměrně zastoupena každá kategorie. Vzorek musí být vhodný tedy poměrově dle zastoupení kategorií v datech
- Rozložení pravděpodobnosti je funkce, která zobrazuje pravděpodobnosti výsledků události nebo experimentu.
- Centrální tendence je centrální (nebo typická) hodnota pravděpodobnosti rozdělení.  
Nejběžnější míry centrální tendence jsou průměr, medián a modus.
  - medián je střední hodnota - hodnota uprostřed, když jsou hodnoty seřazeny v řadě.
  - modus je hodnota, která se objevuje nejčastěji.
- dále lze pozorovat rozptyl - jak se liší od průměru, odchylku
- očekávaná hodnota - vážený průměr všech možných hodnot této veličiny.
- kovariance - vyjadřuje, jak moc se variace dvou proměnných navzájem shodují. zkoumá podobnost 2 hodnot
- Korelace - normalizace kovariance pomocí směrodatné odchylky každé proměnné.  
Zobrazí které parametry spolu souvisí a které ne. Hodnota jak moc spolu korelují jednotlivé hodnoty atributů tj. jak moc si jsou podobné

### Vztahy mezi atributy

Vztahy mezi dvěma atributy jednoho objektu jsou popisovány pomocí pojmu:

- kovariance
- korelace

Vztahy mezi jednotlivými objekty na základě jednoho nebo více atributů (ale u obou objektů těch stejných)

- Vzdálenost
  - podobnost objektů na základě vzdálenosti v kartézském prostoru
  - Čím menší vzdálenost tím jsou si objekty podobnější
  - Manhattan distance 
$$d(X, Y) = \sum_{i=1}^N |x_i - y_i|$$
 - pravoúhlé trasy mezi body
  - Euclid distance 
$$d(X, Y) = \sqrt{\sum_{i=1}^N (x_i - y_i)^2}$$
 - přímá nejkratší vzdálenost
- Podobnost
  - měří blízkost mezi objekty
  - Čím vyšší hodnoty, tím jsou si objekty bližší.
  - Podobnost nevyžaduje kartézské souřadnice.
  - Lze definovat míry podobnosti související s objekty.
  - Kosinusová podobnost měří úhel mezi objekty v euklidovském prostoru.
$$\cos(X, Y) = \frac{X \cdot Y}{\|X\| \|Y\|}$$

- Metody strojového učení (shlukování, klasifikace, vyhodnocení algoritmů).

Shlukování (clustering) zsu prez 6

- Využívají se když chceme rozdělit data do přirozených skupin
- Tyto shluky odrážejí určitý mechanismus, z něhož jsou instance vybírány, který způsobuje, že některé instance nesou silnější podobnost navzájem než se zbývajícími případy.
- vytváří skupiny objektů. Tyto skupiny se mohou překrývat nebo se vylučovat.
- zařazení do skupiny může být pravděpodobnostní
- Většina shluků je založena na podobnosti mezi objekty reprezentovaných vzdáleností nebo funkcí podobnosti.
- algoritmy
- k-means
  - Základní algoritmus, který je založen na vzdálenosti pomocí euklidovské vzdálenosti.
  - k - představuje počet shluků, které budou vygenerovány.
  - Algoritmus je iterační a začíná s náhodným nebo pseudonáhodným bodů. - centroidů
  - Každá iterace se skládá z několika definovaných kroků.
  - Nejprve se všechny instance přiřadí k nejbližším centroidům. Poté jsou centroidy přesunuty k aktuálním středům.
  - algoritmus je velmi rychlý a vyžaduje pouze několik iterací
  - Někdy, v opravdu vzácných případech, vede špatná inicializace k ne dobrému shlukování.
  - Shlukování se může opakovat, ale obvykle vede ke stejnemu výsledku.
  - počet shluků se volí manuálně
  - Optimální počet shluků lze vyhodnotit pomocí SSE (sum of squared errors) pro různé počty shluků. Na základě metody Elbow lze zvolit optimální k. tj. najít zlom za kterým SSE klesá pomaleji
- hierarchické shlukování
  - Vytvoří hierarchickou strukturu nad objekty z datové sady.
  - Hierarchické uspořádání dat umožňuje ještě lepší plošné shlukování
  - metody zdola nahoru
    - Jednotlivé datové objekty jsou aglomerovány do shluků vyšší úrovně.
    - Pro výpočet podobnosti se používá objektivní funkce.
  - metody shora dolů (rozdělovací)
    - Rozdělení datových objektů do stromové struktury.
    - Při vyváženosti stromu je třeba hledat kompromis mezi počtem shluků a počtem objektů v každém shluku/listu.
- Vyhodnocení algoritmů
  - Shlukování seskupuje podobné objekty do skupin.
  - Shluky vytvořené jednotlivými algoritmy se mohou lišit. Nejdůležitější částí je porovnání vlastností objektů mezi skupinami.
  - Ideálním způsobem je vyhodnocení agregovaných vlastností skupin
  - podobně jako u explorační analýzy.
  - Skupiny by se měly lišit alespoň v jedné vlastnosti.

Klasifikace

- známe vstupní data a známe možný výstup
- snažíme se pro každý vstup přiřadit jej do jedné z výstupních kategorií
- k dispozici máme trénovací množinu, pro kterou jsou určeny výstupní kategorie, Pro vstup kde kategorie určena není musí rozhodnout do které jej přiřadit
- klasifikaci lze realizovat mnoha způsoby
  - 1R (1 rule)
    - vyhodnocování pouze podle jednoho parametru

- vybírá se parametr podle něhož má největší úspěšnost
- počítá pro každou hodnotu atributu kolikrát se vyskytuje která výstupní kategorie a vybere z nich tu dominantní, poté se určuje chybovost tj. podle hodnoty atributu se určí o kterou výstupní kategorii se jedná. Vybere se atribut s nejnižší chybovostí
- pravděpodobnostní modelování
  - pro každou hodnotu každého atributu se určí v kolika případech je určena která výstupní kategorie
  - při vstupu se poté podle hodnoty každého atributu přiřadí váhy a součinem se vypočte které kategorii se nejvíce blíží (procentuálně)
- rozhodovací stromy
  - sekvence if else podmínek
  - sestavují se podle hodnot atributů
  - podle dat se hledají pravidla a testuje se míra chybovosti pravidla
  - záleží na pořadí pravidel pro atributy - data se postupně filtruji
- jednovrstvá neurální síť (perceptron) a vícevrstvá
  - vstupní vrstva má vrchol pro každý vstupní atribut. Slouží pouze pro vstup
  - z neuronů (vrcholů) poslední vrstvy se počítá výstupní hodnota např. kategorie podle toho na kterém je 1 tj. one hot kódování
  - u složitějších sítí mohou být další "skryté" vrstvy (další perceptrony)
  - vrcholy jednotlivých vrstev jsou vzájemně propojeny (vždy pouze s vrstvou následující, nelze vrstvu přeskočit) - jsou úplně propojeny každý s každým
  - v síti s více vrstvami nevíme co přesně se děje uvnitř (na každém vrcholu se počítá funkci a rozhoduje se na který další následující vrstvy se půjde)
  - zvládne kopírovat křivky v datech

## Regresce

- dle <https://realpython.com/linear-regression-in-python/>
  - používá se pro zjišťování vztahů mezi jednotlivými atributy
  - např. při zjišťování jestli pohlaví a věk ovlivňují výši platu zaměstnance
  - dá se tedy využít pro predikci - při nalezení nejlépe padnoucích vah pro jednotlivé atributy v testovací množině se dá následně provádět klasifikace, tedy odhadování hodnoty sloupce na základě hodnot jiných sloupců
- pro výpočet konkrétní číselné hodnoty na základě vstupů
- při trénování jsou každému atributu přiřazeny váhy tak aby výstup co nejbliže kopírovat reálné výstupní hodnoty
- při hledání výsledku se každý vstupní atribut přenásobí danou vahou a získáme predikci hodnoty

## Vyhodnocení algoritmů

- Výsledky algoritmů je potřeba vyhodnotit
- Hodnocení se liší u klasifikačních a/nebo shlukovacích algoritmů.
- Hodnocení musí být systematické.
- Hodnocení je také třeba definovat podle hodnocené metriky.

## Shlukování

- interní validační kritéria
- Součet kvadratických vzdáleností k centroidům
  - součet vzdáleností každého bodu k odpovídajícímu centroidu - používá se jako míra kvality, menší hodnota znamená lepší kvalitu shlukování

$$SSQ = \sum_{X \in D} dist(X, C)^2$$

○ - C je nejbližší centroid

- Poměr vzdáleností uvnitř shluku a mezi shluky.
  - Na základě souborů náhodných dvojic objektů.
  - Počítají se vzdálenosti mezi body ve stejném shluku - Intra, a vzdálenosti bodů mezi různými shluky - Inter
  - Čím menší je hodnota Intra/Inter tím lepší je kvalita
- Pravděpodobnostní míra
- Čistota
- gini index - nejlepší místo pro rozdělení skupin
- entropie
- interní kritéria jsou užitečná pokud nejsou dostupná žádná externí kritéria
- Hlavní využití těchto kritérií je pro porovnání algoritmu ze stejné třídy nebo různých běhů stejného algoritmu.
- externí validační kritéria
- data s informací jak by měly shluky vypadat

### Klasifikace

- data set se ideálně rozdělí na tréninková, testovací a validační data (stačí trénovací a testovací)
- Použití stejných dat pro trénink a testování není možné z důvodu nadměrného přizpůsobení a nadhodnocení.
- Validační část se používá pro ladění parametrů nebo řešení modelu. Když je ladění parametrů dokončeno, model se rekonstruuje na celé datové sadě. Znalosti z testovací datové sady by se neměly používat při ladění parametrů.
- Přesnost - podíl testovacích případů, ve kterých je předpovězená hodnota se shodovala s pravdivou hodnotou.
  - ideálně počítat TP / TN / TP + TN + FP + FN
- při trénování je datová množina rozdělena na trénovací a testovací. Získáme zpětnou vazbu jak je model úspěšný na testovací množině
- Cross validace -
  - Využívá se pro vyhodnocení modelů strojového učení na limitované množině dat
  - Vstupní množina dat je rozdělena na podmnožiny. Jedna podmnožina slouží jako testovací množina, zbylé podmnožiny slouží jako trénovací množiny. Klasifikátor natrénuje model na trénovací množině a pomocí testovací množiny testuje přesnost a výkonnost tohoto modelu. Tento proces se několikrát opakuje, pokaždé s jinou podmnožinou tvořící trénovací a testovací množinu. Využívá se také náhodné promíchaní dat ve vstupní množině

**Příklad otázky:** Základní myšlenka strategie rozděl a panuj. Praktická ukázka použití, QuickSort a jeho časová složitost. Srovnání se strategií řešení hrubou silou, BubbleSort.

## Teoretická informatika (ULM, ZDS, UTI, PJP)

- Logika (výroková logika a predikátová logika, syntaxe a sémantika formulí výrokové a predikátové logiky, sémantické a syntaktické dokazování, ekvivalentní úpravy, rezoluční metoda).

- Logika je nástroj, který pomáhá objevovat vztah logického vyplývání, tj. ověřovat platnost argumentů. řešit úlohy typu „Co vyplývá z daných předpokladů“?, a pod.
- Logika pomáhá naší „logické intuici“, která může někdy selhat. Premisy mohou být složitě formulované, „zapletené do sebe a do negací“, vztah vyplývání pak není na první pohled patrný.
- jsou-li pravdivé všechny předpoklady, pak musí být pravdivý i závěr.

### Výroková logika

- Analyzuje způsoby skládání jednoduchých výroků do výroků složených pomocí logických spojek.
- Výrok je tvrzení, o němž má smysl prohlásit, zda je pravdivé či nepravdivé
  - např. „Je-li dnes pondělí, nepůjdu na pivo.“
    - $a \Rightarrow \neg b$
- dvouhodnotová logika - pravda nepravda (existují i vícehodnotové)
- jednoduché výroky - žádná vlastní část jednoduchého výroku již není výrokem
- složené výroky - výrok má části, které jsou také výrokem (spojeny spojkou)
- Formální jazyk (daný abecedou a gramatikou) je převáděn na jazyk výrokové logiky (daný abecedou a symboly negace, konjunkce, disjunkce, implikace, ekvivalence.)
- výroky utvořené dle pravidel gramatiky se nazývají dobře utvořené formule
- sémantika (význam) formulí
  - pravdivostní ohodnocení výrokových symbolů je přiřazení pravdivostní hodnoty 0 nebo 1.
  - Pravdivostní funkce pro každé pravdivostní ohodnocení symbolů přiřazují pravdivostní hodnotu

Tabulka pravdivostních funkcí

| A | B | $\neg A$ | $A \vee B$ | $A \wedge B$ | $A \supset B$ | $A \equiv B$ |
|---|---|----------|------------|--------------|---------------|--------------|
| 1 | 1 | 0        | 1          | 1            | 1             | 1            |
| 1 | 0 | 0        | 1          | 0            | 0             | 0            |
| 0 | 1 | 1        | 1          | 0            | 1             | 0            |
| 0 | 0 | 1        | 0          | 0            | 1             | 1            |

- převádí se do přirozeného jazyka do výrokové logiky, nalezení a rozpoznání výroků (není pravda že, a, nebo, jestliže pak, tehdy a jen tehdy)
- splnitelnost formulí
  - splnitelná - alespoň v 1 případě splnitelná
  - tautologie - splněna ve všech případech
  - kontradikce - nesplnitelná
- (Výrokově) logické vyplývání
  - jestliže jsou pravdivé předpoklady pak musí být pravdivý závěr (implikace)

- Důkaz, že formule je tautologie, nebo že závěr Z logicky vyplývá z předpokladů:
    - Přímo – např. pravdivostní tabulkou
    - Nepřímo, sporem - hledám ohodnocení takové že předpoklady mají log hodnotu 1 závěr 0
    - důkaz tautologie sporem - naleznu spor v celé negované formuli - je tautologie
    - důkaz splnitelnosti sporem - naleznu spor ve formuli která má negovaný závěr
- normální formy
  - existuje mnoho ekvivalentních formulí, normální formy jsou standardizovaný zápis formulí
  - úplná disjunktivní normální forma - formule má tvar disjunkce elementárních konjunkcí (součet součinů) (kontradikce nemá udnf)
  - úplná konjunktivní normální forma - formule má tvar konjunkce elementárních disjunkcí (součin součtu) (tautologie nemá uknf)
- všechny logické funkce lze realizovat pouze pomocí negace, disjunkce a konjunkce
- Rezoluční metoda
  - metoda pro dokazování pravdivostí
  - predikát se rozdělí na elementární disjunkce (tj. konjunkce nový řádek) a tyto řádky se spojují a odstraňují výroky normální s negovaným
    - spojování = rezoluce
      - není ekvivalentní úprava, ale zachovává splnitelnost
      - $(A \vee p) \wedge (B \vee \neg p)$  můžu spojit na  $(A \vee B)$
  - hledáme spor tj prázdný řádek
    - kontradikce - je tam spor
    - tautologie - v negované formuli je spor (negovaná formule je kontradikce -> původní je tautologie)
    - splnitelná - zneguji závěr -> pokud je spor je splnitelná (tzn. nenastane situace že předpoklady jsou pravdivé a závěr ne)
  - Důkaz nesplnitelnosti formule:  $(\neg q \supset p) \wedge (p \vee r) \wedge (q \supset \neg r) \wedge \neg p$ 
    1.  $q \vee p$
    2.  $p \vee r$
    3.  $\neg q \vee \neg r$
    4.  $\neg p$
    5.  $p \vee \neg r$  rezoluce 1, 3
    6.  $p$  rezoluce 2, 5
    7. # spor 4, 6
    - Nepřímý důkaz platnosti úsudku:  
 $p \supset (q \vee r), \neg s \supset \neg q, t \vee \neg r \models p \supset (s \vee t)$ 
      1.  $\neg p \vee q \vee r$
      2.  $s \vee \neg q$
      3.  $t \vee \neg r$
      4.  $\neg p \vee s \vee r$  rezoluce 1, 2
      5.  $\neg p \vee s \vee t$  rezoluce 3, 4
      6.  $p$  nego-
      7.  $\neg s$  vaný
      8.  $\neg t$  závěr
      9.  $\neg p \vee s \vee t, p, \neg s, \neg t \models s \vee t, \neg s, \neg t \models t, \neg t \models \#$
    - lze využít i přímý důkaz a tehdy se dělají rezoluce tak aby se z nich vytvořil závěr

●

## Predikátová logika

- Je rozšířením výrokové logiky. Má bohatší vyjadřovací schopnost.
- Umožňuje rozčlenit individua, predikáty a funkční symboly
- formální jazyk - abeceda
  - logické symboly
    - individuové proměnné:  $x, y, z, \dots$
    - Symboly pro spojky:  $\neg, \wedge, \vee, \supset, \equiv$
    - Symboly pro kvantifikátory:  $\forall, \exists$  - pro všechna, existuje
  - speciální symboly
    - predikátové  $P, Q, \dots$
    - funkční  $f, g, \dots$
- formální jazyk - Gramatika
  - termy
    - každý symbol proměnné  $x, y, \dots$  je term
    - tj. proměnné a funkce
  - atomické formule
    - výrazy (predikátové symboly) obsahující termy
  - formule
    - každá atomická formule je formule
- PL 1. řádu
  - Jediné proměnné, které můžeme používat s kvantifikátory, jsou individuové proměnné
  - Nemůžeme kvantifikovat přes proměnné vlastnosti či funkcí
- Sémantika
  - dle  
[https://is.muni.cz/www/98951/41610771/43823411/43823458/Zaklady\\_informat/Teoreticke\\_zakla/TZI-predikatova-logika.pdf](https://is.muni.cz/www/98951/41610771/43823411/43823458/Zaklady_informat/Teoreticke_zakla/TZI-predikatova-logika.pdf)
    - "Sémantika neboli význam formulí predikátové logiky 1. řádu, je dána jejich interpretaci"
    - sémantikou predikátové logiky se tedy rozumí správné tvoření výrazů a jejich přepis do jazyka predikátové logiky
    - příklad takového přepisu:
      - "Všechny jedle ( $J$ ) jsou stále zelené ( $Z$ )"
      - se přepíše jako:  $\forall x (J(x) \Rightarrow Z(x))$ 
        - samotný zápis " $\forall x (J(x) \Rightarrow Z(x))$ " nic neznamená a může odpovídat nekonečně mnoha výrokům
        - je potřeba definovat co použité symboly znamenají pro ověření správnosti přepisu ->  $J$  znamená \*být jedlí\* a  $Z$  znamená \*být stále zelený\*
    - složitější příklad
      - "Ne každý talentovaný ( $T$ ) spisovatel ( $Sp$ ) je slavný ( $Sl$ )."
      - $\neg \forall x \{ [T(x) \wedge Sp(x)] \Rightarrow Sl(x) \}$ 
        - negací všeobecného kvantifikátoru -  $\forall$  vznikne kvantifikátor existenční -  $\exists$  (funguje to i naopak) a ještě negací implikace vznikne finálně toto:
        - $\exists x \{ T(x) \wedge Sp(x) \wedge \neg Sl(x) \}$

- “Existuje talentovaný (T) spisovatel (Sp), který není slavný (Sl).”
- Rezoluční metoda
  - Typické úlohy:
    - dokázání logické pravdivosti formule
    - Dokázat platnost úsudku v PL1
  - Je aplikovatelná na formuli ve spec. konjunktivní normální formě (Skolemova klausulární forma)
  - Je zobecněním rezoluční metody ve výrokové logice: důkaz sporem
  - Je základem pro automatizované deduktivní metody a logické programování
    - **Převod do klausulární formy (Skolem)**
    1. Utvoření existenčního uzávěru (zachovává splnitelnost)
    2. Eliminace nadbytečných kvantifikátorů
    3. Eliminace spojek  $\supset$ ,  $\equiv$
    4. Přesun negace dovnitř
    5. Přejmenování proměnných
    6. Přesun kvantifikátorů doprava
    7. Eliminace existenčních kvantifikátorů (Skolemizace – zachovává splnitelnost)
    8. Přesun všeobecných kvantifikátorů doleva
    9. Použití distributivních zákonů
    - **Rezoluční metoda - Příklad**
    1.  $P(x, f(x))$
    2.  $Q(y, g(y))$
    3.  $\neg P(\alpha, v) \vee \neg Q(v, w)$ 
      - Nyní se budeme pokoušet o rezoluci. Abychom mohli uplatnit rezoluci na klausule 1. a 3., musíme substituovat term  $\alpha$  za **všechny výskyty** proměnné  $x$  a  $f(a)$  za  $v$ .
    4.  $\neg Q(f(a), w)$  rezoluce 1., 3., substituce  $x / a, v / f(a)$
    5. ■ rezoluce 2., 4., substituce  $y / f(a), w / g(f(a))$

Negovaná formule je kontradikce, původní je tedy logicky pravdivá

    - Pozn.: Je nutno substituovat za **všechny výskyty** proměnné!
  - Postup důkazů rezoluční metodou
    - Důkaz, že formule A je logicky pravdivá:
    - Formuli znegujeme
    - Formuli  $\neg A$  převedeme do klausulární Skolemovy formy  $\neg(A)^S$
    - Postupným uplatňováním rezolučního pravidla se snažíme dokázat nesplnitelnost formule  $\neg(A)^S$  a tedy také formule  $\neg A$ .
    - Důkaz platnosti úsudku
    - Závěr Z znegujeme
    - Formule předpokladů a negovaného závěru převedeme do klausulární formy
    - Postupným uplatňováním rezolučního pravidla se snažíme dokázat nesplnitelnost množiny

## Sémantické a syntaktické dotazování

- Možná je tím myšleno “sémantické ověření správnosti úsudku”

- což je dokazování, že je úsudek správný (to samé o co se snaží rezoluční metoda) akorát je to obtížnější, protože nad tím musíš přemýšlet a logicky najít ten spor
- níže je příklad

**Příklad:** Sémanticky ověrte správnost úsudku:

Marie má ráda pouze vítěze.  
Karel je vítěz.

---

Marie má ráda Karla.

**Řešení:** Úsudek zformalizujeme:

$$\begin{array}{c} \forall x [R(M,x) \Rightarrow V(x)] \\ V(K) \\ \hline R(M,K) \end{array}$$

Aby byly předpoklady pravdivé, pak možné interpretace nad množinou individui D musí mít tvar:

$M_D : \text{Marie}, K_D : \text{Karel}$  (poznámka: realizací těchto konstant mohou být kterékoli jiné prvky D, avšak celková úvaha se tím nijak nemění.)

$$\begin{array}{ll} R_D \subset D \times D : & \{\dots <\text{Marie}, i_1>, <\text{Marie}, i_2>, \dots, <\text{Marie}, i_n> \dots\} \\ V_D \subset D : & \{\dots i_1, i_2, \dots, \text{Karel}, \dots, i_n \dots\} \end{array}$$

- Vidíme, že závěr nevyplývá, neboť není zaručeno, že relace  $R_D$  bude obsahovat dvojici  $\langle \text{Marie}, \text{Karel} \rangle$ .
- logicky: Lidi, které má Marie ráda, jsou vítězové, ale to neznamená, že pokud jsi vítěz, tak tě má Marie ráda.

## Ekvivalentní úpravy (ve výrokové logice)

### Algebraické zákony pro konjunkci, disjunkci a ekvivalence

- $\models (p \vee q) \equiv (q \vee p)$  komutativní zákon pro  $\vee$
- $\models (p \wedge q) \equiv (q \wedge p)$  komutativní zákon pro  $\wedge$
- $\models (p \equiv q) \equiv (q \equiv p)$  komutativní zákon pro  $\equiv$
- $\models [(p \vee q) \vee r] \equiv [p \vee (q \vee r)]$  asociativní zákon pro  $\vee$
- $\models [(p \wedge q) \wedge r] \equiv [p \wedge (q \wedge r)]$  asociativní zákon pro  $\wedge$
- $\models [(p \equiv q) \equiv r] \equiv [p \equiv (q \equiv r)]$  asociativní zákon pro  $\equiv$
- $\models [(p \vee q) \wedge r] \equiv [(p \wedge r) \vee (q \wedge r)]$  distributivní zákon pro  $\wedge, \vee$
- $\models [(p \wedge q) \vee r] \equiv [(p \vee r) \wedge (q \vee r)]$  distributivní zákon pro  $\vee, \wedge$

#### - Zákony pro převody

$$\begin{aligned} \models (p \equiv q) &\equiv (p \supset q) \wedge (q \supset p) \\ \models (p \equiv q) &\equiv (p \wedge q) \vee (\neg q \wedge \neg p) \\ \models (p \equiv q) &\equiv (\neg p \vee q) \wedge (\neg q \vee p) \\ \models (p \supset q) &\equiv (\neg p \vee q) \\ \models \neg(p \supset q) &\equiv (p \wedge \neg q) \text{ Negace implikace} \\ \models \neg(p \wedge q) &\equiv (\neg p \vee \neg q) \quad \text{De Morgan zákony} \\ \models \neg(p \vee q) &\equiv (\neg p \wedge \neg q) \quad \text{De Morgan zákony} \end{aligned}$$

- Tyto zákony jsou také návodem jak **negovat**
- normální formy viz výše
- vše lze realizovat pomocí funkcí negace, disjunkce a konjunkce - funkcionálně úplná soustava (lze si vystačit i pouze s negací a konjunkcí nebo disjunkcí tj. NAND nebo NOR)

- **Booleova algebra (Booleova funkce, minimalizace, vazba na kombinační obvody).**

Dvouhodnotová algebra používána pro reprezentaci pravdivostních hodnot logických funkcí.

Obsahuje 2 binární a jednu unární operaci.

Booleovská funkce

- Dvouhodnotové funkce s dvouhodnotovými proměnnými.
- Každé n - tici hodnot proměnných přiřazují hodnotu 0 nebo 1. - tj pouze hodnoty 0,1
- Popisují chování logických obvodů.
- Úplná booleovská funkce
  - definiční obor  $\{0,1\}$  a obor hodnot je  $\{0,1\}$
  - funkce má přesně definovaný výstup
- základní způsob zobrazení funkce je pravdivostní tabulka
  - Každá kombinaci vstupních proměnných přiřazuje výstupní hodnotu Booleovské funkce.
  - Má  $2^n$  řádků; n je počet vstupních proměnných.
  - vektory definičního oboru lze považovat za binární čísla
- Neúplná booleovská funkce
  - není definována ve všech bodech definičního oboru
  - definiční obor  $\{0,1\}$  a obor hodnot je  $\{0,1,X\}$  X - neurčitá hodnota, tj. libovolná 0 nebo 1

| <i>n</i> | $x_2$ | $x_1$ | $x_0$ | $f_1$ |
|----------|-------|-------|-------|-------|
| 0        | 0     | 0     | 0     | 0     |
| 1        | 0     | 0     | 1     | 1     |
| 2        | 0     | 1     | 0     | 0     |
| 3        | 0     | 1     | 1     | 1     |
| 4        | 1     | 0     | 0     | 1     |
| 5        | 1     | 0     | 1     | 1     |
| 6        | 1     | 1     | 0     | 1     |
| 7        | 1     | 1     | 1     | 0     |

- vektory definičního oboru lze považovat za binární čísla

| <i>n</i> | $x_2$ | $x_1$ | $x_0$ | $f_2$ |
|----------|-------|-------|-------|-------|
| 0        | 0     | 0     | 0     | 0     |
| 1        | 0     | 0     | 1     | X     |
| 2        | 0     | 1     | 0     | 0     |
| 3        | 0     | 1     | 1     | 1     |
| 4        | 1     | 0     | 0     | X     |
| 5        | 1     | 0     | 1     | 1     |
| 6        | 1     | 1     | 0     | 1     |
| 7        | 1     | 1     | 1     | 0     |

Booleova algebra

- Booleova algebra je základní matematika pro studium návrhu logických digitálních systémů. Je to soubor pravidel pro zápis a vyhodnocování logických vztahů.
- Struktura:
  - Logické proměnné (nabývají pouze dvou hodnot)
  - Logické operátory
    - komplement (negace)
    - Logický součin (konjunkce), funkce AND
    - Logický součet (disjunkce), funkce OR

- definováno axiomy a teorémy

## Axiomy Booleovy algebry

|           |                                  |           |                             |
|-----------|----------------------------------|-----------|-----------------------------|
| <b>A1</b> | $x = 0$ když $x \neq 1$          | <b>A3</b> | $0 \cdot 0 = 0$             |
|           | $x = 1$ když $x \neq 0$          |           | <u><math>1+1=1</math></u>   |
| <b>A2</b> | Když $x = 0$ , pak $\bar{x} = 1$ | <b>A4</b> | $1 \cdot 1 = 1$             |
|           | Když $x = 1$ , pak $\bar{x} = 0$ |           | $0 + 0 = 0$                 |
|           |                                  | <b>A5</b> | $1 \cdot 0 = 0 \cdot 1 = 0$ |
|           |                                  |           | $0 + 1 = 1 + 0 = 1$         |

## Teorémy pro jednu proměnnou

- Komplementarita  
(log. rozpor, zákon vyloučení třetího)
- Neutralita 0 a 1
- Agresivita 0 a 1
- Zákon opakování (idempotence)
- Zákon dvojí negace

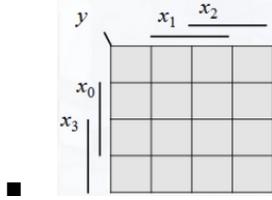
|                       |                 |
|-----------------------|-----------------|
| $x \cdot \bar{x} = 0$ | $x \cdot x = x$ |
| $x + \bar{x} = 1$     | $x + x = x$     |
| $x \cdot 1 = x$       | $=$             |
| $x + 0 = x$           | $x = x$         |
| $x \cdot 0 = 0$       |                 |
| $x + 1 = 1$           |                 |

Teorémy pro dvě a více proměnných 1/3

- ▶ Komutativní zákon  $x_0 \cdot x_1 = x_1 \cdot x_0$   
 $x_0 + x_1 = x_1 + x_0$
- ▶ Asociativní zákon  $x_0 \cdot (x_1 \cdot x_2) = (x_0 \cdot x_1) \cdot x_2$   
 $x_0 + (x_1 + x_2) = (x_0 + x_1) + x_2$
- ▶ Distributivní zákon  $x_0 \cdot (x_1 + x_2) = x_0 \cdot x_1 + x_0 \cdot x_2$   
 $x_0 + (x_1 \cdot x_2) = (x_0 + x_1) \cdot (x_0 + x_2)$

- zápis booleovských funkcí

- pravdivostní tabulkou – jednoznačně definuje úplnou a neúplnou Booleovskou funkci (název je z výrokové logiky, kde se hodnotám 0 a 1 přiřazuje význam nepravdivosti nebo pravdivosti).
  - Kanonická forma
    - úplná disjunktivní normální forma - disjunkce konjunkcí (součet součinů), tj. součet mintermů (dílčích součinů)
    - úplná konjunktivní normální forma - konjunkce disjunkcí (součin součtů), tj. součin maxtermů (dílčích součtů)
  - Karnaughova mapa
    - Grafická metoda pro zápis a minimalizaci Booleovské funkce.
    - Každá proměnná v mapě dělí všechny buňky mapy na dvě poloviny, kde v jedné polovině proměnná má hodnotu log 1 a v druhé hodnotu log 0.



- sousední buňky se liší pouze jednou proměnnou a jejich počet je dán počtem proměnných dané funkce
- Každá buňka má přiřazen jedinečný minterm a maxterm.

### Minimalizace booleovských funkcí

- hledáme ekvivalentní výraz, který má nejmenší četnost vyskytujících se proměnných
  - nejčastěji minimální disjunktivní tvar nebo minimální součtový tvar
- minimalizujeme aby počet prvků pro realizaci byl co nejnižší
- nejčastější způsoby:
  - algebraická minimalizace
    - aplikace axiomů a teorému booleovy algebry na zápis log fce ve formě udnf nebo uknf
    - úroveň zjednodušení je dána zkušenostmi a intuicí
  - minimalizace pomocí Karnaughovy mapy
    - Princip spočívá v grafickém sloučení sousedních čtverečků a jejich popsání pomocí vstupních proměnných.
    - pro disjunktivní tvar vytváříme smyčky které pokryjí log 1, smyčky děláme co největší aby byly realizovány co nejmenším počtem proměnných, smyčky mohou obsahovat 1,2,4,8 atd prvků tj. mocnina čísla 2
    - počet smyček musí být minimální a musí pokrýt všechny 1
    - z každé smyčky vyjádříme součiny, kde prom s log 1 se uvede jako přímá a s 0 jako negovaná

| $f_1$ | $x_1$ | $x_2$ |   |
|-------|-------|-------|---|
| $x_0$ | 0     | 0     | 1 |
| 1     | 1     | 0     | 1 |

$$f_1(x_2, x_1, x_0) = \overline{x_2} \cdot x_0 + x_2 \cdot \overline{x_1} + x_2 \cdot x_0 = \overline{x_2} \cdot x_0 + \overline{x_1} \cdot x_0 + x_2 \cdot \overline{x_0}$$

- 
- kro konjunktivní tvar vytváříme co největší smyčky, které pokryjí čtverečky, v nichž funkce nabývá hodnotu log 0.
- počet smyček musí být minimální a musí pokrýt všechny 0
- Z každé smyčky vyjádříme součty, kde proměnná s hodnotou 0 se uvede jako přímá, proměnná s hodnotou 1 jako negovaná.

► V konjunktivním tvaru:

| $f_1$ | $x_1$ | $x_2$ |   |
|-------|-------|-------|---|
| $x_0$ | 0     | 0     | 1 |
|       | 1     | 1     | 0 |
|       | 1     | 0     | 1 |

$f_1(x_2, x_1, x_0) = (x_2 + x_0)(\bar{x}_2 + \bar{x}_1 + \bar{x}_0)$

### Vazba na kombinační obvody

- logická funkce
  - dvouhodnotová funkce s dvouhodnotovými proměnnými
  - popisuje chování logického obvodu
- logický kombinační obvod
  - Obvod, jehož výstupní proměnné závisí pouze na logických hodnotách vstupních proměnných.
  - m vstupních a n výstupních proměnných
- logický člen
  - základní stavební prvek logických obvodů, který realizuje logickou funkci
  - má 1 nebo více vstupů a 1 výstup
  - sestaven z elektronických součástek (tranzistory, diody, rezistory,...)
  - logické členy:

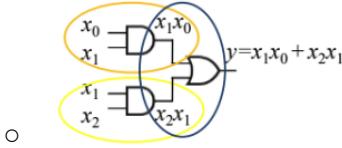
| Gate   | ANSI/MIL | IEC | DIN | British (BS 3939) |
|--------|----------|-----|-----|-------------------|
| Buffer |          |     |     |                   |
| NOT    |          |     |     |                   |
| AND    |          |     |     |                   |
| NAND   |          |     |     |                   |
| OR     |          |     |     |                   |
| NOR    |          |     |     |                   |
| XOR    |          |     |     |                   |
| XNOR   |          |     |     |                   |

- Realizace logických funkcí
- pomocí AND a OR
  - Vycházíme ze součtového tvaru
  - Dílčí součiny realizujeme logickými členy AND
  - Výsledný součet realizujeme logickým členem OR

| $y$   | $x_1$ | $x_2$ |   |
|-------|-------|-------|---|
|       | 0     | 0     | 1 |
| $x_0$ | 0     | 1     | 0 |

$$y = f(x_2, x_1, x_0)$$

$$y = x_1 \cdot x_0 + x_2 \cdot x_1$$



- pomocí NAND a NOR
  - Vycházíme ze součtového tvaru
  - Výraz upravíme na tvar obsahující pouze součiny (pomocí zákona o dvojí negaci a De Morganových pravidel).

$$\overline{A+B} = \overline{A} \cdot \overline{B}$$

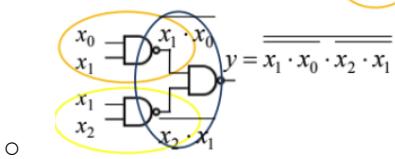
■ de morgan       $\overline{A \cdot B} = \overline{A} + \overline{B}$

- Dílčí negované součiny realizujeme logickými členy NAND a výsledný negovaný součin realizujeme opět logickým členem NAND.

| $y$   | $x_1$ | $x_2$ |   |
|-------|-------|-------|---|
|       | 0     | 0     | 1 |
| $x_0$ | 0     | 1     | 1 |

$$y = f(x_2, x_1, x_0)$$

$$y = x_1 \cdot x_0 + x_2 \cdot x_1 = \\ = \overline{\overline{x_1 \cdot x_0} + \overline{x_2 \cdot x_1}} = \\ = \overline{\overline{x_1} \cdot \overline{x_0} + \overline{x_2} \cdot \overline{x_1}}$$



- pomocí OR a AND
  - Vycházíme ze součinového tvaru, stejné
- pomocí NOR a NOR
  - Vycházíme ze součinového tvaru, stejné
- základní kombinační obvody - multiplexor, demultiplexor, kodér, dekodér

- **Množiny, relace a funkce (operace na množinách, druhy a vlastnosti relací, relace typu, ekvivalence a uspořádání, vlastnosti funkcí, induktivní definice, a důkazy).**

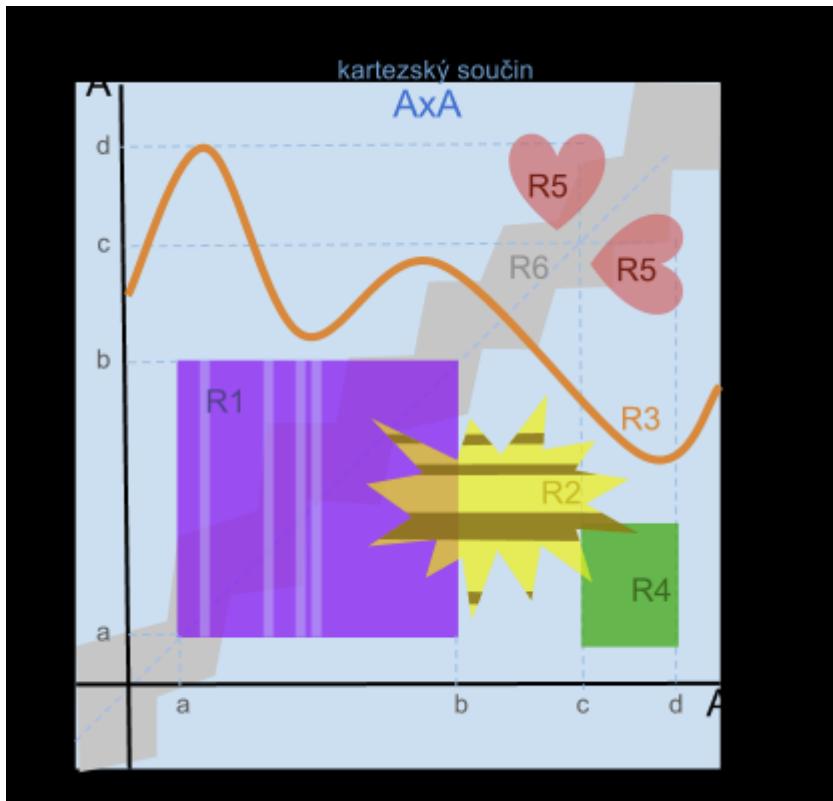
### Množiny

- Množina je soubor prvků a je svými prvky plně určena; množinu s prvky a, b, c značíme: {a, b, c}

- Prvkem množiny může být opět množina, množina nemusí mít žádné prvky (značíme  $\emptyset$ ), prvky množiny se nemůžou opakovat
  - Množiny jsou identické, právě když mají stejné prvky
  - množinové operace:
  - Sjednocení  $A \cup B = \{x \mid x \in A \text{ nebo } x \in B\}$ 
    - „Množina všech  $x$  takových, že  $x$  je prvkem  $A$  nebo  $x$  je prvkem  $B$ .“
    - $\{a, b, c\} \cup \{a, d\} = \{a, b, c, d\}$
  - Průnik:  $A \cap B = \{x \mid x \in A \text{ a } x \in B\}$ 
    - „Množina všech  $x$  takových, že  $x$  je prvkem  $A$  a současně  $x$  je prvkem  $B$ .“
    - $\{a, b, c\} \cap \{a, d\} = \{a\}$
  - Rozdíl:  $A \setminus B = \{x \mid x \in A \text{ a } x \notin B\}$ 
    - $\{a, b, c\} \setminus \{a, b\} = \{c\}$
  - Doplňek (komplement)
    - Nechť  $A \subseteq M$ . Doplňek  $A$  vzhledem k  $M$  je množina  $A' = M \setminus A$ , tzn. co je navíc v  $M$  oproti  $A$ .
  - Kartézský součin:  $A \times B = \{(a, b) \mid a \in A, b \in B\}$ ,
    - kde  $a, b$  je uspořádaná dvojice (záleží na pořadí)
    - výstupem je množina uspořádaných dvojic  $(a, b)$  - u n-tic záleží na pořadí (narozdíl od množin)
    - obsahuje kombinace všech prvků - každý s každým, záleží na pořadí
  - Potenční množina:  $P(A) = \{B \mid B \subseteq A\}$ , značíme také  $2^A$ 
    - $P(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
    - všechny kombinace prvků
  - vztahy mezi množinami:
  - Množina  $A$  je podmnožinou množiny  $B$ , značíme  $A \subseteq B$ , právě když každý prvek  $A$  je také prvkem  $B$ .
  - Množina  $A$  je vlastní podmnožinou množiny  $B$ , značíme  $A \subset B$ , právě když každý prvek  $A$  je také prvkem  $B$  a ne naopak.
- platí:
- $\{1, 2, 3\} \subseteq \{1, 2, 3\}$
- $\{1, 2, 3\} \subseteq \{1, 2, 3, 4\}$
- $\{1, 2, 3\} \subset \{1, 2, 3, 4\}$
- neplatí:
- $\{1, 2, 3\} \subset \{1, 2, 3\}$
- 

Relace <http://lucie.zolta.cz/index.php/zaklady-teoreticke-informatiky/17-relace/20-relace>

- je podmnožina Kartézského součinu.
- binární relace - podmnožina kartézského součinu 2 množin
- ternární - .. 3 množin
- Každou relaci lze znázornit tabulkou, kde řádky jsou jednotlivé n-tice
- Pokud jsou množiny kartézského součinu shodné jde o relaci homogenní v opačném případě, jde o heterogenní relaci.
- Vlastnosti relací:

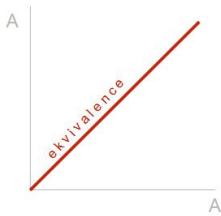


- 
- **reflexivní**, jestliže pro všechna  $a \in A$  platí  $(a, a) \in R$ .
  - Relací musí obahovat celou osu kartézského součinu (obr. šedá relace R6).
  - tzn. prvky jsou v relaci samy se sebou.
- **ireflexivní**, jestliže pro všechna  $a \in A$  platí  $(a, a) \notin R$ .
  - Relací nesmí obsahovat ani kousek osy kartézského součinu (obr. R4, R5).
- **symetrická**, jestliže pro všechna  $a, b \in A$  platí, že pokud  $(a, b) \in R$ , pak  $(b, a) \in R$ .
  - Relace je symetrická podle osy kartézského součinu (obr. R1, R5).
- **asymetrická**, jestliže pro všechna  $a, b \in A$  platí, že pokud  $(a, b) \in R$ , pak  $(b, a) \notin R$ .
  - Relace se nachází pouze v jedné polovině kart. součinu rozděleného osou, přičemž se nesmí dotýkat osy (obr. R4).
- **antisymetrická**, jestliže pro všechna  $a, b \in A$  platí, že pokud  $(a, b) \in R$  a  $(b, a) \in R$ , pak  $a = b$ .
  - Relace se nachází pouze v jedné polovině kart. součinu rozděleného osou a může obsahovat osu.
- **tranzitivní**, jestliže pro všechna  $a, b, c \in A$  platí, že pokud  $(a, b) \in R$  a  $(b, c) \in R$ , pak  $(a, c) \in R$ .

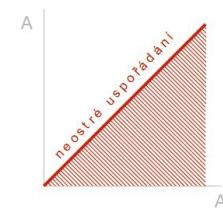
Typy binárních relací:

<http://lucie.zolta.cz/index.php/zaklady-teoreticke-informatiky/17-relace/21-typy-binarnich-relaci>

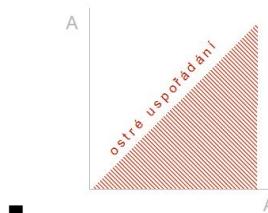
- Ekvivalence je binární relace, která je reflexivní, symetrická, tranzitivní



- Uspořádání
  - Binární relaci nazveme **neostrým uspořádáním** pokud je:
    - reflexivní, antisymetrická, tranzitivní



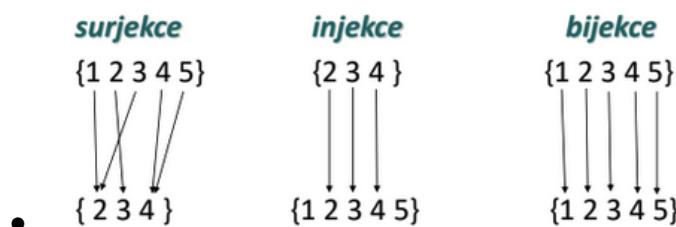
- Binární relaci nazveme **ostrým uspořádáním** pokud je:
  - asymmetrická, ireflexivní, tranzitivní



- Uspořádání je úplné (lineární) pokud neexistují neporovnatelné prvky.

### Vlastnosti funkcí (zobrazení)

- $n$ -árni funkce  $f$  na množině  $M$  je speciální zprava jednoznačná  $(n+1)$ -árni relace  $f \subseteq M \times \dots \times M - t.j. f$  funkce je podmnožinou relace
- ke každé  $n$ -tici prvků  $a \subseteq M \times \dots \times M$  existuje nanejvýš jeden prvek  $b \in M$
- $M \times \dots \times M$  nazýváme definiční obor funkce, množinu  $M$  pak obor hodnot
- typy
- Surjekce
  - jestliže k libovolnému  $b \in B$  existuje  $a \in A$  takový, že  $f(a)=b$ .
- Injekce
  - jestliže pro všechna  $a \in A, b \in A$  taková, že  $a \neq b$  platí, že  $f(a) \neq f(b)$ .
- Bijekce
  - bijekce (prosté zobrazení  $A$  na  $B$ ), jestliže  $f$  je surjekce a injekce.
  - velikost množin musí být stejná



## Induktivní definice a důkazy

- U induktivní definice se jedná jen o další způsob jak popsat množinu jinak než výčtem prvků.
  - induktivní definice se skládá z
    - několika bázických prvků = prostě 1 nebo více prvků, o kterých řeknu že jsou v té množině
      - např.  $x \in M$
    - induktivních pravidel
      - např. "je-li  $x \in M$  tak  $x + 1 \in M$ "
- Důkaz indukcí
  - Používá se k dokazování matematických vět.
  - Skládá se ze dvou kroků:
    - dokážu, že něco platí pro nějaký snadno dokazatelný případ z dané množiny (např. pro číslo 1)
    - Indukční krok - pokud to platí pro prvek z předchozího kroku, tak to musí platit i pro tento prvek, protože ten je rozložitelný na ten prvek z předchozího kroku (např. číslo 2, lze rozložit na 1 a 1 a pro tyto tvrzení platí, protože jsem to dokázal v předchozím kroku)
- 

## • Teorie formálních jazyků a automatů (formální jazyky, operace na jazycích, formální prostředky používané pro popis jazyků – automaty, gramatiky, regulární výrazy, konečné, deterministické a nedeterministické automaty, bezkontextové gramatiky, zásobníkové automaty, Chomského hierarchie).

- (Formální) jazyk  $L$  v abecedě  $\Sigma$  je nějaká libovolná podmnožina množiny  $\Sigma^*$ , tj.  $L \subseteq \Sigma^*$
- $\Sigma^*$  - množina všech slov
- Abeceda je libovolná neprázdná konečná množina symbolů (znaků).
- Slovo v dané abecedě je libovolná konečná posloupnost symbolů z této abecedy.
- Řekněme, že jsme nějaké jazyky již popsali. Z těchto jazyků můžeme vytvářet další nové jazyky pomocí nejrůznějších operací na jazycích.

## Operace na jazycích

- Popis nějakého komplikovaného jazyka můžeme tedy "dekomponovat" tím způsobem, že tento jazyk vyjádříme jako výsledek aplikování nějakých operací na nějaké jednodušší jazyky.
- jazyky jsou množiny a můžeme s nimi provádět množinové operace - sjednocení, průnik, doplněk, rozdíl
- zřetězení jazyků
  - Zřetězení jazyků  $L_1$  a  $L_2$  označujeme zápisem  $L_1 \cdot L_2$ .

Příklad:

$$\begin{aligned}L_1 &= \{abb, ba\} \\L_2 &= \{a, ab, bbb\}\end{aligned}$$

Jazyk  $L_1 \cdot L_2$  obsahuje slova:

- abba abbab abbbbb baa baab babbb
- zřetězení každý s každým

- mocnina jazyka
  - $L^k$
  - k krát opakování jazyka - zřetězení každý prvek s každým
- Iterace jazyka
  - $L^*$  je jazyk tvořený slovy vzniklými zřetězením libovolného počtu slov z jazyka  $L$ . (sjednocení všech mocnin jazyka  $L$ )
  - Slovo patří do  $L^*$  pokud existuje taková posloupnost  $w_1, w_2, \dots$  že  $w = w_1 w_2 \dots$

**Příklad:**  $L = \{aa, b\}$

  - $L^* = \{\epsilon, aa, b, aaaa, aab, baa, bb, aaaaaa, aaaab, aabaa, aabb, \dots\}$
- zrcadlový obraz
  - je jazyk tvořený zrcadlovými obrazy všech slov jazyka  $L$
  - $L^R$

Formální prostředky pro popis jazyků:

Regulární výrazy

- $\emptyset, \epsilon, a$  (kde  $a \in \Sigma$ ) jsou regulární výrazy:
  - $\emptyset \dots$  označuje prázdný jazyk
  - $\epsilon \dots$  označuje jazyk  $\{\epsilon\}$
  - $a \dots$  označuje jazyk  $\{a\}$
- Jestliže  $\alpha, \beta$  jsou regulární výrazy, pak i  $(\alpha + \beta), (\alpha \cdot \beta), (\alpha^*)$  jsou regulární výrazy:
  - $(\alpha + \beta) \dots$  označuje sjednocení jazyků označených  $\alpha$  a  $\beta$
  - $(\alpha \cdot \beta) \dots$  označuje zřetězení jazyků označených  $\alpha$  a  $\beta$
  - $(\alpha^*) \dots$  označuje iteraci jazyka označeného  $\alpha$
- pomocí těchto operací se definují jazyky

**Příklady:** Ve všech případech  $\Sigma = \{a, b\}$ .

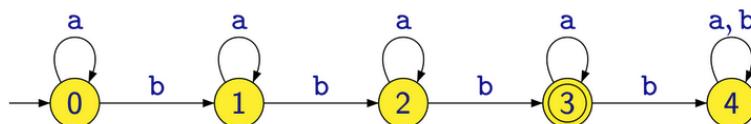
- $a \dots$  jazyk tvořený jediným slovem  $a$
- $ab \dots$  jazyk tvořený jediným slovem  $ab$
- $a + b \dots$  jazyk tvořený dvěma slovy  $a$  a  $b$
- $a^* \dots$  jazyk tvořený slovy  $\epsilon, a, aa, aaa, \dots$
- $(ab)^* \dots$  jazyk tvořený slovy  $\epsilon, ab, abab, ababab, \dots$
- $(a + b)^* \dots$  jazyk tvořený všemi slovy nad abecedou  $\{a, b\}$
- $(a + b)^*aa \dots$  jazyk tvořený všemi slovy končícími  $aa$
- $(ab)^*bbb(ab)^* \dots$  jazyk tvořený všemi slovy obsahujícími podslovo  $bbb$
- předcházené i následované libovolným počtem slov  $ab$

Konečné automaty

- rozpoznávání jazyka
- Deterministický konečný automat se skládá ze stavů a přechodů. Jeden ze stavů je označen jako počáteční stav a některé ze stavů jsou označeny jako přijímající.
- Formálně je deterministický konečný automat (DFA) definován jako pětice  $(Q, \Sigma, \delta, q_0, F)$ 
  - $Q$  je neprázdná konečná množina stavů
  - $\Sigma$  je abeceda (neprázdná konečná množina symbolů)
  - $\delta : Q \times \Sigma \rightarrow Q$  je přechodová funkce
  - $q_0 \in Q$  je počáteční stav

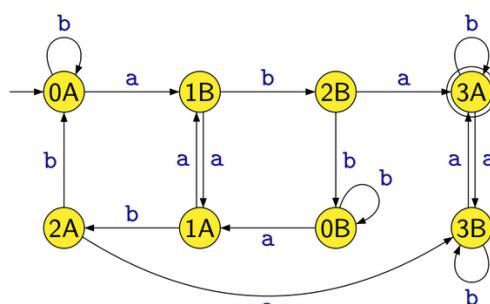
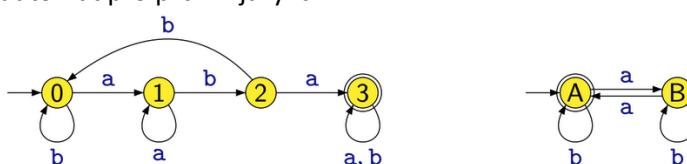
- $F \subseteq Q$  je množina přijímajících stavů
  - slovo je přijímáno pokud se automat po přečtení celého vstupu nachází v přijímajícím stavu, pro každý stav musí být definovány přechody pro všechny symboly abecedy
  - Jazyk  $L$  je regulární právě tehdy, když existuje nějaký deterministický konečný automat  $A$ , který jej přijímá.
  - popsáný grafem stavů a přechodů nebo tabulkou
- Příklad:** Automat rozpoznávající jazyk  $L$  nad abecedou  $\{a, b\}$  tvořený slovy, která obsahují právě tři výskytu symbolu  $b$ , tj.

$$L = \{w \in \{a, b\}^* \mid |w|_b = 3\}$$

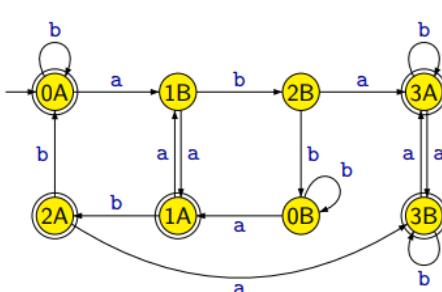


|                 | a | b |
|-----------------|---|---|
| $\rightarrow 0$ | 0 | 1 |
| 1               | 1 | 2 |
| 2               | 2 | 3 |
| $\leftarrow 3$  | 3 | 4 |
| 4               | 4 | 4 |

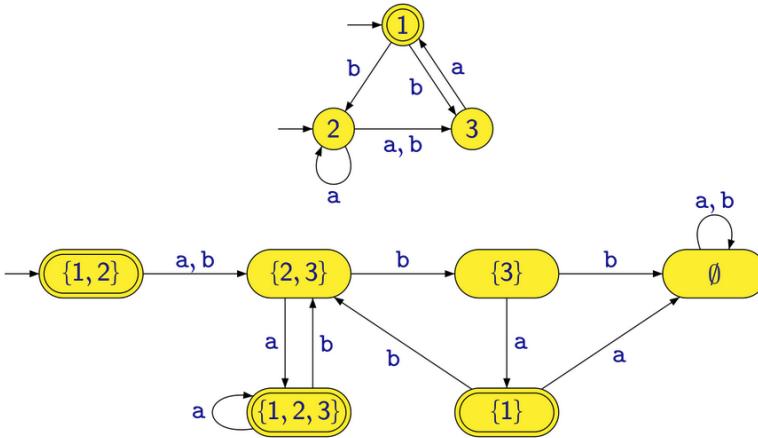
- může mít dosažitelné (existuje přechodová fce do stavu) nebo nedosažitelné stavy (Do nedosažitelných stavů nevede v grafu automatu žádná orientovaná cesta z počátečního stavu. Můžeme je z automatu odstranit)
- pro komplikovanější jazyky obsahující více podmínek může být snazší je konstruovat samostatně a poté je spojit
- automat pro průnik jazyků



- vytvoření nových stavů pro všechny možné přechody
- automat pro sjednocení jazyků - stejný postup akorát přijímající stavy v novém jsou už pokud alespoň jeden z původních je přijímající



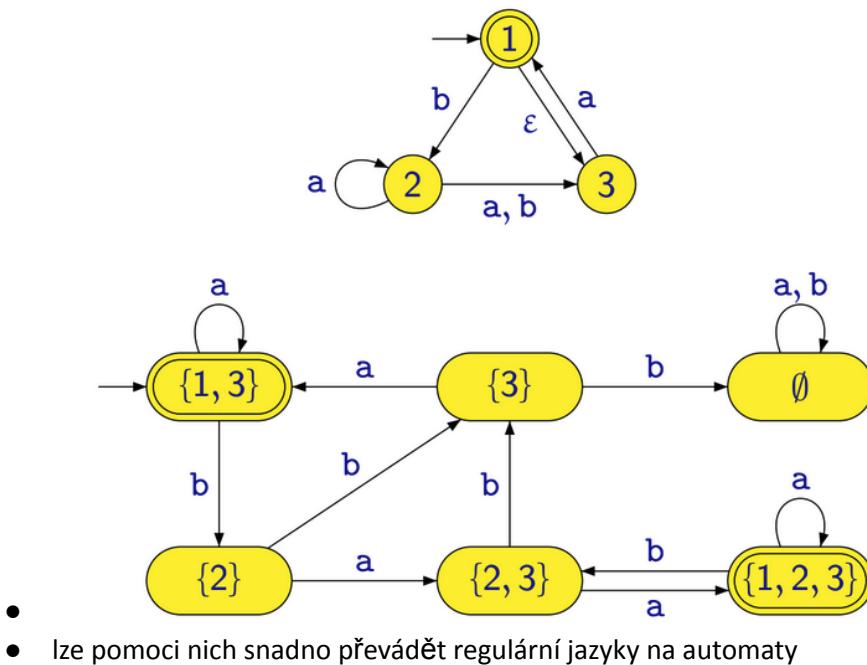
- Nedeterministický konečný automat
- stejný akorát Z jednoho stavu může vézt libovolný (i nulový) počet přechodů označených stejným symbolem a v automatu může být víc než jeden počáteční stav.
- tj. má množinu počátečních stavů ne pouze jeden  $(Q, \Sigma, \delta, I, F)$   $I \subseteq Q$  je množina počátečních stavů
- lze jej převést na deterministický - vzniknou nové stavы



- 
- Při převodu nedeterministického automatu, který má  $n$  stavů, může mít výsledný deterministický automat až  $2^n$  stavů.

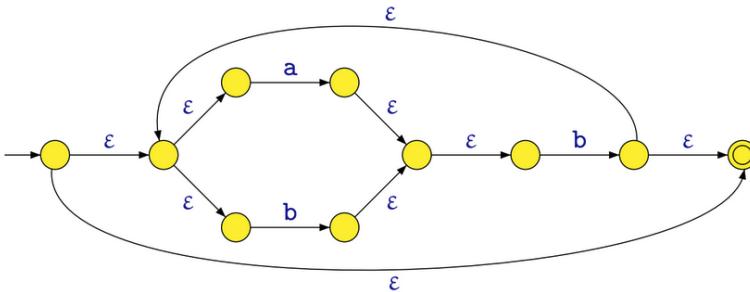
Zobecněný nedeterministický konečný automat ZNKA

- Oproti nedeterministickému konečnému automatu má zobecněný nedeterministický konečný automat tzv.  $\epsilon$ -přechody, tj. přechody označené symbolem  $\epsilon$ .
- Při provádění  $\epsilon$ -přechodu se mění pouze stav řídící jednotky, ale hlava na páscce se neposouvá. - tzn automat se rozhodne jestli jej využije (něco jako by se vytvořila druhá instance automatu a pokračovala s testováním po použití přechodu)
- převod na DKA



- lze pomocí nich snadno převádět regulární jazyky na automaty

**Příklad:** Konstrukce automatu pro výraz  $((a + b) \cdot b)^*$ :



- konečný automat lze převést na regulární výraz
- Jazyk je regulární právě tehdy, když je ho možné popsat regulárním výrazem.

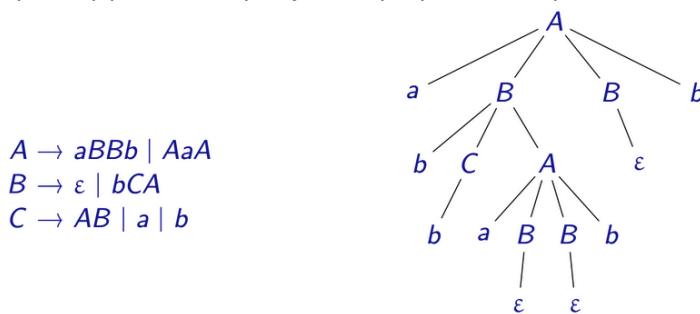
### Bezkontextové gramatiky

- Formálně je bezkontextová gramatika definována jako čtveřice  $G = (\Pi, \Sigma, S, P)$ 
  - $\Pi$  je konečná množina neterminálních symbolů (neterminálů)
  - $\Sigma$  je konečná množina terminálních symbolů (terminálů), přičemž  $\Pi \cap \Sigma = \emptyset$
  - $S \in \Pi$  je počáteční neterminál
  - $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$  je konečná množina přepisovacích pravidel
- slouží ke generování slov
- umožňuje vytvářet např kontrolu závorek - vlevo musí být stejný počet závorek jako vpravo
- neterminály se přepisují na terminály dokud nejsou všechny přepsány

$$\begin{aligned} A &\rightarrow aBBb \mid AaA \\ B &\rightarrow \epsilon \mid bCA \\ C &\rightarrow AB \mid a \mid b \end{aligned}$$

Například slovo *abbabb* je možné v gramatice  $G$  vygenerovat následujícím způsobem:

- $A \Rightarrow aBBb \Rightarrow abC\cancel{A}Bb \Rightarrow abCa\cancel{B}BbBb$
- derivace - vygenerování slova přepsáním neterminálů
- Každá derivaci odpovídá nějaký derivační strom:
  - Vrcholy stromu jsou ohodnoceny terminály a neterminály.
  - Kořen stromu je ohodnocen počátečním neterminálem.
  - Listy stromu jsou ohodnoceny terminály nebo symboly  $\epsilon$ .
  - Ostatní vrcholy stromu jsou ohodnoceny neterminály.
  - Pokud je vrchol ohodnocen neterminálem A, pak jeho potomci jsou ohodnoceni symboly pravé strany nějakého přepisovacího pravidla



- levá derivace - vždy se přepisuje nejlevější neterminál
- pravá derivace - přepisuje se nejpravější

- Každému derivačnímu stromu odpovídá právě jedna levá a právě jedna pravá derivace.
- Gramatika G je nejednoznačná, jestliže existuje nějaké slovo kterému přísluší dva různé derivační stromy, resp. dvě různé levé či dvě různé pravé derivace.

### Zásobníkové automaty

- dokáže rozpoznávat např. správné uzávorkování, ale i vše co bezkontext
- k tomu využívá zásobník do nějž může ukládat symboly a číst je z něj
- může být nedeterministický a může mít  $\epsilon$ -přechody.
- Zásobníkový automat (ZA) je uspořádaná šestice  $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ ,
  - $Q$  je konečná neprázdná množina stavů
  - $\Sigma$  je konečná neprázdná množina zvaná vstupní abeceda
  - $\Gamma$  je konečná neprázdná množina zvaná zásobníková abeceda
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow P(Q \times \Gamma^*)$  je (nedeterministická) přechodová funkce
  - $q_0 \in Q$  je počáteční stav
  - $Z_0 \in \Gamma$  je počáteční zásobníkový symbol
- slovo je přijímáno pokud se končí prázdným zásobníkem

**Příklad:**  $L = \{ w \in \{a, b\}^* \mid w = w^R \}$

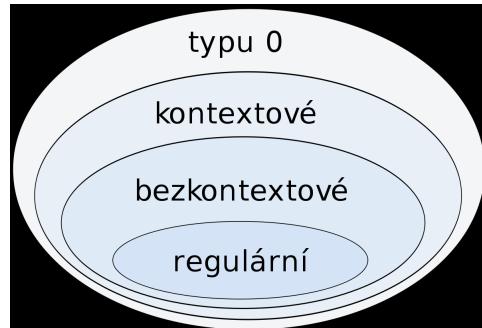
$M = (Q, \Sigma, \Gamma, \delta, q_1, Z)$ , kde

- $Q = \{q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{Z, A, B\}$
- $$\begin{array}{lll} q_1 Z \xrightarrow{a} q_1 AZ & q_1 Z \xrightarrow{b} q_1 BZ & q_2 Z \xrightarrow{\epsilon} q_2 \\ q_1 A \xrightarrow{a} q_1 AA & q_1 A \xrightarrow{b} q_1 BA & q_2 A \xrightarrow{a} q_2 \\ q_1 B \xrightarrow{a} q_1 AB & q_1 B \xrightarrow{b} q_1 BB & q_2 B \xrightarrow{b} q_2 \\ q_1 Z \xrightarrow{a} q_2 Z & q_1 Z \xrightarrow{b} q_2 Z & q_1 Z \xrightarrow{\epsilon} q_2 Z \\ q_1 A \xrightarrow{a} q_2 A & q_1 A \xrightarrow{b} q_2 A & q_1 A \xrightarrow{\epsilon} q_2 A \\ q_1 B \xrightarrow{a} q_2 B & q_1 B \xrightarrow{b} q_2 B & q_1 B \xrightarrow{\epsilon} q_2 B \end{array}$$
- zásobníkové automaty jsou ekvivalentní bezkontextovým gramatikám - Ke každé bezkontextové gramatice G lze sestrojit nedeterministický zásobníkový automat M přijímající prázdným zásobníkem.

### Chomského hierarchie

- Podle tvaru pravidel, která jsou v gramatice povolena, je možné rozdělit gramatiky na následující čtyři typy:
  - Typ 0 — obecné generativní gramatiky pravidla bez omezení
    - Jazyk L je rekurzivně spočetný (či typu 0), jestliže existuje generativní gramatika, která tento jazyk generuje.
  - Typ 1 — kontextové gramatiky pravidla tvaru  $\alpha X \beta \rightarrow \alpha \gamma \beta$ , kde  $|\gamma| \geq 1$ 
    - Jazyk L je kontextový (či typu 1), jestliže existuje kontextová gramatika, která tento jazyk generuje.
  - Typ 2 — bezkontextové gramatiky pravidla tvaru  $X \rightarrow \gamma$ 
    - Jazyk L je bezkontextový (či typu 2), jestliže existuje bezkontextová gramatika, která tento jazyk generuje.
  - Typ 3 — regulární gramatiky pravidla tvaru  $X \rightarrow wY$  (resp.  $X \rightarrow Yw$ ) nebo  $X \rightarrow w$ 
    - Jazyk L je regulární (či typu 3), jestliže existuje regulární gramatika, která

tento jazyk generuje.



- ○ tj. typu 0 jsou všechny jazyky a jednotlivé další jsou jeho podmnožinou
- Příklad jazyka, který je bezkontextový, ale není regulární:  $\{a^n b^n \mid n \geq 1\}$
- Příklad jazyka, který je kontextový, ale není bezkontextový:  $\{a^n b^n c^n \mid n \geq 1\}$
- Příklady jazyků, které jsou typu 0, ale nejsou kontextové:
  - Jazyk tvořený slovy, která reprezentují logicky platné formule predikátové logiky.
  - Jazyk tvořený slovy, která reprezentují kódy těch Turingových strojů, které při výpočtu nad prázdným slovem po konečném počtu kroků zastaví.
- Příklady jazyků, které nejsou typu 0:
  - Jazyk tvořený slovy, která reprezentují právě ty formule predikátové logiky, které nejsou logicky platné.
  - Jazyk tvořený slovy, která reprezentují kódy těch Turingových strojů, které při výpočtu nad prázdným slovem nikdy nezastaví.
  - Jazyk tvořený slovy, která reprezentují kódy těch Turingových strojů, které při výpočtu nad libovolným slovem vždy po konečném počtu kroků zastaví.
- reprezentace jazyků automaty:
- Další možné charakterizace regulárních jazyků:
  - jazyky přijímané konečnými automaty (deterministickými, nedeterministickými, zobecněnými nedeterministickými)
- Další možná charakterizace bezkontextových jazyků:
  - jazyky přijímané nedeterministickými zásobníkovými automaty
- Další možná charakterizace kontextových jazyků:
  - jazyky přijímané nedeterministickými lineárně omezenými automaty
- Další možná charakterizace jazyků typu 0:
  - jazyky přijímané (deterministickými či nedeterministickými) Turingovými stroji

- **Vyčíslitelnost a složitost (výpočetní modely – stroje RAM, Turingovy stroje, výpočetní složitost algoritmů, algoritmicky nerozhodnutelné problémy, třídy složitosti, třídy PTIME, NPTIME a PSPACE, převody mezi problémy, NP-úplné a PSPACE-úplné problémy).**

Výpočetní model je nějaký idealizovaný matematický model počítače. Abstrahuje od nepodstatných implementačních detailů. Cheme analyzovat vlastnosti algoritmů nehledě na technické prostředky stroje, který bude algoritmus provádět. Příklady výpočetních modelů = konečné automaty, zásobníkové automaty, Turingovy stroje, RAM stroje.

prezentace 8

Turingovy stroje

- zařízení podobné konečnému automatu s rozdíly:
  - pohyb hlavy oběma směry
  - možnost zápisu na pásku na aktuální pozici hlavy
  - páska je nekonečná
- Formálně je Turingův stroj definován jako šestice  $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ 
  - $Q$  je konečná neprázdná množina stavů
  - $\Gamma$  (gama) je konečná neprázdná množina páskových symbolů (pásková abeceda)
  - $\Sigma \subseteq \Gamma$  je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
  - $\delta : (Q - F) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$  je přechodová funkce
  - $q_0 \in Q$  je počáteční stav
  - $F \subseteq Q$  je množina koncových stavů
- Konfigurace Turingova stroje je dáná:
  - stavem řídící jednotky
  - obsahem pásky
  - pozicí hlavy
- na začátku a na konci je prázdný znak (značený jako čtverec), který tvoří zarážku pásky na obou koncích
- Turingův stroj provádí kroky tak dlouho, dokud stav řídící jednotky není stav z množiny  $F$  tj. do koncových konfigurací. V nich výpočet končí.
- přijímá slovo právě tehdy když výpočet skončí v přijímajícím koncovém stavu  $q_{acc}$ . Dokáže rozpoznávat jazyk - slova jazyka  $>$  končí ve stavu  $q_{acc}$ , slova která do jazyka nepatří  $>$  končí v nejpřijímacím stavu  $q_{rej}$

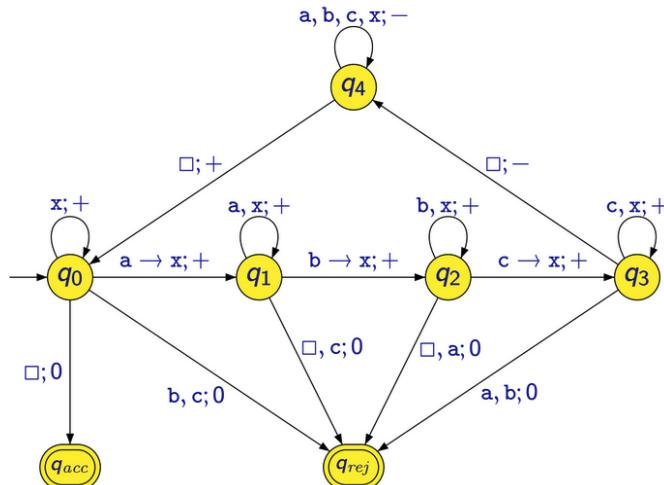
Jazyk  $L = \{a^n b^n c^n \mid n \geq 0\}$

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_{acc}, q_{rej}\} \quad F = \{q_{acc}, q_{rej}\}$$

$$\Sigma = \{a, b, c\} \quad \Gamma = \{\square, a, b, c, x\}$$

| $\delta$ | $\square$               | a                 | b                 | c                 | x              |
|----------|-------------------------|-------------------|-------------------|-------------------|----------------|
| $q_0$    | $(q_{acc}, \square, 0)$ | $(q_1, x, +1)$    | $(q_{rej}, b, 0)$ | $(q_{rej}, c, 0)$ | $(q_0, x, +1)$ |
| $q_1$    | $(q_{rej}, \square, 0)$ | $(q_1, a, +1)$    | $(q_2, x, +1)$    | $(q_{rej}, c, 0)$ | $(q_1, x, +1)$ |
| $q_2$    | $(q_{rej}, \square, 0)$ | $(q_{rej}, a, 0)$ | $(q_2, b, +1)$    | $(q_3, x, +1)$    | $(q_2, x, +1)$ |
| $q_3$    | $(q_4, \square, -1)$    | $(q_{rej}, a, 0)$ | $(q_{rej}, b, 0)$ | $(q_3, c, +1)$    | $(q_3, x, +1)$ |
| $q_4$    | $(q_0, \square, +1)$    | $(q_4, a, -1)$    | $(q_4, b, -1)$    | $(q_4, c, -1)$    | $(q_4, x, -1)$ |

•



- Turingův stroj nemusí dávat jen odpověď Ano nebo Ne, ale může realizovat nějakou funkci, která každému slovu ze  $\Sigma^*$  přiřazuje nějaké jiné slovo (z  $\Gamma^*$ ).

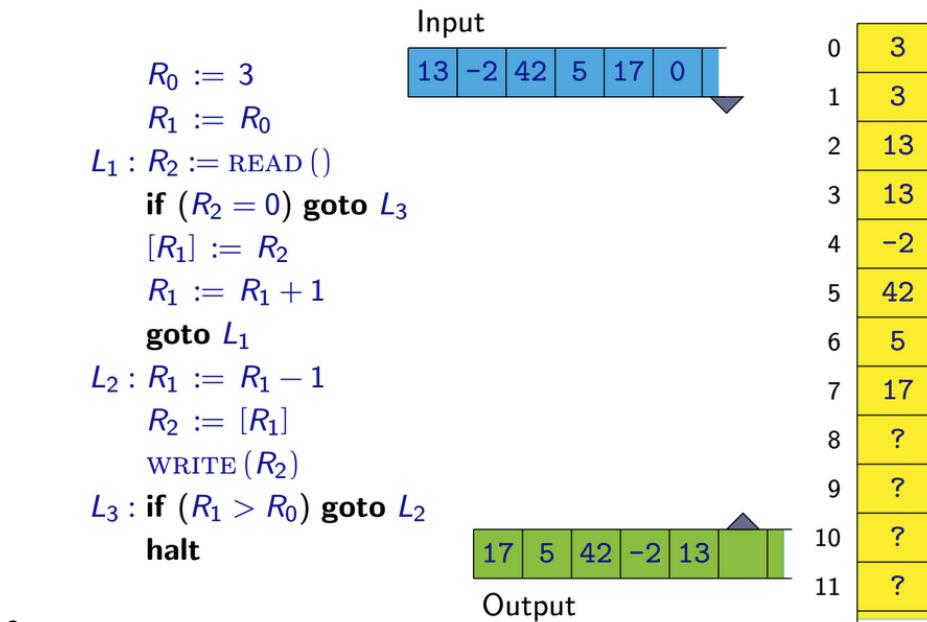
- Slovo přiřazené slovu  $w$  je slovo, které zůstane zapsáno na pásce po výpočtu nad slovem  $w$ , když odstraníme všechny prázdné znaky
- Jsou i nedeterministické turingovy stroje, kde každý stav a symbol může existovat více přechodových funkcí

### RAM stroje

- Stroj RAM (Random Access Machine) je idealizovaný model počítače.
- Skládá se z těchto částí:
  - Programová jednotka – obsahuje program stroje RAM a ukazatel na právě prováděnou instrukci
  - Pracovní paměť tvořená buňkami očíslovanými  $0, 1, 2, \dots$ . Obsah buněk je možno číst i do nich zapisovat
  - Vstupní páska – je z ní možné pouze číst
  - Výstupní páska – je na ni možno pouze zapisovat

Přehled instrukcí:

|                                                                                                         |                                                           |
|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| $R_i := c$                                                                                              | – přiřazení konstanty                                     |
| $R_i := R_j$                                                                                            | – přiřazení                                               |
| $R_i := [R_j]$                                                                                          | – load (čtení z paměti)                                   |
| $[R_i] := R_j$                                                                                          | – store (zápis do paměti)                                 |
| $R_i := R_j op R_k$<br>nebo $R_i := R_j op c$                                                           | – aritmetické instrukce, $op \in \{+, -, *, /\}$          |
| <b>if</b> ( $R_i$ rel $R_j$ ) <b>goto</b> $\ell$<br>nebo <b>if</b> ( $R_i$ rel $c$ ) <b>goto</b> $\ell$ | – podmíněný skok, $rel \in \{=, \neq, \leq, \geq, <, >\}$ |
| <b>goto</b> $\ell$                                                                                      | – nepodmíněný skok                                        |
| $R_i := \text{READ}()$                                                                                  | – čtení ze vstupu                                         |
| $\text{WRITE}(R_i)$                                                                                     | – zápis na výstup                                         |
| <b>halt</b>                                                                                             | – zastavení programu                                      |



- Rozdíly oproti skutečnému počítači:

- Velikost paměti není omezena
- Velikost obsahu jednotlivých buněk není omezena
- Čte data sekvenčně ze vstupu, který je tvořen sekvencí celých čísel. Ze vstupu lze pouze číst.
- Zapisuje data sekvenčně na výstup, který je tvořen sekvencí celých čísel. Na výstup je možné pouze zapisovat.
- Pokud má stroj jako výsledek dát jen odpověď Ano/Ne (tj. přijmout nebo nepřijmout daný vstup), nemusí mít výstupní pásku. Instrukce halt je pak nahrazena instrukcemi accept a reject.

### Výpočetní složitost algoritmů

- Časová složitost algoritmu — jak závisí doba výpočtu na množství vstupních dat
- Paměťová (resp. prostorová) složitost algoritmu — jak závisí množství použité paměti na množství vstupních dat
- Pro různé vstupy provede program různý počet instrukcí.
- Pro zjednodušení se analyzuje doba vykonání pouze v závislosti na velikosti vstupu ne na typu dat a funkce vyjadřující, jak roste doba výpočtu nebo množství použité paměti v závislosti na velikosti vstupu, se nepočítají přesně — počítají se odhady těchto funkcí.
- Nepočítá se přesná doba vykonání instrukce, ale zapisuje se jako asymptotická notace

### Asymptotická notace

Vezměme si libovolnou funkci  $g : \mathbb{N} \rightarrow \mathbb{R}$ . Zápis  $O(g)$ ,  $\Omega(g)$ ,  $\Theta(g)$ ,  $o(g)$  a  $\omega(g)$  označují **množiny funkcí** typu  $\mathbb{N} \rightarrow \mathbb{R}$ , kde:

- $O(g)$  – množina všech funkcí, které rostou nejvýše tak rychle jako  $g$
- $\Omega(g)$  – množina všech funkcí, které rostou alespoň tak rychle jako  $g$
- $\Theta(g)$  – množina všech funkcí, které rostou stejně rychle jako  $g$
- $o(g)$  – množina všech funkcí, které rostou pomaleji než funkce  $g$
- $\omega(g)$  – množina všech funkcí, které rostou rychleji než funkce  $g$

**Poznámka:** Toto nejsou definice! Ty následují na následujících slidech.

- $O$  – velké „O“
- $\Omega$  – velké řecké písmeno „omega“
- $\Theta$  – velké řecké písmeno „theta“
- $o$  – malé „o“
- $\omega$  – malé „omega“
- analyzuje se kód algoritmu a sleduje se kolikrát se operace provede. Hlavně se řeší cykly např 2 cykly v sobě jdoucí od 0 O ( $n^2$ ), je ale potřeba sledovat iterační proměnné v cyklech a jak se mění. např pokud se proměnná po průchodu i\*i tak má cyklus logaritmickou složitost
- nepočítá se přesně, zaokrouhuje se na některou z asymptotických notací

- problémy které nejsou algoriticky řešitelné
- např.
- Problém zastavení (Halting problem)
  - Vstup: Zdrojový kód programu P v jazyce L, vstupní data x.
  - Zastaví se program P po nějakém konečném počtu kroků, pokud dostane jako vstup data x?
  - podprogram co by to rozhodoval. problém pokud by podprogram dostal svůj vlastní zdrojový kód
- postův korespondenční problém - generování stejných slov z podslov kartiček

### převody mezi problémy

- Pokud máme o nějakém (rozhodovacím) problému dokázáno, že je nerozhodnutelný, můžeme ukázat nerozhodnutelnost dalších problémů pomocí redukcí (převodů) mezi problémy.
- Problém P1 je převeditelný na problém P2, jestliže existuje algoritmus Alg takový, že:
  - Jako vstup může dostat libovolnou instanci problému P1. K instanci problému P1, kterou dostane jako vstup, vyprodukuje jako svůj výstup instanci problému P2. Pro stejný vstup musí v obou problémech stejný výstup.
- Pokud P2 je rozhodnutelný tak i P1 je rozhodnutelný

### třídy složitosti

- různé (algoritmické) problémy jsou různě těžké.
- Obtížnější jsou ty problémy, k jejichž řešení potřebujeme více času a paměti.
- Obtížnost problémů chceme nějak posuzovat,
  - absolutně – kolik času a kolik paměti potřebujeme k jejich řešení,
  - relativně – o kolik je jejich řešení obtížnější nebo naopak jednodušší oproti jiným problémům.
- složitost algoritmu — funkce, která vyjadřuje, jaká bude pro daný algoritmus maximální doba výpočtu pro vstup velikosti n
- složitost problému — jaká je časová složitost „nejfektivnějšího“ algoritmu, který řeší daný problém
- Pojem „složitost problému“ se jako takový nedefinuje (bylo by obtížné), ale obchází se zavedením tzv. tříd složitosti.
- Třídy složitosti jsou podmnožiny množiny všech (algoritmických) problémů.
- Daná konkrétní třída složitosti je vždy charakterizována nějakou vlastností, kterou mají problémy do ní patřící. např. pro daný problém existuje algoritmus s nějakým omezením - časové nebo prostorové složitosti
- $T(n)$  – třída všech rozhodovacích problémů pro něž existuje algoritmus s časovou složitostí  $O(n)$
- $T(n^2)$  – třída všech rozhodovacích problémů pro než existuje algoritmus s časovou složitostí  $O(n^2)$
- $T(n \log n)$  – třída všech rozhodovacích problémů pro než existuje algoritmus s časovou složitostí  $O(n \log n)$
- PTIME je třída všech rozhodovacích problémů, pro které existuje algoritmus s polynomiální časovou složitostí, tj. s časovou složitostí  $O(n^k)$ , kde k je nějaká konstanta.
- PSPACE je třída všech rozhodovacích problémů, pro které existuje algoritmus s polynomiální prostorovou složitostí, tj. s prostorovou složitostí  $O(n^k)$ , kde k je nějaká

konstanta.

- LOGSPACE – množina všech rozhodovacích problémů, pro které existuje algoritmus s prostorovou složitostí  $O(\log n)$
- NPTIME (někdy se píše jen NP) je třída všech problémů, pro které existuje nedeterministický algoritmus s polynomiální časovou složitostí. Patří zde problémy, u kterých je možné pro daný vstup rychle ověřit, že odpověď je Ano, pokud nám ten, kdo nás o tom chce přesvědčit, dodá nějakou dodatečnou informaci.

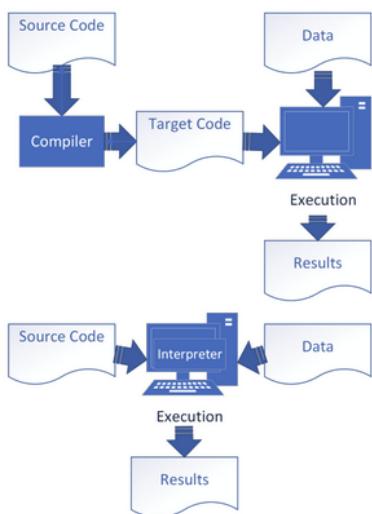
#### NP-úplné a PSPACE-úplné problémy

- NP-úplné problémy,
  - patří do třídy NPTIME, tj. jsou řešitelné v polynomiálním čase nedeterministickým algoritmem
  - jsou tedy řešitelné deterministickým algoritmem v exponenciálním čase
  - není pro ně znám žádný deterministický algoritmus s polynomiální časovou složitostí
  - NP težký problém se stane NP-úplným jakmile se pro něj najde nedeterministický algoritmus který jej řeší v polynomiálním čase
  - Pokud bychom pro nějaký NP-težký problém  $P$  nalezli polynomiální algoritmus, získali bychom tím polynomiální algoritmus pro každý problém  $P'$  z NPTIME:
  - Na druhou stranu, pokud existuje alespoň jeden problém z NPTIME, pro který neexistuje polynomiální algoritmus, tak z předchozího plyne, že pro žádný NP-težký problém nemůže existovat polynomiální algoritmus.
  - Jedna z těchto možností platí - dosud to je otevřený problém
  - tj. je to skupina algoritmů které dosud nemají v polynomiálním čase známé řešení. Pokud by se nalezlo řešení jednoho z nich existuje řešení pro všechny. ASI
  - problémy: SAT - splnitelnost booleovských formulí, IS - problém nezávislé množiny, vrcholové barvení grafu, problém hamiltonovského cyklu, obchodní cestující ..
- PSPACE-úplné problémy
  - problém  $P'$  z PSPACE polynomiálně převeditelný na problém  $P$  a navíc sám patří do třídy PSPACE.

- **Překladače (základní funkce a typy překladačů, fáze při překladu, formální prostředky využívané při tvorbě překladače, speciální podtřídy bezkontextových gramatik jako např. LL1 a regulární výrazy, techniky používané při implementaci jako např. rekurzivní sestup, generátory překladačů).**

programovací jazyk - Programovací jazyk je strojově čitelný umělý jazyk určený k vyjádření výpočtů, které může provádět stroj, zejména počítač. "Jazyk pro komunikace člověka a PC". Programovací jazyky by měly být schopny implementovat libovolný algoritmus tj. měly by být turingovsky kompletní.

- Hlavní funkcí překladače je překlad zdrojového jazyka do cílového jazyka.
  - Nejčastěji je zdrojovým jazykem vysokoúrovňový programovací jazyk čitelný pro člověka (c,c++,c#, java,...) a cílovým jazykem je pak nízkoúrovňový jazyk srozumitelný počítači.
  - Existují i další typy překladačů, které například: extrahuje některá data ze zdrojových kódů (Doxygen), generují dokumenty ze zdrojových kódů (Latex), pracují s daty (XSLT)
- překladače dále realizují
  - řešení a hlášení chyb
  - optimalizace
  - pomocné funkce např. debugging
- Cílovým jazykem nejčastěji bývá
  - strojový kód
  - objektový kód - moduly ve strojovém kódu (knihovny)
  - bytecode, intermediate language, portable code - kód assembleru nebo strojový kód pro virtuální stroj
- 2 typy jazyků:
- komplikované
  - přeloží se celý program a až poté jej lze spustit
  - rychlejší - lepší optimalizace, nekomplikuje se za běhu
- Interpretované
  - překlad probíhá až za běhu, přeloží se příkaz a rovnou se vykoná (řeší interpret)
  - platformě nezávislejší a umožňuje modifikace za běhu
  - využíváno pro skriptování a webové jazyky



- způsoby komplikace:
- ahead of time AOT - program je před jeho spuštěním překládán do strojového kódu
- just in time JIT - AOT + interpretace
  - provádí překlad na strojový kód během vykonávání programu
  - překlad na strojový kód bývá realizován až z intermediate jazyka, který vznikl překladem z původního jazyka
- jednoprůchodové - výstupem strojový kód C
- víceprůchodové - intermediate language > JIT překládá na strojový kód C#

#### Fáze překladu

- lexikální analýza - vytvoření posloupnosti tokenů (převod textu na tokeny, token má typ a hodnotu)

- syntaktická analýza - vytváří se stromová hierarchie - abstract syntax tree, využívá se bezkontextová gramatika
- sémantická analýza - odchytává problémy které nelze zjistit bezkontextovými jazyky, přidává sémantickou informaci do stromu, zajišťuje dodržení pravidel jazyka (datové typy, definice, proměnné)
- převod do intermediate reprezentace (trojstavový kód, zásobníkový kód)
- optimalizace
- generování cílového kódu

formální prostředky využívané při tvorbě překladače

- jazyky jsou tvořeny
- abecedou - uzavřená množina symbolů
- slovy z abecedy, slova tvoří jazyk
- jazyk může být popsán gramatikou (bezkontextovou, kontextovou, regulární jazyky) - generativní systémy nebo automaty - detekční systémy

speciální podtřídy bezkontextových gramatik jako např. LL1 a regulární výrazy

- LL1
- gramatika, která když tvořím levou derivaci lze rozhodovat deterministicky, které pravidlo se použije, rozhoduje se podle 1 nahlíženého symbolu ze vstupu
- Pro rozhodování se vytvářejí množiny First a Follow
- First
  - hledá všechny terminální symboly, které se mohou objevit na začátku pravidla
  - do množiny first přidávám terminální symboly, pokud lze neterminál přepsat na epsilon jdu na další pokud ne konec
  - poznačím co začíná terminálem
  - řeším neterminály - pokud nejde přepsat na epsilon vezmu jeho terminály a končím, pokud lze i poslední neterm. přepsat na epsilon přidám ho tam taky
- Follow
  - množina symbolů, které se mohou vyskytovat za zadaným symbolem
  - využívá se pro rozhodování, které pravidlo využít
  - hledají se výskyty neterminálu a do množiny follow daného neterminálu se přidává: first symbolů které se vyskytují za neterminálem, pokud first obsahuje epsilon (tj. i pokud se hledá prázdný first) tak se do množiny přidá follow levé strany odkud se pravidlo vzalo
- aby byla LL1 musí nesmí gramatika obsahovat
  - levou rekurzi tj. Neterminál levé strany se vyskytuje na začátku v pravé straně
 

Removing left recursion

From:  $A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$

To:  $A \rightarrow \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \epsilon$
  - a levý faktor tj. v různých pravidlech stejné počáteční terminály (stejné množiny first)

Left factorization (removing common prefix)

From:  $A \rightarrow \beta\alpha_1 | \beta\alpha_2 | \dots | \beta\alpha_n$

To:  $A \rightarrow \beta A'$

$A' \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$
- aby bylo LL1 musí být průnik firstů pravidel prázdný
- Regulární výrazy se využívají pro generování slov jazyka

- ideální je kombinace bezkontextové a regulární > backus-nař form
  - zakomponování regulárních výrazů do bezkontextové gramatiky
  - Jedná se o metasyntaktický zápis pro bezkontextové gramatiky.
  - Často se používá k definici syntaxe programovacích jazyků.

techniky používané při implementaci jako např. rekurzivní sestup

- rekurzivní sestup - technika pro implementaci LL1 parseru - syntaktická analýza
  - Množina vzájemně rekurzivních procedur, kde každá taková procedura implementuje jeden z neterminálů gramatiky.
  - může být implementováno pomocí volání funkcí jednotlivých neterminálů vyhodnocováno zleva (backtracking)
  - Každý neterminál je reprezentován funkcí, která provádí jeho analýzu.
  - testuje se jestli se jedná o očekávaný token - pokud ne chyba

generátory překladačů

- existují nástroje pro implementaci parseru
- vstupem je většinou definice gramatiky v BNF (backus-nař form) a výstupem program ve zvoleném/podporovaném programovacím jazyce (realizuje např. rekurzivní sestup)
- nástroje - ANTLR4, JavaCC, Coco, GNU Bison
- nástrojem vygenerovaný programu lze rozšiřovat o vlastní funkčnost
- výhodou je úspora času, díky nástrojům není nutno implementovat parser

**Příklad otázky:** Bezkontextové gramatiky a jejich souvislost se zásobníkovými automaty. Využití bezkontextových gramatik a zásobníkových automatů při konstrukci překladačů.

## Matematika pro informatiku (MA I, MA II, LA, DIM)

- Diskrétní matematika (posloupnosti, rekurentní rovnice, kombinatorické výběry, diskrétní pravděpodobnost, řešení kongruencí).

### Posloupnosti (DIM)

- Posloupnost je seřazení několika prvků. Může být konečná nebo nekonečná
- prvky posloupnosti se mohou opakovat (na rozdíl od množin)
- posloupnost může být i prázdná

### Aritmetická posloupnost

- má tvar:  $a, a+d, a+2d, a+3d, \dots$ 
  - $a$  je počáteční člen
  - $d$  je diference posloupnosti
- každý další člen vznikne z předchozího čísla přičtením stejné diference

#### Příklady

- 2, 3, 8, 13, 18, …      první člen –2, diference 5
- 3, 2, 7, 12, 17, …      první člen –3, diference 5
- 20, 9, –2, –13, –24, …      první člen 20, diference –11
- $\sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \dots$       první člen  $\sqrt{2}$ , diference 0

#### Příklad

Sestavte vztahy pro  $n$ -tý člen posloupnosti  $a_n$  z předchozího příkladu

$$-2, 3, 8, 13, 18, \dots \quad a_n = -2 + (n-1)5$$

$$-3, 2, 7, 12, 17, \dots \quad a_n = -3 + (n-1)5$$

$$20, 9, -2, -13, -24, \dots \quad a_n = 20 - (n-1)11$$

$$\sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \dots \quad a_n = \sqrt{2}$$

- součet  $n$  členů posloupnosti

$$a_1 + a_2 + \dots + a_n = \sum_{i=1}^n a_i$$

$$\sum_{i=1}^n (a_1 + (i-1)d)$$

$$na_1 + \frac{n(n-1)d}{2}$$

- může být nekonečná
- součet nekonečné aritmetické posloupnosti neexistuje. Lze určit jestli diverguje k  $+\infty$  pokud  $d > 0$  nebo  $-\infty$   $d < 0$
- aritmetickou posloupnost lze zadat rekurentně tj. prvním členem a diferencí  $a_n = a_{n-1} + d$ ,  $a_1 = a$

### Příklad spojení

#### Příklad

Strýček Skrblík má v sejfu 4 514 centů. Každý týden ukládá do sejfu 24 centů. Sestavte vztah pro  $n$ -tý člen posloupnosti.

$$4\ 514, 4\ 538, 4\ 562, 4\ 586, \dots = 4\ 514 + 24(n - 1) = 4\ 490 + 24n.$$

#### Příklad

Strýček Skrblík má v sejfu 4 514 centů. Třem synovcům dá každému kapesné jeden cent a každý týden každému kapesné o cent zvýší.

- a) Sestavte vztah pro celkovou výši kapesného v  $n$ -tém týdnu.
- b) Sestavte vztah pro počet centů v sejfu za  $n$  týdnů.

a) kapesné  $k = 3 + 3(n - 1) = 3n$

• b) v sejfu  $s = 4\ 514 - 3n$

#### Geometrická posloupnost

- má tvar  $a, a \cdot q, a \cdot q^2, a \cdot q^3$
- reálné číslo  $a$  je počáteční člen a reálné číslo  $q$  je kvocient posloupnosti
- každý další člen vznikne vynásobením předchozího kvocientem

#### Příklady

2, 10, 50, 250, 1250, ... první člen 2, kvocient 5

9, 6, 4,  $\frac{8}{3}$ ,  $\frac{16}{9}$ , ... první člen 9, kvocient  $\frac{2}{3}$

4, -2, 1,  $-\frac{1}{2}$ ,  $\frac{1}{4}$ , ... první člen 4, kvocient  $-\frac{1}{2}$

$\sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \dots$  první člen  $\sqrt{2}$ , kvocient 1

#### Příklad

Sestavte vztahy pro  $n$ -tý člen  $a_n$  posloupností z předchozího příkladu.

2, 10, 50, 250, 1250, ...  $a_n = 2 \cdot 5^{n-1}$

9, 6, 4,  $\frac{8}{3}$ ,  $\frac{16}{9}$ , ...  $a_n = 9 \cdot \left(\frac{2}{3}\right)^{n-1} = \frac{27}{2} \cdot \left(\frac{2}{3}\right)^n$

4, -2, 1,  $-\frac{1}{2}$ ,  $\frac{1}{4}$ , ...  $a_n = 4 \cdot \left(-\frac{1}{2}\right)^{n-1} = -8 \cdot \left(-\frac{1}{2}\right)^n$

•  $\sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \sqrt{2}, \dots$   $a_n = \sqrt{2}$

- součet  $n$  členů posloupnosti

○ 
$$\sum_{i=1}^n (a_1 \cdot q^{i-1})$$

○ 
$$a_1 \frac{q^n - 1}{q - 1}$$

- součet nekonečné geometrické posloupnosti

- neexistuje pro  $q >= 1$
- pro  $q=1$  - konstantní posloupnost součet závisí na  $a_1$
- pro  $q=-1$  - oscilující posloupnost, součet nemá
- pro  $q < 1$  konečný součet  $a_1 / (1-q)$

### Příklad spojení

#### Příklad

Strýček Skrblík má v bance uloženo 4 514 centů. Každý rok mu úspory zúročí dvěma procenty (bez zaokrouhlení). Sestavte vztah pro částku za  $n$  let.

$$4\ 604.3, 4\ 696.4, 4\ 790.3, 4\ 886.1, 4\ 983.8, \dots = 4\ 514 \cdot 1.02^{n-1}.$$

#### Příklad

Strýček Skrblík má v sejfu 4 514 centů. Třem synovcům dá každému kapesné jeden cent a každý týden každému kapesné zdvojnásobí.

- a) Sestavte vztah pro celkovou výši kapesného v  $n$ -tému týdnu.
- b) Sestavte vztah pro počet centů v sejfu za  $n$  týdnů.

- a) kapesné  $k = 3 \cdot 2^{n-1} = \frac{3}{2} \cdot 2^n$
- b) v sejfu  $s = 4\ 514 - 3 \cdot 2^{n-1}$

### Rekurentní rovnice

- rekurentně zadané posloupnosti, tj. vztah pro  $n$ -tý člen
- výpočet  $n$  tého prvku posloupnosti bez znalosti znalostí předchozích prvků. tj. rovnice do které dosadíme který  $n$  tý prvek chceme a vypočteme hodnotu

#### Příklad

Najděte řešení rekurentní rovnice  $a_n = a_{n-1} + 2a_{n-2}$ , kde  $a_0 = 2$ ,  $a_1 = 7$

Postupujeme podle návodu uvedeného dříve:

Předpokládáme řešení tvaru  $a_n = r^n$ . Dosazením do rekurentní rovnice dostaneme charakteristickou rovnici

$$\begin{aligned} r^2 - r - 2 &= 0 \\ (r+1)(r-2) &= 0. \end{aligned}$$

Charakteristické kořeny jsou  $r_1 = 2$ ,  $r_2 = -1$ . Obecné řešení je tvaru

$$a_n = \alpha_1 2^n + \alpha_2 (-1)^n.$$

Dosazením  $a_0$ ,  $a_1$  dostaneme dvě rovnice o neznámých  $\alpha_1$ ,  $\alpha_2$ .

$$\begin{aligned} a_0 = 2 &= \alpha_1 \cdot 1 + \alpha_2 \cdot 1 \\ a_1 = 7 &= \alpha_1 \cdot 2 + \alpha_2 \cdot (-1) \end{aligned}$$

Řešením soustavy dostaneme  $\alpha_1 = 3$ ,  $\alpha_2 = -1$ , proto obecné řešení je

- $a_n = 3 \cdot 2^n - 1 \cdot (-1)^n$ .
- charakteristická rovnice - využívá se k řešení lineárních homogenních rekurentních rovnic, konstantními koeficienty se převedou do tvaru  $a_n = r^n$  kde  $r$  je vhodná konstanta.  $r$  je hledaná konstanta tj. kořeny charakteristické rovnice

## Kombinatorické výběry

- výběr prvků nějaké množiny
- permutace, kombinace, variace
- permutace bez opakování
  - $P(n)$
  - libovolné uspořádání všech prvků množiny do nějaké posloupnosti
  - počet všech permutací n prvkové množiny  $P(n) = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1 = n!$
  - úlohy: počet uspořádání nějaké množiny, počet bijektivních zobrazení, počet různých rozmíchání karet, přidělení startovních čísel, počet různých obsazení hotelových pokojů
- kombinace bez opakování
  - libovolný neuspořádaný výběr k prvkům z n prvkové množiny
  - $C(n,k)$
- $C(n,k) = \frac{n!}{k! \cdot (n-k)!} = \binom{n}{k}$
- počet výběrů
- výběr je k prvková podmnožina
- úlohy: počet podmnožin s předepsaným počtem prvků
- variace bez opakování
  - libovolný uspořádaný výběr k prvkům z n prvkové množiny - záleží na pořadí
  - $V(n,k)$
- $V(n,k) = n \cdot (n - 1) \cdots (n - k + 1) = \frac{n!}{(n - k)!}$
- počet výběrů
- úlohy: výběr k prvkové posloupnosti z n prvků, počet možných pořadí prvních tří vítězů závodu

### Příklady

- čtyřčlenný tým z deseti zaměstnanců  
jedná se o *kombinace* (nezávisí na pořadí vybraných zaměstnanců)  
$$C(10, 4) = \binom{10}{4} = \frac{10!}{4! \cdot 6!} = \frac{10 \cdot 9 \cdot 8 \cdot 7}{4 \cdot 3 \cdot 2} = \frac{10 \cdot 3 \cdot 7}{1} = 210$$
- počet zápasů tenisového turnaje sedmi hráčů  
jedná se o *kombinace* (dvouprvkové podmnožiny sedmi prvkové množiny)  
$$C(7, 2) = \binom{7}{2} = 21$$
- počet výsledných pořadí tenisového turnaje sedmi hráčů  
jedná se o *permutace*  
$$P(7) = 7! = 5040$$
- počet trojic na stupni vítězů tenisového turnaje sedmi hráčů  
jedná se o 3-prvkové *variace*, neboť „závisí na pořadí“  
$$V(7, 3) = \frac{7!}{4!} = 7 \cdot 6 \cdot 5 = 210$$

- kombinatorické pravidlo součtu a součinu - více výběrů a buď součet nebo součin podle toho jestli na sobě výběry závisí dim prez2 str11

### Příklad

Hokejový trenér sestavuje formaci (tři útočníky, dva obránce a jednoho brankáře). K dispozici má 12 útočníků, 8 obránců a dva brankáře.

Kolik různých formací lze sestavit?

Protože mezi výběrem útočníků, výběrem obránců ani výběrem brankářů není žádná vazba, můžeme počítat

$$\binom{12}{3} \cdot \binom{8}{2} \cdot \binom{2}{1} = \frac{12 \cdot 11 \cdot 10}{6} \cdot \frac{8 \cdot 7}{2} \cdot 2 = 220 \cdot 28 \cdot 2 = 12320.$$

- pravidlo součinu,

součet by byl např opakované hody kostkou

- výběry s opakováním
- Permutace s opakováním
  - libovolné uspořádání prvků množiny, každý prvek se v této množině vyskytuje předepsaný počet krát.
  - $P^*(m_1, m_2, \dots, m_n)$ .  

$$P^*(m_1, m_2, \dots, m_n) = \frac{(m_1 + m_2 + \dots + m_n)!}{m_1! \cdot m_2! \cdots m_n!}$$
  - počet výběrů
    - kde  $m_i$  označuje kolikrát se může daný prvek opakovat
- Kombinace s opakováním
  - neuspořádaný výběr k prvků, prvky se mohou opakovat v libovolném počtu identických kopií
  - $C^*(n, k)$   

$$C^*(n, k) = \binom{k+n-1}{n-1}$$
  - počet výběrů
  - úlohy: losování n prvků kdy po losování vracíme do osudí
- Variace s opakováním
  - k prvková posloupnost vybírána z n prvkové množiny, každý prvek se v posloupnosti může opakovat libovolný počet krát
  - $V^*(n, k)$   

$$V^*(n, k) = \underbrace{n \cdot n \cdots n}_k = n^k$$
  - počet výběrů

### Diskrétní pravděpodobnost

- pravděpodobnost že nastane nějaký jev
- konečný pravděpodobnostní prostor je dvojice  $(\Omega, P)$ , tj. množina elementárních jevů a funkce pravděpodobnosti
- Náhodný jev je libovolná podmnožina elementárních jevů a pravděpodobnost jevu je  $P(A)$
- $P(A) = |A| / |\Omega|$
- tj. počet hledaných jevů / počet všech jevů
- pravděpodobnost může být nulová např. že bude číslo 1 a zároveň sudé číslo

## Příklad

Náhodný proces, kdy zjišťujeme součet ok při hodu dvěma kostkami.

Množina všech možných součtů dvou kostek je  $\Omega = \{2, 3, \dots, 12\}$ .

**Pravděpodobnosti elementárních jevů jsou různé!** Součet 2 lze získat jediným způsobem  $1 + 1$ , součet 7 lze získat šesti způsoby, bude častější.

Celkem je  $6 \cdot 6 = 36$  možností hodů dvou kostek. Z toho, jak jsme si všimli, součet 7 padne šesti způsoby, součet 6 pěti způsoby, atd. Užitím úvahy o poměrném počtu způsobů získáme funkci pravděpodobnosti:

$$\begin{aligned}P(2) &= P(12) = \frac{1}{36}, \\P(3) &= P(11) = \frac{2}{36} = \frac{1}{18}, \\P(4) &= P(10) = \frac{3}{36} = \frac{1}{12}, \\P(5) &= P(9) = \frac{4}{36} = \frac{1}{9}, \\P(6) &= P(8) = \frac{5}{36}, \\P(7) &= \frac{6}{36} = \frac{1}{6}.\end{aligned}$$

- Doplňkový náhodný jev  $A^- = \Omega \setminus A$  tj. pravděpodobnost že jev nenastane platí  $P(\bar{A}) = 1 - P(A)$ .
- Podmíněná pravděpodobnost jevu A jevem B
  - je pravděpodobnost jevu A pokud víme že nastal jev B.
  - $P(A|B)$

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

## Příklad

Jaká je pravděpodobnost, že při současném hodu dvěma kostkami padl součet 7 víme-li, že na některé kostce padlo číslo 5.

Snadno určíme  $P(A \cap B) = \frac{2}{36}$  (5+2 nebo 2+5).

Při určení  $P(B)$  nesmíme možnost 5+5 započítat dvakrát, a určíme  $P(B) = \frac{11}{36}$ .

$$\text{Hledaná podmíněná pravděpodobnost je } P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{\frac{2}{36}}{\frac{11}{36}} = \frac{2}{11}$$

- Závislé a nezávislé jevy
  - pravděpodobnost toho, že nastane jeden jev je/není ovlivněna výsledkem druhého jevu
  - Nezávislé jevy
    - pravděpodobnost jevu A při současném jevu B je stejná jako pravděpodobnost A obecně
    - $P(A \cap B) = P(A) \cdot P(B)$
    - dva hody toutéž kostkou za sebou
    - jeden hod více kostkami
    - postupné losování s vrácením do osudí
  - Závislé jevy
    - vrchní a spodní číslo padlé při hodu kostkou
    - postupné losování z osudí
- střední hodnota
  - průměrná hodnota výskytu číselných jevů

### Příklad

Jaká je střední hodnota (průměr) čísel padlých na šestistěnné kostce?

$$E(K) = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = \frac{21}{6} = 3.5.$$

### Příklad

Jaká je střední hodnota čísel padlých na kostce, kde číslo 6 má dvakrát větší pravděpodobnost výskytu než ostatní?

$$E(K) = \frac{1}{7} \cdot 1 + \frac{1}{7} \cdot 2 + \frac{1}{7} \cdot 3 + \frac{1}{7} \cdot 4 + \frac{1}{7} \cdot 5 + \frac{2}{7} \cdot 6 = \frac{27}{7} \doteq 3.8571.$$

### Příklad

V kapse je dvacetikoruna, pětikoruna a koruna. Náhodně jednu minci vytáhneme. Větší mince má vždy dvakrát větší šanci být vytažena než menší mince. Jaká je střední hodnota vytažené mince  $M$ ?

$$p_1 = \frac{1}{7}, \quad p_5 = \frac{2}{7}, \quad p_{20} = \frac{4}{7}, \quad E(M) = \frac{1}{7} \cdot 1 + \frac{2}{7} \cdot 5 + \frac{4}{7} \cdot 20 = \frac{91}{7} = 13.$$

## Řešení kongruencí

- mějme celá čísla  $a, b$  a přirozené číslo  $m$ . Čísla  $a, b$  jsou konguentní modulo  $m$ , jestliže dávají stejný zbytek po dělení číslem  $m$
- $a \equiv b \pmod{m}$
- využití v kryptografii

### Příklad

Platí  $7 \equiv 1 \pmod{2}$ , protože číslo 7 je liché.

Platí  $12 \equiv 0 \pmod{2}$ , protože číslo 12 je sudé.

- Platí  $61\ 725 \equiv 0 \pmod{3}$ , protože 61 725 je násobek čísla 3.
- kongruence se stejným modulem můžeme sčítat i násobit
- Jestliže platí  $a \equiv b \pmod{m}$ ,  $c \equiv d \pmod{m}$ , potom platí také  $a + c \equiv b + d \pmod{m}$ ,  $ac \equiv bd \pmod{m}$ .

Protože  $7 \equiv 12 \pmod{5}$  a  $-7 \equiv 3 \pmod{5}$ , tak také

- $7 - 7 \equiv 12 + 3 \pmod{5}$  a  $7 \cdot (-7) \equiv 12 \cdot 3 \pmod{5}$ .

- V kongruencích můžeme krátit číslem nesoudělným s modulem  $m$

- Vyřešit kongruenci znamená najít všechny hodnoty proměnné  $x$ , pro které je kongruence splněna

### Příklad

Řešením kongruenze  $x \equiv 1 \pmod{2}$  jsou právě všechna lichá čísla.

Řešením kongruenze  $x \equiv 4 \pmod{7}$  jsou všechna čísla  $x = 7k + 4$ ,  $k \in \mathbb{Z}$ .

Řešením kongruenze  $3x \equiv 0 \pmod{7}$  jsou všechna čísla  $x = 7k$ ,  $k \in \mathbb{Z}$ .

Řešením kongruenze  $3x \equiv 4 \pmod{7}$  jsou všechna čísla  $x = 6 + 7k$ ,  $k \in \mathbb{Z}$ .

- Kongruence  $3x \equiv 1 \pmod{6}$  nemá žádné řešení.
- pro řešení jsou vhodné inverze modulo tj. přirozené číslo pro které platí číslo \*  $a \equiv 1 \pmod{m}$ .

- Číslo 3 je inverzí čísla 5 modulo 7 protože  $3 \cdot 5 \equiv 1 \pmod{7}$ .
- Číslo 3 je inverzní samo k sobě modulo 8 protože  $3 \cdot 3 \equiv 1 \pmod{8}$ .
- Číslo 3 je inverzí čísla 7 modulo 10 protože  $3 \cdot 7 \equiv 1 \pmod{10}$ .
- Číslo 8 nemá inverzi modulo 10 protože  $8 \cdot x$  je sudé a  $8 \cdot x \not\equiv 1 \pmod{10}$
- Protože  $(3, 7) = 1$ , tak můžeme napsat  $1 = 5 \cdot 3 - 2 \cdot 7 \equiv 3 \cdot 5 \pmod{7}$ .
- Číslo 5 je inverzí k číslu 3 modulo 7, platí  $\bar{3} = 5$ .
- Řešení lineárních kongruencí

### Příklad

Najděte všechna řešení lineární kongruence  $3x \equiv 4 \pmod{7}$ .

Nejprve najdeme k číslu 3 inverzi modulo 7. Podle předchozího příkladu  $\bar{3} = 5$ .

Nyní obě strany kongruence vynásobíme inverzí 5. Dostaneme

$$\begin{aligned} 5 \cdot 3x &\equiv 5 \cdot 4 \pmod{7} \\ x &\equiv 20 \pmod{7} \\ x &\equiv 6 \pmod{7} \end{aligned}$$

Řešením jsou všechna čísla, která po dělení 7 dávají zbytek 6.

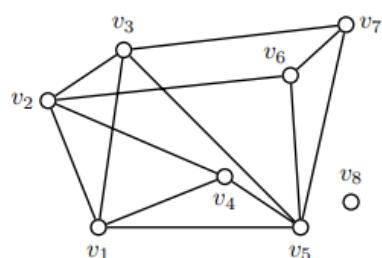
- Řešení jsou čísla  $x = 7k + 6$ , kde  $k \in \mathbb{Z}$ .

### • Teorie grafů (grafy, vzdálenost v grafu, míry souvislosti, stromy a kostry, barvení grafů a jejich aplikace, toky v sítích).

dle: [https://homel.vsb.cz/~kov16/files/uvod\\_do\\_teorie\\_grafu.pdf](https://homel.vsb.cz/~kov16/files/uvod_do_teorie_grafu.pdf)

Pojem graf je definován jako dvojice neprázdná množina vrcholů a množina hran. Jedná se o algebraickou strukturu popisující objekty a vztahy mezi nimi. Výhodou grafů je jejich přehledné a intuitivní znázornění - vrcholy jako puntíky/kroužky a hrany jsou spoje mezi těmito vrcholy. Vrcholy v grafu obvykle značíme malými písmeny z konce abecedy (u,v,w).

Graf G



jeho množina  $V = \{v_1, v_2, \dots, v_8\}$  a množina  $E = \{v_1v_2, v_1v_3, v_1v_4, v_1v_5, v_2v_3, v_2v_4, v_2v_6, v_3v_5, v_3v_7, v_4v_5, v_5v_6, v_5v_7, v_6v_7\}$ .

Konečné grafy - mají neprázdnou konečnou množinu vrcholů.

Typy grafů - ? (kompletní, cykly,...)

Incidence - vztah mezi vrcholem a hranou. "hrana je incidentní se svými koncovými vrcholy"

sled = posloupnost vrcholů taková, že mezi každými dvěma po sobě jdoucími vrcholy je hrana

cesta = sled, ve kterém se žádné dva vrcholy neopakují

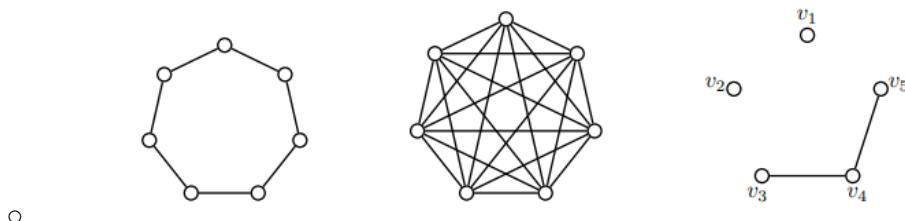
tah = sled, ve kterém se žádné dvě hrany neopakují

### vzdálenost v grafu

- vzdálenost  $\text{dist}_G(u,v)$  v grafu  $G$  mezi vrcholy  $u$  a  $v$  je dán *nejkratším sledem mezi vrcholy  $u$  a  $v$*
- nejkratší sled je vždy cestou

### souvislost grafu

- laická definice: Graf je souvislý pokud z každého vrcholu existuje sled do všech ostatních vrcholů.
- Graf je souvislý, pokud je tvořen jednou komponentou.



- souvislý, souvislý a nesouvislý graf

### hranová souvislost grafu

- Graf je hranově  $k$ -souvislý ( $k \geq 1$ ) pokud po odebrání  $k-1$  hran zůstane souvislý.
  - např. graf cyklus je hranově 2-souvislý protože když odeberu jednu hranu tak stále zůstane souvislý.

### vrcholová souvislost grafu

- Graf  $G$  je vrcholově  $k$ -souvislý, pokud  $|V(G)| > k = 1$  a po odebrání libovolných  $k - 1$  vrcholů z grafu  $G$  zůstane výsledný indukovaný podgraf souvislý. Stupeň vrcholové souvislosti grafu  $G$  je takové největší číslo  $k$ , že graf  $G$  je vrcholově  $k$ -souvislý.

U hranové i vrcholové souvislosti, aby byl graf hranově nebo vrcholově  $k$ -souvislý musí být možno odebrat libovolně dané množství hran nebo vrcholů - tzn. nestačí když se dá odebrat jen nějaké vybrané hrany nebo vrcholy aby zůstal souvislý.

### Eulerovský, Hamiltonovský graf?

Hamiltonovský - takový graf ve kterém existuje taková cesta přes všechny vrcholy (taková že každý vrchol je navštíven právě jednou)

Eulerovský - má všechny vrcholy sudého stupně a existuje tah obsahující všechny jeho hrany

### stromy a kostry

*Strom* je souvislý graf, který neobsahuje cykly. *Les* je graf, jehož komponenty jsou stromy.

#### Vlastnosti stromu

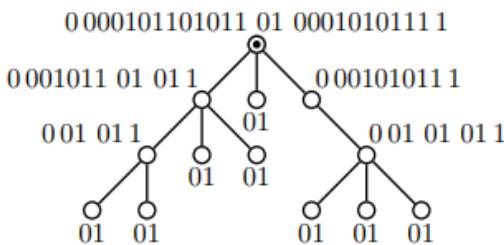
- Strom s  $n$  vrcholy má právě  $n - 1$  hran
- Mezi každými dvěma vrcholy stromu existuje právě jedna cesta

### Kódování uspořádaného stromu? - (využívá se pak u určování isomorfismu)

uspořádáný (neboli pěstovaný) strom = pro každý jeho vrchol je jednoznačně dán pořadí jeho potomků (zleva doprava při kreslení)

- mám dva uspořádané stromy
- začnu od listů - těm přiřadím hodnoty 01

- kód rodiče dostanu tak, že spojím kódy jeho potomků zleva doprava a takto spojené kódy potomků obalím zleva 0 a zprava 1



• Isomorfismus stromů? - (definice jsou nicneříkající, při vrtání do isomorfismu a jak se to určuje tato otázka hodně nabude na objemu)

- Dva kořenové stromy  $(T, r)$  a  $(T', r')$  jsou isomorfní pokud existuje isomorfismus mezi stromy  $T$  a  $T'$ , který zobrazí kořen  $r$  na kořen  $r'$
- Dva uspořádané kořenové stromy (pěstované stromy) jsou isomorfní, jestliže pro ně existuje isomorfismus kořenových stromů, který navíc zachová pořadí potomků každého vrcholu.
- obecně neexistuje žádný rychlý algoritmus pro určení *isomorfismu dvou grafů*, ale existuje algoritmus pro určení isomorfismu u dvou uspořádaných stromů
- algoritmus pro určení isomorfismu dvou stromů
  - Dva uspořádané kořenové (pěstované) stromy jsou isomorfní právě tehdy, když jejich kódy jsou shodné řetězce.
- isomorfismus kořenových stromů
  - určování probíhá pohledově jen tak, že se kontroluje od kořene počet přímých potomků dvou odpovídajících vrcholů v porovnávaných stromech - pokud počet nesedí - nejsou isomorfní

### kostra grafu

faktor grafu  $G$  = takový podgraf, který obsahuje všechny vrcholy grafu  $G$  (odstraní se "nepotřebné" hran)

kostra souvislého grafu = takový faktor grafu, který je stromem

váha kostry = v případě ohodnoceného grafu je to součet vah všech hran grafu

algoritmus pro hledání minimální kostry grafu? - (to už bych jebal)

### barvení grafu

využití v následující problémech:

- skladovací problém - ve skladu je hodně druhů potravin a některé druhy nemůžou být ve stejné místnosti s některými jinými druhy. Kolik nejméně místností potřebujeme abyhom mohli skladovat ty potraviny? (neřeší se velikost skladu, ani velikost místností, prostě jenom kolik nejméně místností je potřeba)
- optimalizace křížovatek - několikaproudá křížovatka, pruhy, které se kříží nemůžou mít zelenou ve stejný interval kdy svítí zelená- Kolik různých intervalů je potřeba aby každý pruh měl zelenou alespoň jednou?
  - graf, kde dva spojené vrcholy představují pruhy, které se kříží

Obarvení grafu  $G$  je takové zobrazení ( $c : V(G) \rightarrow \{1, 2, \dots, k\}$ ), ve kterém každé dva vrcholy spojené hranou mají různou barvu. (Můžeme obarvit každý vrchol jinou barvou, ale nás zajímá nejmenší počet barev) = *dobré vrcholové obarvení*

*horní odhad počtu barev* - pokud je graf kompletní, počet barev se rovná počtu vrcholů, jinak je horní odhad o jedno menší než počet vrcholů grafu

*dolní odhad počtu barev*

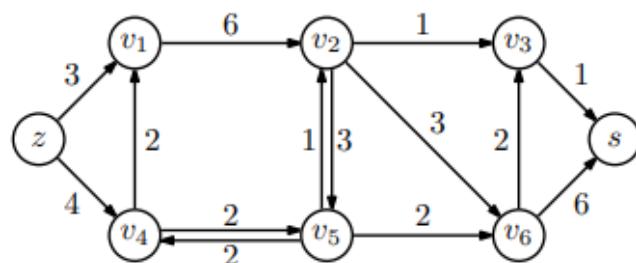
- graf má počet barev = 1 právě tehdy, když nemá žádné hrany
- graf má počet barev = 2 právě tehdy, když neobsahuje jako podgraf žádný cyklus liché délky.
- jestliže v grafu je kompletní podgraf na  $k$  vrcholech, tak k obarvení takového grafu je potřeba alespoň  $k$  barev

### toky v sítích

teoreticky řeší problémy:

- Jaký největší objem látky nebo dat můžeme přepravit po síti s danými omezeními z výchozího vrcholu  $z$  do cílového vrcholu  $s$ . Této úloze se říká hledání největšího toku v síti.
- kolik maximálně může procházet/téct ze zdroje do stoku v jeden moment

síť = Síť je čtveřice  $S = (G, z, s, w)$ , G je graf, z je zdroj, s je stok a w je funkce  $w : E(G) \rightarrow \mathbb{R}^+$ (každé hraně přiřadí kladné ohodnocení)



ohodnocení hran znamená kolik maximálně může hranou téct v jeden moment

pro síť platí

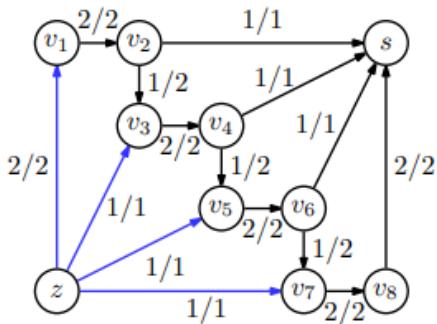
- tok hranou nesmí překročit kapacitu hranы
- zákon kontinuity
  - "to co do vrcholu přiteče tak musí i odtéct"
  - kromě zdroje a stoku
- zdroj = má více odchozích hran než příchozích
- stok = má více příchozích hran než odchozích
- velikost toku = rozdíl toho co do stoku přiteče a toho co odteče

### hledání největšího toku

řez = taková podmnožina hran ze síťe, kterou když odeberu ze síťe, tak v ní nezůstane žádná orientovaná cesta ze zdroje do stoku ( nemusí se nutně jednat o rozdělení grafu do dvou komponent, kde v jedné je zdroj a ve druhé je stok )

kapacita řezu = kapacita řezu C se značí jako  $|C|$  a je rovna součtu kapacit všech hran řezu C

velikost největšího toku v síti = kapacita řezu



hrany řezu jsou na obrázky modře

- Lineární algebra (matice a maticové operace, inverzní matice, úpravy a řešení soustav lineárních rovnic, vektorové prostory a báze vektorových prostorů, lineární zobrazení, spektrální teorie).

<https://homel.vsb.cz/~ber95/LA/la.htm>

<https://cs.wikipedia.org/wiki/Matice>

### Matice a maticové operace

- Matice typu  $(m, n)$  ( $m \times n$  matice) je obdélníková tabulka, která má  $mn$  prvků  $a_{ij}$  uspořádaných do  $m$  řádků a  $n$  sloupců.

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$

- Prvky  $a_{11}, a_{12}, \dots, a_{mn}$  jsou z dané množiny  $F$ . Prvky z této množiny nazýváme také skaláry (lze je sčítat násobit obdobně jako čísla).
- Množinu všech matic typu  $(m, n)$  s prvky z množiny  $F$  budeme znacit  $F^{m,n}$ , např. matice  $A$  typu  $(7,8)$ , která bude z množiny reálných čísel, pak  $A \in \mathbb{R}^{7,8}$ .
- Jestliže  $m = n$ , pak se matice nazývá **čtvercová matice rádu  $n$** .
- Matice typu  $(1, n)$  nazýváme **řádkovým vektorem rádu  $n$** .
- Matice typu  $(m, 1)$  nazýváme **sloupcovým vektorem rádu  $m$** .
- Diagonálu matice tvoří prvky  $a_{11}, a_{22}, \dots, a_{ss}$ .
- Prvek v  $i$ -tém řádku a  $j$ -tém sloupci matice  $A$  značíme  $[A]_{ij}$ .
- Pokud jsou matice  $A$  a  $B$  stejné, tj. mají stejně odpovídající prvky ( $[A]_{ij} = [B]_{ij}$ ), pak značíme  $A = B$ , jinak  $A \neq B$ .

### Typy matic

- **Nulová matice** (všechno nuly) typu  $(m, n)$  se značí  $\mathbf{0}_{mn}$ .
- **Opačná matice** k matici  $A$  je matice  $-A$  a platí:  $-A = (-1) * A$

- **Jednotková matice** je čtvercová matice s 1 na diagonále a značí se  $I_n$ .

○ 
$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Maticové operace

- **násobení skaláru  $\alpha$  a matice A.**

○ 
$$2 \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix}$$

- **Součet matic A a B** stejného typu je matice stejného typu jako A a B.

○ 
$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 5 \\ 4 & 5 \end{bmatrix}$$

- pro libovolné matice stejného typu a skaláry platí vztahy:

$$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C} \quad (1)$$

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \quad (2)$$

$$\alpha(\mathbf{A} + \mathbf{B}) = \alpha\mathbf{A} + \alpha\mathbf{B} \quad (3)$$

$$(\alpha + \beta)\mathbf{A} = \alpha\mathbf{A} + \beta\mathbf{A} \quad (4)$$

$$\alpha(\beta\mathbf{A}) = (\alpha\beta)\mathbf{A} \quad (5)$$

$$1\mathbf{A} = \mathbf{A} \quad (6)$$

○ Obdobně jako v případě vektorů, vlastnosti (3),(4),(5) jsou velmi důležité při výpočtech s velmi velkými maticemi.

○ Např. stokrát výpočet  $2.0\mathbf{A} + 3.0\mathbf{A}$  s čtvercovou maticí řádu 1000 potřebuje 4.562711 sekund, zatímco výpočet  $(2.0 + 3.0)\mathbf{A}$  pouze 1.895391 sekund.

- **odečítání matic A a B**

○ 
$$\mathbf{B} = \begin{bmatrix} 1 & -2 \\ 3 & 0 \end{bmatrix}, \mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & -2 \end{bmatrix},$$
  

$$\mathbf{B} - \mathbf{A} = \mathbf{B} + (-\mathbf{A}) = \begin{bmatrix} 1 & -2 \\ 3 & 0 \end{bmatrix} + \begin{bmatrix} -2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} -1 & -3 \\ 2 & 2 \end{bmatrix}$$

- **Transponované matice**

○ 
$$[\mathbf{A}^\top]_{ij} = [\mathbf{A}]_{ji}$$

- $$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

- pro matice stejného typu a skalár platí:

$$(\mathbf{A} + \mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top,$$

$$(\alpha \mathbf{A})^\top = \alpha \mathbf{A}^\top.$$

- **Násobení matic a vektoru**

- násobením matice  $A$  typu  $(m,n)=(2,2)$  a sloupcového vektoru  $u$  dimenze  $n = 2$  vyjde vektor dimenze  $m$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix}$$

- 

- **Násobení matic**

- $AB$ , počet sloupců v matici  $A$  se musí rovnat počtu řádků v matici  $B$ , jinak násobit nelze.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

- 
- 
- 

$$\begin{bmatrix} A \\ 1 & 2 \\ 0 & 1 \\ \hline 2 & 3 \end{bmatrix} \begin{bmatrix} B \\ 2 & 1 \\ 0 & -1 \\ -2 & 3 \end{bmatrix} \text{ nelze násobit!!!}$$

- Výsledná matice, po vynásobení matic  $A$  typu  $(m,n)$  a  $B$  typu  $(n,p)$ , bude mít typ  $(m,p)$ .
- **Násobení matic je velmi výpočetně nákladné.** Je proto velmi důležité operace dělat co nejfektivněji. Tj. místo  $AC + BC$  je rychlejší  $(A + B)C$ .

Pro násobení matic  $\mathbf{A}$  typu  $(m, p)$ , matic  $\mathbf{B}$  typu  $(p, q)$  a matic  $\mathbf{C}$  typu  $(q, n)$  platí také tzv. *asociativní zákon*, tj.

- 

$$\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$$

- **Násobení matic není komutativní!**

$$\mathbf{AB} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}, \mathbf{BA} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}$$

takže  $\mathbf{AB} \neq \mathbf{BA}$ .

- 

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$$

$$\mathbf{B}^2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \mathbf{O}.$$

Pro násobení matic tedy *neplatí komutativní zákon a mocnina*

- *nenulové matice může být nulová matice!*

## Inverzní matice

- **Inverzní matici** k dané regulární čtvercové matici je taková matici, která po vynásobení s původní maticí dá jednotkovou matici.
- Čtvercová matici, ke které **existuje** inverzní se nazývá **regulární** matici. Pokud **neexistuje**, tak se nazývá **singulární** matici.
- Ke každé regulární matici existuje právě jedna inverzní matici.

**LEMMA 1** Má-li matici  $A$  nulový řádek pak je singulární.

DŮKAZ: Je-li  $B$  libovolná matici, pak platí

$$AB = \begin{bmatrix} \cdot & \cdots & \cdot \\ 0 & \cdots & 0 \\ \cdot & \cdots & \cdot \end{bmatrix} \neq I.$$

Výsledná matici po vynásobení  $AB$  bude mít jeden řádek nulový, tzn. to není určitě jednotková matici, takže matici  $A$  je singulární.

- Výpočet inverzní matici
- Pomocí inverzní matici lze řešit i soustavy rovnic.

### Použití

- K nalezení inverzní matici je zapotřebí asi  $n^3$  operací násobení.
  - Nevyplatí se řešit jednu soustavu pomocí inverzní matici.
  - Může být výhodné řešit soustavy s více pravými stranami pomocí inverzní matici (řešení pro každou pravou stranou pak vyžaduje asi  $n^2$  násobení).
- Inverzní matici je spíše teoretický prostředek.

## Úpravy a řešení soustav lineárních rovnic

Základní myšlenka řešení soustavy lineárních rovnic spočívá v nahrazení dané soustavy jinou soustavou, která má stejné řešení, avšak jednodušší. K tomu slouží ekvivalentní úpravy. U matic jim říkáme *elementární (řádkové) operace*.

Elementární (řádkové) operace:

- Vzájemná výměna libovolných dvou řádků.
- Násobení některého řádku nenulovým číslem.
- Přičtení nenulového násobku některého řádku k jinému řádku.

Máme-li dvě matice, z nichž jedna vznikla z druhé pomocí elementárních řádkových operací, říkáme, že matice jsou **řádkově ekvivalentní**.

Zápis soustavy do matice:

$$\left[ \begin{array}{ccc|c} a_{11} & \dots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m1} & \dots & a_{mn} & b_m \end{array} \right] \quad - \text{Rozšířená matici soustavy}$$

Schodový tvar matice - Jestliže má první nenulové prvky řádků zvané vedoucí prvky uspořádány jako schody klesající zleva doprava.

- Každý vedoucí prvek (první nenulové číslo řádku zleva) musí být více vpravo než vedoucí prvek o řádek výše.

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 0 & 2 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 2 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 & 3 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Soustava nemá řešení:

- Jestliže poslední nenulový řádek rozšířené matice soustavy má nenulový pouze poslední prvek, pak daná rovnice nemá řešení, tudíž celá soustava nemá řešení.

$$\left[ \begin{array}{ccc|c} 1 & 2 & -1 & 1 \\ 0 & 1 & -1 & 2 \\ 0 & 0 & 0 & -3 \end{array} \right]$$

Tato rozšířená matice soustavy odpovídá soustavě:

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 1 \\ x_2 - x_3 &= 2 \\ 0 &= -3 \end{aligned} \Rightarrow \text{N.R.}$$

Nekonečně mnoho řešení:

$$\begin{aligned} x_1 + x_2 + x_3 &= 1 \\ -x_2 - 2x_3 &= 0 \\ 0 &= 0 \end{aligned}$$

Soustava má tedy nekonečně mnoho řešení ve tvaru  $x_3$  libovolné,  $x_2 = -2x_3, x_1 = 1 + x_3$ . Můžeme je zapsat také pomocí libovolného parametru  $p$  ve tvaru  $x_3 = p, x_2 = -2p, x_1 = 1 + p$ .

**Poznámka:** Má-li soustava nekonečně mnoho řešení, je *množina* řešení určena jednoznačně, nikoliv však její *parametrizace*. Například  $x_2 = p, x_3 = -\frac{1}{2}p$  a  $x_1 = -\frac{1}{2}p + 1$  je *jiný* tvar téhož řešení.

Metody řešení:

Gaussova eliminační metoda:

1. *dopředná redukce*, tj. redukce na schodový tvar
  2. *zpětná substituce*, tj. řešení soustavy se schodovou maticí
- 
- Gaussova eliminační metoda je velmi efektivní pro ruční řešení malých soustav a pro počítačové řešení soustav stovek až tisíců rovnic.
  - Pro rozsáhlejší soustavy existují efektivnější metody, rozvíjeny i v dnešní době.

#### DEFINICE 4

Budeme říkat, že matice je v *normovaném schodovém tvaru*, jestliže je v takovém schodovém tvaru, že všechny prvky nad vedoucími prvky jsou nulové a navíc vedoucí prvky jsou rovny jedné.

Gaussova–Jordanova metoda:

1. *dopředná redukce*, tj. redukce na schodový tvar
  2. úprava na normovaný schodový tvar
    - (a) dělení řádků matice vedoucími prvky
    - (b) nulování prvků nad vedoucími prvky pomocí elementárních řádkových operací (e1),(e2) a (e3)
- 

Vektorové prostory a báze vektorových prostorů

<https://www.matweb.cz/vektorove-prostory/>

(definice podle vlastních slov) - **Vektorový prostor** je množina vektorů, ve které provádíme operace: sčítání vektorů nebo násobení vektoru se skalárem (číslem). Tyto operace musí splňovat určité pravidla (viz obrázek níže s axiomy), aby se to dalo nazývat vektorový prostor.

**Vektorový prostor** (nebo též lineární prostor) je neprázdná množina  $V$ , jejíž prvky nazýváme vektory. Dále musí na množině existovat dvě operace:

- sčítání vektorů, tj. zobrazení  $V \times V \rightarrow V$ 
  - př.

$$V = \mathbb{R}^2 = \mathbb{R} \times \mathbb{R} = \{(a, b) \mid a, b \in \mathbb{R}\}$$

všechny vektory budou dvourozměrné z množiny reálných čísel a výsledek tohoto sečtení musí být také z této množiny.

$$(1,2) + (2,2) = (3,4)$$

- násobení vektoru číslem (skalárem), tj. zobrazení  $K \times V \rightarrow V$ , kdy  $K$  je těleso (množina prvků, např. čísel, pomocí kterých násobíme vektor)

$$2 \cdot (1,2) = (2,4)$$

- Pro tyto dvě operace platí následující axiomy/pravidla:

**VP1**  $\forall \mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathcal{V} : \mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$

**VP2**  $\exists \mathbf{o} \in \mathcal{V} \forall \mathbf{u} \in \mathcal{V} : \mathbf{u} + \mathbf{o} = \mathbf{o} + \mathbf{u} = \mathbf{u}$

**VP3**  $\forall \mathbf{u} \in \mathcal{V} \exists -\mathbf{u} \in \mathcal{V} : \mathbf{u} + (-\mathbf{u}) = -\mathbf{u} + \mathbf{u} = \mathbf{o}$

**VP4**  $\forall \mathbf{u}, \mathbf{v} \in \mathcal{V} : \mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$

**VP5**  $\forall \alpha \in \mathbb{R}(\mathbb{C}) \forall \mathbf{u}, \mathbf{v} \in \mathcal{V} : \alpha(\mathbf{u} + \mathbf{v}) = \alpha\mathbf{u} + \alpha\mathbf{v}$

**VP6**  $\forall \alpha, \beta \in \mathbb{R}(\mathbb{C}) \forall \mathbf{u} \in \mathcal{V} : (\alpha + \beta)\mathbf{u} = \alpha\mathbf{u} + \beta\mathbf{u}$

**VP7**  $\forall \alpha, \beta \in \mathbb{R}(\mathbb{C}) \forall \mathbf{u} \in \mathcal{V} : \alpha(\beta\mathbf{u}) = (\alpha\beta)\mathbf{u}$

**VP8**  $\forall \mathbf{u} \in \mathcal{V} : 1\mathbf{u} = \mathbf{u}$

( $\mathbf{o}$  - nulový vektor)

- příklady

- **jedná** se o vektorový prostor

$$V = \mathbb{R}^2 = \mathbb{R} \times \mathbb{R} = \{(a, b) \mid a, b \in \mathbb{R}\}$$

-dvourozměrný vektor z množiny kladných reálných čísel

$$K = \mathbb{R}$$

-skalár z množiny reálných čísel

Jakýkoliv zvolím vektor nebo skalár ať už při sčítání, nebo násobení, vždycky výjde dvourozměrný vektor z množiny reálných čísel.

- **NEjedná** se o vektorový prostor

$$V = \mathbb{R}^+ = \{(a) \mid a \in \mathbb{R}^+\}$$

- jednorozměrný vektor z množiny kladných reálných čísel

$$K = \mathbb{R} \quad - \text{skalár z množiny reálných čísel}$$

Při sčítání nebude problém, avšak když při násobení skaláru s vektorem jako skalár zvolím záporné číslo, tak výjde i záporná složka vektoru.

$-2 * (1,2) = (-2,-4) \Rightarrow$  TENTO VEKTOR UŽ NENÍ Z MNOŽINY Kladných reálných čísel, tzn. NEjedná se o vektorový prostor

- **Vektorový PODprostor** je podmnožina vektorového prostoru, která sama o sobě je vektorový prostor, jelikož splňuje všechny axiomy a podporuje sčítání a násobení.
  - příklad

**PŘÍKLAD 8** Nechť  $\mathcal{V}$  je libovolný prostor. Pak  $\mathcal{O} = \{\mathbf{o}\}$  je podprostorem  $\mathcal{V}$ , neboť  $\mathbf{o} + \mathbf{o} = \mathbf{o}$  a pro libovolný skalár  $\alpha$  platí, že  $\alpha\mathbf{o} = \mathbf{o}$ . Vektorový prostor  $\mathcal{O}$  je nejmenší podprostор daného vektorového prostoru a nazývá se *nulovým podprostorem*.

$$\mathcal{O} \subseteq \mathcal{V}$$

(o - nulový vektor)

### Báze vektorových prostorů

- Konečná množina  $\mathcal{E}$  vektorů vektorového prostoru  $\mathcal{V}$  je *báze vektorového prostoru  $\mathcal{V}$* , jestliže:

1. Množina  $\mathcal{E}$  je nezávislá.

- Neprázdná konečná množina vektorů  $\mathcal{E} = \{v_1, \dots, v_k\}$  vektorového prostoru  $\mathcal{V}$  je **(lineárně) nezávislá**, jestliže rovnice:

$$\alpha_1 v_1 + \dots + \alpha_k v_k = \mathbf{o}$$

má jediné řešení

$$\alpha_1 = \dots = \alpha_k = 0$$

pokud existuje i jiné řešení (jiné hodnoty koeficientů  $\alpha$  než 0), pro které dostanu nulový vektor tak je lineárně závislé skupina vektorů je lineárně nezávislá pokud jsou v ní vektory, které mají různý směr a je jich minimální počet pro popis libovolného vektoru v prostoru (tj. ve 2D 2 vektory - pokud by byl 1 odebrán už nepopisuje celý prostor, pokud lze ze skupiny vektor odebrat aniž by byla omezena popisovací schopnost jedná se o lineárně závislou skupinu)

- Pokud existuje i nenulové řešení, pak je **(lineárně) závislá**.
  - pokud množina  $\mathcal{E}$  obsahuje vektory, které jsou vůči sobě násobkem  $\Rightarrow \mathcal{E}$  je (lineárně) závislá.
- 17 - Lineární závislost a nezávislost (MAT - Lineární algebra)

**PŘÍKLAD 1** Jestliže  $\mathbf{v}_1 = [2, -1, 0]$ ,  $\mathbf{v}_2 = [1, 2, 5]$  a  $\mathbf{v}_3 = [7, -1, 5]$ , pak množina vektorů  $\mathcal{S} = \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$  je lineárně ~~závislá~~, neboť  $3\mathbf{v}_1 + \mathbf{v}_2 - \mathbf{v}_3 = \mathbf{o}$ .

*nezávislá*

- 2. Každý vektor  $v \in V$  lze vyjádřit jako lineární kombinací vektorů z množiny  $\mathcal{E}$ .

Vektor  $v$  z vektorového prostoru  $\mathcal{V}$  budeme nazývat *lineární kombinací* vektorů  $v_1, \dots, v_k \in \mathcal{V}$ , jestliže existují skaláry  $\alpha_1, \dots, \alpha_k$  tak, že

$$v = \alpha_1 v_1 + \dots + \alpha_k v_k.$$



Příklad:

Lze zapsat vektor  $v$  jako lineární kombinace vektorů  $u_1, u_2, u_3$ ?

$$u_1 = (1; 0; 3); u_2 = (3; -2; 1); u_3 = (8; -4; 8); v = (5; -2; 9)$$

$$\begin{array}{ccc|c} u_1 & u_2 & u_3 & v \\ a & +3b & +8c & 5 \\ 0a & -2b & -4c & -2 \\ 3a & + b & +8c & 9 \end{array}$$

$$\left( \begin{array}{ccc|c} 1 & 3 & 8 & 5 \\ 0 & -2 & -4 & -2 \\ 3 & 1 & 8 & 9 \end{array} \right) \sim \dots \sim \left( \begin{array}{ccc|c} 1 & 3 & 8 & 5 \\ 0 & -2 & -4 & -2 \\ 0 & 0 & 0 & 2 \end{array} \right)$$

$$\begin{aligned} 0a + 0b + 0c &= 2 \\ 0 &\neq 2 \end{aligned}$$

Pokud soustava nemá řešení => nejde zapsat jako Lineární kombinace

Má jedno řešení => jde zapsat jako unikátní LK

Nekonečno řešení => jde zapsat jako LK nekonečně mnoha způsoby



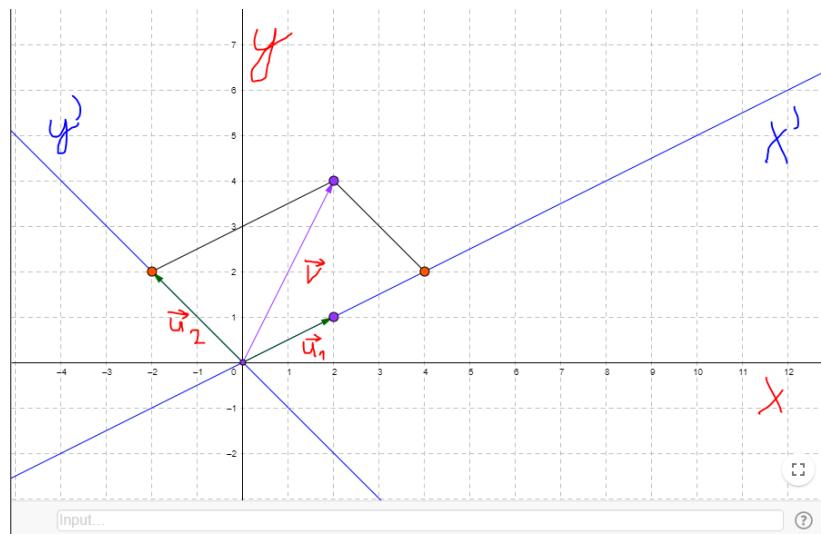
## ■ 16 - Příklady lineární kombinace (MAT - Lineární algebra)

- PŘÍKLAD 6** Vektory  $e_1 = [1, 0, 0], e_2 = [0, 1, 0], e_3 = [0, 0, 1]$  tvoří bázi  $\mathcal{V} = \mathbb{R}^3$ . Jakýkoli vektor  $v = [v_1, v_2, v_3]$  tohoto prostoru lze vyjádřit ve tvaru  $v = v_1 e_1 + v_2 e_2 + v_3 e_3$ . Báze  $\mathcal{E} = (e_1, e_2, e_3)$  je zvláštním případem *standardní báze*  $\mathbb{R}^n$ , která je tvořena řádky či sloupci jednotkové matice  $I_n$ .

- 22 - Souřadnice vektoru v bázi (MAT - Lineární algebra)

- Báze, kterou jsme zvyklý používat je Kartézská soustava souřadnic. Jde o Standardní bázi - je tvořena dvěma vektory se souřadnicemi 1,0 a 0,1, respektive osami x a y.
  - Tato báze splňuje obě dvě pravidla:
    - Vektory nejsou vůči sobě násobkem => jsou Lineárně nezávislé
    - Lineární kombinací těchto vektorů lze vyjádřit všechny vektory z množiny čísel Kartézské soustavy souřadnic.
- Báze je vlastně nová soustava souřadnic (nemusí být na sebe kolmá). Vektor zobrazený v kartézské, bude stejně umístěný, jako v nové soustavě (bázi), akorát osy budou nové a to podle těch dvou vektorů.

- $u_1 = (2, 1)$ ,  $u_2 = (-2, 2)$ ,  $v = (2, 4)$
- Nová báze  $\alpha = (u_1, u_2)$ , na obrázku zobrazena osami x a y s čárkou.
- vektory na jednotlivých nových osách nám budou určovat jejich krok. Jako na ose x a y je krok 1, tak na nových osách jako jeden krok je považován daný vektor.



$$\left( \begin{array}{cc|c} (\vec{u}_1) & (\vec{u}_2) & (\vec{v}) \\ a & b & \\ \hline 2 & -2 & 2 \\ 1 & 2 & 4 \end{array} \right) \xrightarrow{\begin{matrix} \cdot \frac{1}{2} \\ -2 \\ \end{matrix}} \left( \begin{array}{cc|c} 1 & -1 & 1 \\ 0 & 3 & 3 \end{array} \right)$$

$$2a - 2b = 2$$

$$3b = 3$$

$$\begin{array}{l} 2a = 4 \\ \underline{a = 2} \end{array} \quad \begin{array}{l} b = 1 \\ \underline{b = 1} \end{array}$$

Pomocí lineární kombinace přepočítám/vyjádřím vektor  $v$  do báze  $\alpha$ .

Výsledek:

$a = 2$ , tj. že vektor  $u_1$  na ose  $x'$  vezmu dvakrát.

$b = 1$ , tj. že vektor vezmu jen jednou.

A lze vidět, že vektor je v obou bázích, ať už ve standardní (kartézská soustava) nebo v nové bázi, na stejném místě, akorát se liší jeho "popis"/vyjádření.

## Lineární zobrazení

Pojmem lineární zobrazení (lineární transformace) se v matematice označuje takové zobrazení mezi vektorovými prostory X a Y, které zachovává vektorové operace sčítání a násobení skalárem.

Název lineární je odvozen z faktu, že grafem obecného lineárního zobrazení  $\mathbb{R} \rightarrow \mathbb{R}$  je přímka.

- Nechť  $U, V$  jsou vektorové prostory,  $u$  a  $v$  jsou vektory,  $\alpha$  je skalár (číslo).
- Zobrazení  $A: U \rightarrow V$  (neboli  $A \in$ ) se nazývá lineární zobrazení (operátor) jestliže platí:
  1.  $\forall u, v \in U : A(u + v) = A(u) + A(v)$
  2.  $\forall u \in U, \alpha \in K : A(\alpha u) = \alpha A(u)$
- Lineární zobrazení  $U \rightarrow U$  se často nazývá *lineární transformace*.
- Pokud nulový vektor není zobrazen na nulový vektor, **nejedná** se o lineární zobrazení.
- příklad:

$$A: \mathbb{R}^2 \rightarrow \mathbb{R}^2, A(x, y) = (x + 3, y - 2)$$

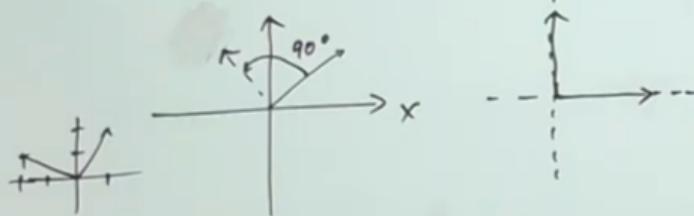
u tohoto příkladu lze ihned po zobrazení nulové vektoru zjistit, že se nejedná o linární transformaci.

$$A(0, 0) = (0 + 3, 0 - 2)$$

$A(0, 0) = (3, -2) \Rightarrow$  nulový vektor není zobrazen na nulový vektor,  
tzn. **nejedná** se o linární transformaci

- určení přepisu zobrazení

Určete předpis zobrazení, které určuje rotaci vektorů v  $\mathbb{R}^2$  o  $90^\circ$ .



$$\varphi: \mathbb{R}^2 \rightarrow \mathbb{R}^2 \quad S = (u_1, v_1) = ((1; 0), (0; 1))$$

$$\varphi(x; \gamma) = (-\gamma; x) \quad \begin{aligned} (1; 0) &\rightarrow (0; 1) \\ (0; 1) &\rightarrow (-1; 0) \end{aligned}$$

$$\begin{aligned} \varphi((x; \gamma)) &= \varphi((x; 0) + (0; \gamma)) = \\ &= \varphi((x; 0)) + \varphi((0; \gamma)) = \\ &= \varphi(\underline{x} \cdot (1; 0)) + \varphi(\underline{\gamma} \cdot (0; 1)) = \\ &= x \cdot \varphi((1; 0)) + \gamma \cdot \varphi((0; 1)) = \\ &= x \cdot (0; 1) + \gamma \cdot (-1; 0) = \\ &= (0; x) + (-\gamma; 0) = \underline{(-\gamma; x)} \end{aligned}$$

$\varphi(1; 0)$

- pomocí druhého pravidla vytkl x a y zevnitř (červené řádky).
- matice zobrazení
  - slouží pro jednodušší zobrazení vektoru
  - Nechť  $A$  je matice zobrazení a  $u$  je vektor. Zobrazený vektor dostanu, tak že původní vynásobím s maticí.
  - $\varphi(u) = A * u$
  - Skládání lineárních zobrazení se tak redukuje na násobení matic, původní působení lineárního zobrazení na vektor je nyní představováno násobením vektoru maticí.
  - Stejné zobrazení jako v příkladu výše (rotace vektoru o  $90^\circ$ )

Určete matici zobrazení  $\varphi$ :

a)  $\varphi: \mathbb{R}_s^2 \rightarrow \mathbb{R}_s^2$ ;  $\varphi(x_{12}) = (-x_1, x_2)$

$$S = \left( \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) \quad \varphi(u_1) = A_s \cdot u_1$$

$$\varphi\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}; \quad \varphi\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$\boxed{A_s = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}}$$

$$u_1 = \underbrace{\begin{pmatrix} 2 \\ 1 \end{pmatrix}}_{=} \rightarrow \varphi\left(\begin{pmatrix} 2 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} -1 \\ 2 \end{pmatrix}}_{=}$$

$$\varphi\left(\begin{pmatrix} 2 \\ 1 \end{pmatrix}\right) = \underbrace{\begin{pmatrix} -1 \\ 2 \end{pmatrix}}_{=}$$

o

- $S$  - standardní báze v prostoru  $\mathbb{R}^2$
- $A_s$  - matice zobrazení (dolní index s značí, že pracujeme se standardní bází)
- Každé lineární zobrazení jde promítnout do matice.

### Spektrální teorie

(definici/úvod jsem nějak nebyl schopný dát dokupu)

Nechť  $A: V \rightarrow V$  je lineární transformace definovaná na vektorovém prostoru  $V$ . Jestliže existuje nenulový vektor  $e \in V$  a skalár  $\lambda$  (lambda) tak, že

$$Ae = \lambda e$$

pak se

$\lambda$  nazývá *vlastní číslo* transformace  $A$ ,

$e$  se nazývá *vlastní vektor* příslušný k  $\lambda$ ,

$(\lambda, e)$  se nazývá *dvojice transformace  $A$* .

Tato rovnice lze zapsat pomocí identity ve tvaru (přepis této rovnice na obrázku níže)

$$(A - \lambda I)e = o$$

takže  $\lambda$  je vlastním číslem  $A$ , právě když  $A - \lambda I$  není prosté zobrazení, a vlastní vektory  $A$  příslušné k  $\lambda$  jsou prvky jádra  $A - \lambda I$ .

$I$  - jednotková matice

$o$  - nulový vektor, který vznikl po odečtení  $\lambda e$  z pravé strany

Vezmeme si rovnici

$$Ae = \lambda e$$

a vynásobíme ji zleva  $I$  (jednotkovou maticí)

$$IAe = I\lambda e,$$

protože  $IA = A, I\lambda = \lambda I, \lambda$  je skalár

$$Ae = \lambda I e,$$

odečteme  $\lambda I e$

$$Ae - \lambda I e = o$$

a vytkneme

$$(A - \lambda I)e = o.$$

Mějme čtvercovou matici  $A$ . **Spektrem matice  $A$**  nazýváme množinu všech **vlastních čísel** matice  $A$ .

Spektrum se obvykle označuje  $\sigma(A)$

Je-li **A** čtvercová matice, tak násobení maticí  $A - \lambda I$  není prosté zobrazení, právě když  $A - \lambda I$  je singulární. Pak skalár  $\lambda \in \sigma(A)$ , právě když  $\det(A - \lambda I) = 0$ .

Tady využijeme vlastnosti singulární matice, že její determinant je roven 0.

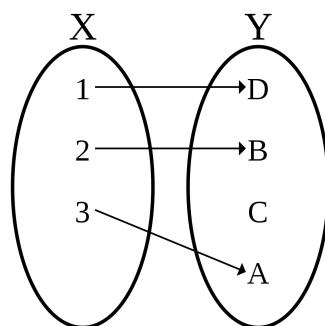
legenda:

výraz  $\det(A - \lambda I)$  se nazývá charakteristický mnohočlen

výraz  $|A - \lambda I| = 0$  se nazývá charakteristická rovnice

singulární - nemá inverzní matici

prosté zobrazení - každý vzor má max. jeden obraz, a každý obraz má max. jeden vzor (viz obrázek níže). Vzory jsou X a obrazy Y.



**LEMMA 1** Nechť  $A = [a_{ij}]$  je čtvercová matice  $n$ -tého řádu. Pak

1.  $\lambda_1 \cdot \dots \cdot \lambda_n = \det A$
2.  $\lambda_1 + \dots + \lambda_n = a_{11} + \dots + a_{nn}$  (tzv. stopa matice)

vlastnosti:

- Nula je vlastním číslem matice právě tehdy, když je matice singulární.
- Je-li matice symetrická a reálná, pak všechna její vlastní čísla jsou reálná.
- Jestliže k matici  $A$  existuje inverzní matice  $A^{-1}$ , pak  $\lambda$  je vlastním číslem matice  $A$  právě tehdy, je-li  $\frac{1}{\lambda}$  vlastním číslem matice  $A^{-1}$ . A také vlastní vektory matice  $A$  odpovídající vlastnímu číslu  $\lambda$  jsou stejné jako vlastní vektory matice  $A^{-1}$  odpovídající vlastnímu číslu  $\frac{1}{\lambda}$ .

Příklad výpočtu vlastních čísel a vektorů matice

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$A - \lambda I = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \lambda \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} =$$

$$= \begin{bmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{bmatrix}$$

$$\det(A - \lambda I) = \begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} \stackrel{\text{II}}{=} (2-\lambda)(2-\lambda) - 1 \cdot 1 = 4 - 4\lambda + \lambda^2 - 1$$

$$= \lambda^2 - 4\lambda + 3 = 0$$

Vlastní čísla  $\rightarrow \underline{\lambda_1 = 3, \lambda_2 = 1}$

$$\begin{bmatrix} 2-3 & 1 \\ 1 & 2-3 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{aligned} x_2 &= t \\ -x_1 + t &= 0 \\ x_1 &= t \end{aligned}$$

$$(x_1, x_2) = (t, t) = t \cdot (1, 1)$$

$$\begin{bmatrix} 2-1 & 1 \\ 1 & 2-1 \end{bmatrix} \stackrel{(-1)}{=} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{aligned} x_2 &= t \\ x_1 + t &= 0 \\ x_1 &= -t \end{aligned}$$

Vlastní vektory  $\rightarrow e_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$$e_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$e_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$(x_1, x_2) = (-t, t) = t \cdot (-1, 1)$$

$$e_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Kdyby jsme neskončili vlastními čísly a vektory matice  $A$ , tak by jsme pokračovali na celý spektrální rozklad matice  $A$  (postup

[https://homel.vsb.cz/~ber95/LA/Podklady\\_od\\_cvicicich/Raiman/09%20Spektr%C3%A1ln%C3%AD%20rozklad,%20Ger%C5%A1gorinova%20v%C4%9Bta.pdf](https://homel.vsb.cz/~ber95/LA/Podklady_od_cvicicich/Raiman/09%20Spektr%C3%A1ln%C3%AD%20rozklad,%20Ger%C5%A1gorinova%20v%C4%9Bta.pdf) )

Spektrální rozklad matice

## Definice ortogonální matice a věta o spektrální rozkladu

Matice  $Q$  se nazývá ortogonální, jestliže

$$QQ^T = I.$$

Věta

Nechť  $A$  je reálná symetrická matice. Pak existuje ortogonální matice  $Q$  a diagonální matice  $D$  tak, že

$$A = Q^T D Q.$$

Navíc řádky matice  $Q$  tvoří ortonormální vlastní vektory matice  $A$  a diagonální prvky matice  $D$  jsou jim odpovídající vlastní čísla. Rozklad  $Q^T D Q$  nazýváme spektrálním rozkladem matice  $A$ .

Tento příklad navazuje na minulý obrázek.

$$\begin{aligned}
 & \text{overení} \quad \text{ověření kolmosti vektorů (orthogonalitě)} \\
 e_1 &= (t, t) \quad (e_1, e_2) = t \cdot t + t \cdot t = -t^2 + t^2 = 0 \Rightarrow \text{jsou kolmé} \\
 e_2 &= (-t, t) \\
 & \text{Normalizujeme} \\
 q_1 &= \frac{e_1}{\|e_1\|} = \frac{(t, t)}{\sqrt{t^2+t^2}} = \underbrace{\frac{(t, t)}{t\sqrt{2}}}_{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}} = \left( \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right) \\
 q_2 &= \frac{e_2}{\|e_2\|} = \frac{(-t, t)}{\sqrt{t^2+t^2}} = \frac{(-t, t)}{t\sqrt{2}} = \left( -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right) \\
 \lambda_1 &= 3, \lambda_2 = 1 \\
 D &= \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}, \quad Q^T = (q_1 \ q_2) = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}
 \end{aligned}$$

Vlastní vektory  $e_1$  a  $e_2$  napřed ověříme zda jsou ortogonální, pokud jsou tak znormalizujeme. Pokud by nebyly ortogonální, tak je musíme ortogonalizovat pomocí Gram-Schmidtova ortogonalizačního procesu (viz obrázek níže). Do diagonální matice  $D$  patří na diagonálu vlastní čísla matice.

Gram-Schmidt ortogonalizační proces

$$\textcolor{teal}{1.} \quad f_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, f_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Ad 1. Nejprve  $f_1 = e_1$ , proto:

$$e_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Nyní vypočteme  $e_2$

$$e_2 = f_2 - \alpha e_1$$

Nyní spočteme

$$\alpha = \frac{(f_2, e_1)}{(e_1, e_1)} = \frac{1.2 + 0.1}{2.2 + 1.1} = \frac{2}{5}$$

Nyní jsme schopni spočítat  $e_2$

$$e_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \frac{2}{5} \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{5} \\ -\frac{2}{5} \end{pmatrix}$$

e1 a e2 jsou ortogonální (kolmé k sobě)

## Geršgorinova věta

- Slouží k lokalizování spektra matice (množina všech **vlastních čísel** matice)

**Věta:** (Geršgorinova)

Nechť  $A = [a_{ij}]$  je komplexní čtvercová matice řádu  $n$ . Nechť

$$r_i = |a_{ii}| + \dots + |a_{i,i-1}| + |a_{i,i+1}| + \dots + |a_{ii}|, S_i = \{z \in C : |z - a_{ii}| \leq r_i\}, i = 1, \dots, n.$$

Pak  $\sigma(A) \subset S_1 \cup \dots \cup S_n$ .

**Příklad:**

Pomocí Geršgorinovy věty lokalizujte spektrum matice

$$A = \begin{bmatrix} 1+i & -1 & 0 \\ -1 & 4 & i \\ 0 & -1 & 2 \end{bmatrix}.$$

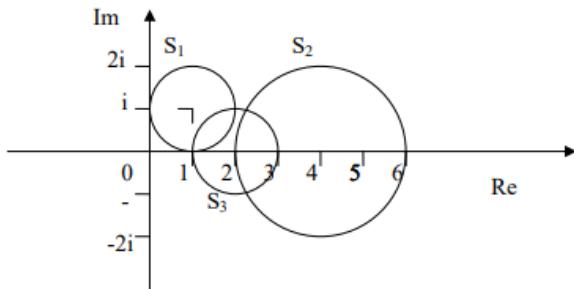
**Řešení:**

$$r_1 = |-1| + |0| = 1, \quad S_1 = \{z \in C : |z - (1+i)| \leq 1\},$$

$$r_2 = |-1| + |i| = 2, \quad S_2 = \{z \in C : |z - 4| \leq 2\},$$

$$r_3 = |0| + |-1| = 1, \quad S_3 = \{z \in C : |z - 2| \leq 1\}.$$

Nalezené množiny vykreslíme do komplexní roviny:



Spektrum se pak nachází ve sjednocení všech tří kruhů  $\sigma(A) \subset S_1 \cup S_2 \cup S_3$ .

- $S_i$  se dovnitř do absolutní hodnoty dávají hodnoty z diagonály.

- Diferenciální a integrální počet funkcí jedné proměnné (posloupnosti, limita a spojitost, derivace, extrémy, neurčité a určité integrály).**

### posloupnosti

- speciální případ funkce, definované na množině přirozených čísel
- posloupnost je zobrazení ( $a: N \rightarrow R$ ) jehož hodnoty značíme obvykle  $a_n$ , celou posloupnost značíme  $\{a_n\}_{n=1}^{\infty}$
- je zadána buď výčtem nebo rekurentním vzorcem ( $a_{n+1} = a_n + d, a_1 = 1$ ) nebo vzorcem pro  $n$ -tý člen ( $a_n = 1/n$ )
- **limita posloupnosti** = "Řekneme, že posloupnost  $\{a_n\}$  má limitu  $A$ , jestliže ke každému  $\epsilon > 0$  existuje  $n_0 \in N$  takové, že pro každé  $n \geq n_0$  platí, že  $|a_n - A| < \epsilon$ ."

- limita posloupnosti nám pomáhá porozumět chování zadané posloupnosti pro velká  $n$
- pokud je hodnota limity nějaké konkrétní číslo = jedná se o *vlastní limitu posloupnosti*
  - pokud má posloupnost vlastní limitu tak se jedná o *konvergentní*
- pokud je hodnota limity +- nekonečno = *nevlastní limita posloupnosti*
  - posloupnost *divergentní*
- definice limity
  - $\forall \varepsilon > 0 \exists n_0 \forall n \geq n_0 : |a_n - a| < \varepsilon$
  - pro všechna epsilon větší než 0 existují taková  $n$  větší nebo rovna  $n_0$ , pro která platí  $|a_n - a| < \varepsilon$  ( $a$  = hodnota limity, epsilon definuje nějaké okolí kolem té hodnoty limity)
  - tzn. pro mnou nalezenou hodnotu limity ať už si zvolím jakoukoli hodnotu epsilon (= jakkoli velké okolí kolem limity) tak pokud se v tom okolí nachází prvek  $a_n$  tak se v tom okolí nachází všechny následující prvky ( $a_{n+1}, a_{n+2}, \dots$ )
- věty o limitách
  - 1, každá posloupnost má nejvýše 1 limitu
  - 2, každá konvergentní posloupnost je omezená (posloupnost někde začíná -  $n=0$  a z druhé strany je omezená právě tou limitou)
  - 3, pokud posloupnosti  $a_n$  a  $b_n$  jsou konvergentní tak:
    - $\lim(a_n + b_n)$  je také konvergentní  $\Rightarrow \lim(a_n) + \lim(b_n)$
    - $\lim(a_n - b_n)$  je také konvergentní  $\Rightarrow \lim(a_n) - \lim(b_n)$
    - $\lim(a_n * b_n)$  je také konvergentní  $\Rightarrow \lim(a_n) * \lim(b_n)$
    - $\lim(a_n / b_n)$  je také konvergentní  $\Rightarrow \lim(a_n) / \lim(b_n)$  (!nesmí být 0)
    - $\lim(c * a_n)$  je také konvergentní  $\Rightarrow c * \lim(a_n) = c * a$  ( $c$  je nějaká konstanta z oboru reálných čísel)
- počítání limit posloupností
  - do výrazu posloupnosti dosadím za  $n$  nekonečno a pokud nedostanu *neurčitý výraz* tak jsem schopen vyčíst hodnotu limity
  - pokud dostanu neurčitý výraz tak bude potřeba ještě nějaká úprava výrazu posloupnosti
    - neurčitý výraz:  $0/0, \infty/\infty, \infty-\infty, 0^*\infty, 0^0, \infty^0, 1^\infty$
    - $\infty + \infty = \infty, -\infty - \infty = -\infty, -\infty * \infty = -\infty, (-\infty) * (-\infty) = \infty$
    - $\infty \pm k = \infty, -\infty \pm k = -\infty, k * \infty = (k > 0) \Rightarrow \infty, k * \infty = (k < 0) \Rightarrow -\infty$
    - $k / \infty = 0, n$ -tá odmocnina z  $\infty$  je  $\infty, \infty^n = \infty$
    - $a^\infty$  (pokud  $a \in (0,1)$  tak výsledek je 0) (pokud  $a > 1$  tak výsledek je  $\infty$ )

$$\lim \frac{n^2 \left(3 + \frac{1}{n^2}\right)}{n^2 \left(\frac{3}{n} + 1\right)} = \lim \frac{3 + \frac{1}{n^2}}{\frac{3}{n} + 1} = \frac{3 + 0}{0 + 1} = 3.$$

- isibalo bomber dobře vysvětuje limity posloupností tady a v následujících videích
-  22 - Konvergentní posloupnosti a vlastní limita (MAT - Posloupnosti a nekonečné řady)

## limita a spojitost

### limita funkce

4 typy:

- vlastní limita ve vlastním bodě
  - $\lim_{x \rightarrow a} (f(x)) = A$
  - $\lim_{x \rightarrow 4} (x^2) = 16$
  -

- vlastní limita v nevlastním bodě
  - $\lim_{x \rightarrow \infty} (f(x)) = A$
  - $\lim_{x \rightarrow \infty} (2/x) = 0$
  -
- nevlastní limita ve vlastním bodě
  - $\lim_{x \rightarrow a} (f(x)) = +\infty$
  - $\lim_{x \rightarrow 0} (2/x) = +\infty$
  -
- nevlastní limita v nevlastním bodě
  - $\lim_{x \rightarrow \infty} (f(x)) = +\infty$
  - $\lim_{x \rightarrow \infty} (x+2) = +\infty$

počítání - u limit v nevlastním bodě stejně jako u limit posloupností, u těch ve vlastním bodě prostě dosazuju

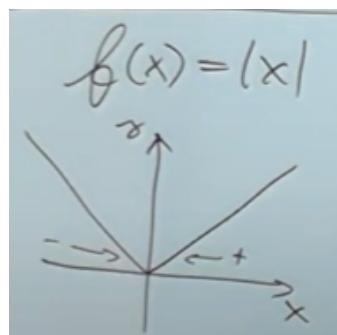
## derivace

### derivace v bodě

- je směrnice tečny v daném bodě
- derivace v bodě nám řekne, jak funkce v daném bodě roste nebo klesá
- k výpočtu můžeme použít limity - snažíme se najít směrnici co nejmenší možné sečny grafu dané funkce

$$\lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

- kdy derivace v daném bodě existuje?
  - pokud se derivace v daném bodě zleva rovná derivaci v daném bodě z prava
  - "pokud je v grafu funkce špička tak funkce v tom bodě nemá derivaci"
  - derivace zleva = k bodu x se blížím zleva na ose x, tedy ze záporných čísel
  - derivace zprava = k bodu x se blížím z prava na ose x, tedy ze kladných čísel
  - např. funkce *absolutní hodnota* nemá derivaci v bodě 0, protože limita zleva je rovna -1 a limita zprava = 1



- výpočet derivace funkce  $f(x) = x^2$  v bodě  $x_0 = 2$  (lze počítat i bez znalosti derivačních vzorců, ale díky nich se pak toto počítání mnohem zrychlí):

$$\begin{aligned}
 a = f'(z) &= \lim_{x \rightarrow z} \frac{f(x) - f(z)}{x - z} = \\
 &= \lim_{x \rightarrow z} \frac{x^2 - 4}{x - z} = \lim_{x \rightarrow z} \frac{(x-z)(x+z)}{x-z} = \\
 &= \lim_{x \rightarrow z} (x+z) = z + z = \underline{\underline{4}}
 \end{aligned}$$

derivace funkce na intervalu

- použitím derivačních vzorců získám jinou funkci, do které když dosadím bod tak zjistím směrnici původní derivované funkce v tomto bodě
- derivační vzorce

|      |                                                  |
|------|--------------------------------------------------|
| (1)  | $(c)' = 0$                                       |
| (2)  | $(x^n)' = nx^{n-1}$                              |
| (3)  | $(a^x)' = a^x \ln a$                             |
| (4)  | $(e^x)' = e^x$                                   |
| (5)  | $(\log_a x)' = \frac{1}{x \ln a}$                |
| (6)  | $(\ln x)' = \frac{1}{x}$                         |
| (7)  | $(\sin x)' = \cos x$                             |
| (8)  | $(\cos x)' = -\sin x$                            |
| (9)  | $(\operatorname{tg} x)' = \frac{1}{\cos^2 x}$    |
| (10) | $(\operatorname{cotg} x)' = -\frac{1}{\sin^2 x}$ |
| (11) | $(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$          |
| (12) | $(\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$         |
| (13) | $(\operatorname{arctg} x)' = \frac{1}{1+x^2}$    |
| (14) | $(\operatorname{arcotg} x)' = -\frac{1}{1+x^2}$  |

- derivace složených funkcí

- složené funkce = funkce, které mají jako svůj argument nějakou jinou funkci

- postup
  - zderivuj tu vnější funkci a tu vnitřní nechám v argumentu netknutou a potom výsledek vynásobím s derivací té vnitřní funkce

$$\underline{(\underline{f(g(x))})'} = \underline{f'(g(x))} \cdot g'(x)$$

- l'Hospitalovo pravidlo? ještě to rozvadet?
  - pomuze nam v některých obtížných případech vypočítat limitu podílu dvou funkcí
  - aby se dalo použít musí být splněny 2 podmínky ( limity obou funkcí musí být 0 a nebo limita absolutní hodnoty jmenovatele musí jít do +nekonečna)
  - pokud ty podmínky platí tak můžu říct, že *limita podílu těchto dvou funkcí je rovna limitě podílu derivací těchto funkcí*

## extrémy

- extrémy funkce je jedna z charakteristik průběhu funkce
- existují další charakteristiky popisující průběh funkce jako je :
- definiční obor, sudost a lichost, průsečíky s osami, konvexnost a konkávnost,...
- průběh funkce je potřeba počítat v případě neelementárních předpisů funkcí (např.  $f(x) = \ln x + \sin x - 1/x$ , u této funkce na první pohled asi těžko nakreslím její graf), u těch elementárních jsme schopni říct jaký má funkce průběh relativně rychle bez počítání
- intervaly monotónnosti funkce (intervaly kde funkce roste nebo klesá)
  - 1. určím definiční obor funkce
  - 2. vypočítám první derivaci funkce
  - 3. položím první derivaci rovnu nule a vypočítám čemu se rovná  $x$  - tím zjistím nulové body první derivace funkce
  - 4. nakreslím si osu  $x$  a na ní zaznačím definiční obor funkce a nulové body její první derivace - nulové body mi rozdělují definiční obor na intervaly
  - 5. z vzniklých intervalů si vyberu vhodně hodnotu (nevybírat nulové body) a dosadím ji do funkce - pokud je výsledek kladný - funkce v daném intervalu roste (pokud záporný tak klesá)
- lokální extrém funkce
  - funkce má v bodě  $M$  lokální maximum/minimum, pokud existuje nějaké okolí bodu, pro jehož všechny body platí, že funkční hodnota těchto bodů není menší/větší než funkční hodnota bodu  $M$
  - "lokální extrém je tam, kde derivace dané funkce rovna 0" - !! ale pozor např. u funkce *absolutní hodnota* v lokálním minimu funkce není definována !!
- výpočet lokálního extrému funkce:
  - 1. zjistím intervaly monotónnosti funkce rozdělené nulovými body první derivace
  - 2. podívám se na nulové body první derivace a pokud se v daném bodě funkce mění z rostoucí na klesající => jedná se o lokální maximum ( změna z klesající na rostoucí => lokální minimum)
    - "Funkce  $f$  má v bodě  $M \in D(f)$  lokální maximum, pokud existuje nějaké okolí  $U = (M-\epsilon, M+\epsilon)$ , kde  $\epsilon > 0$  takové, že pro všechna  $x \in U \cap D(f)$  platí  $f(x) \leq f(M)$ ."

$f: \gamma = \frac{\ln x}{x} \quad ; \quad g: \gamma = x^3 - 3x \quad ; \quad h: \gamma = \frac{x^2}{x-1}$

$g: \gamma = x^3 - 3x \quad ; \quad D_g = \mathbb{R} \quad \text{not} \quad \cancel{\gamma}$

$(x) = (x^3 - 3x)' = 3x^2 - 3 = 3(x^2 - 1) \quad \sqrt{x^2} = |x|$

$\begin{array}{c} \oplus \\ -1 \end{array} \quad \begin{array}{c} \ominus \\ 1 \end{array} \quad \begin{array}{c} \oplus \\ \end{array}$

$\rightarrow 3(x^2 - 1) = 0 / :3 \quad x^2 - 1 = 0 \quad |x| = 1$

$x^2 = 1 \quad / \sqrt{} \quad \rightarrow x = 1 \quad x = -1$

Postouci  $\rightarrow (-\infty; -1) \cup (1; +\infty)$   
 Hesajici  $\rightarrow (-1, 1)$   
 Loka/na max  $\rightarrow x = -1$   
 Loka/na min  $\rightarrow x = 1$

- globální extrémy funkce
  - potřebujeme mít zadaný interval na kterém hledám globální extrém
  - pokud je v daném intervalu více maxim/minim tak chci najít to největší maximum nebo nejmenší minimum
  - výpočet:
    - 1. najdu lokální extrémy
    - 2. porovnám funkční hodnoty lokálních extrémů a ještě i krajních bodů intervalu (pokud je uzavřený) a ten s nejvyšší funkční hodnotou je globální maximum (ten s nejnižší je globální minimum)

$f: \gamma = x^3 - 12x + 20 \quad ; \quad \langle -5, 5 \rangle \quad ; \quad (-5, 5), \langle -5, 5 \rangle, \langle -5, 5 \rangle$

$f'(x) = 3x^2 - 12 = 3(x^2 - 4) \quad \downarrow$

$\begin{array}{c} \oplus \\ 5 \end{array} \quad \begin{array}{c} \ominus \\ -2 \end{array} \quad \begin{array}{c} \oplus \\ 2 \end{array} \quad \begin{array}{c} \oplus \\ 5 \end{array}$

$x^2 - 4 = 0 \quad \downarrow$

$x = 2 \quad x = -2$

$f(5) = 125 - 60 + 20 = 85 \quad \rightarrow \text{absolutni max}$   
 $f(2) = 8 - 24 + 20 = 4$   
 $f(-2) = -8 + 24 + 20 = 36$   
 $f(-5) = -125 + 60 + 20 = -45 \quad \rightarrow \text{absolutni min}$

## neurčité a určité integrály

### integrály

- integrace je opačný proces k derivaci
- pomocí integrování hledám *primitivní funkci* k funkci  $f(x)$  - tzn. hledám takovou funkci  $(F(x))$ , kterou když derivuji tak získám funkci  $f(x)$
- pokud hledám *primitivní funkce*  $F(x)$  k funkci  $f(x)$ , tak takových funkcí je nekonečně mnoho, ale liší se jen o nějakou konstantu (tato konstanta mi při derivování zmizí - proto je jich nekonečně mnoho)
- způsob zápisu

$$\int f(x) dx = F(x) + C; \\ C \in \mathbb{R}$$

- čteme: integrál funkce  $f(x)$  podle proměnné  $x$  (to  $dx$  znamená, že podle proměnné  $x$ ) je roven primitivní funkci  $F(x) + C$  (= nějaká konstanta, takových primitivních funkcí je nekonečně mnoho ale liší se pouze touto konstantou)
- výpočet:
  - pomocí integračních vzorců
  - někdy pro složitější integrály je potreba metoda *per partes* nebo *substituční* metoda
- integrační vzorce

#### Pravidla pro integrování

1.  $\int k \cdot f(x) dx = k \cdot \int f(x) dx$

Funkce a exponenty

3.  $\int 0 dx = C$

4.  $\int 1 dx = x + C$

5.  $\int x^\alpha dx = \frac{x^{\alpha+1}}{\alpha+1} + C, \alpha \neq -1$

6.  $\int a^x dx = \frac{a^x}{\ln a} + C$

Logaritmy a exponenciála

7.  $\int \frac{1}{x} dx = \ln|x| + C$

8.  $\int e^x dx = e^x + C$

2.  $\int (f(x) \pm g(x)) dx = \int f(x) dx \pm \int g(x) dx$

Funkce vedoucí na goniometrické funkce

9.  $\int \cos x dx = \sin x + C$

10.  $\int \sin x dx = -\cos x + C$

11.  $\int \frac{dx}{\cos^2 x} = \operatorname{tg} x + C$

12.  $\int \frac{dx}{\sin^2 x} = -\operatorname{cotg} x + C$

Funkce vedoucí na cyklometrické funkce

13.  $\int \frac{dx}{\sqrt{1-x^2}} = \arcsin x + C$

14.  $\int \frac{dx}{1+x^2} = \arctg x + C$

#### Vzorce pro použití metod

Metoda per partes

#### Neurčitý integrál

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

#### Určitý integrál

16.  $\int_a^b u' \cdot v = [u \cdot v]_a^b - \int_a^b u \cdot v'$

Metoda substituce

#### Neurčitý integrál

17.  $\int f(g(x)) \cdot g'(x) dx = \left| \begin{array}{l} g(x) = t \\ g'(x) dx = dt \end{array} \right| = \int f(t) dt = \dots = F(t) = F(g(x)) + C$

#### Určitý integrál

18.  $\int_{g(a)}^{g(b)} f(g(x)) \cdot g'(x) dx = \left| \begin{array}{l} g(x) = t & a \rightarrow g(a) \\ g'(x) dx = dt & b \rightarrow g(b) \end{array} \right| = \int_{g(a)}^{g(b)} f(t) dt = [F(t)]_{g(a)}^{g(b)} = F(g(b)) - F(g(a))$

#### určitý integrál

- k čemu?

- umožní nám vypočítat obsah plochy pod křivkou (zadanou funkcí) na zadaném intervalu

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

- určitý integrál funkce  $f(x)$  podle proměnné  $x$  na intervalu od  $a$  do  $b$



- výpočet

$$\int_1^3 x dx = \left[ \frac{x^2}{2} \right]_1^3 = \frac{9}{2} - \frac{1}{2} = \frac{8}{2} = 4$$

- 1. integruji danou funkci a zakreslím do hranatých závorek kde do pravého horního a dolního rohu závorek napíšu ten interval na kterém to počítám
- 2. vypočítám tak, že do získaného výrazu v hranatých závorkách dosadím nejdříve tu vyšší hranici intervalu a potom tu nižší a tyto dvě hodnoty od sebe odečtu

## • Diferenciální a integrální počet funkcí více proměnných (parciální derivace, gradient, extrémy, dvojné a trojně integrály).

### funkce 2 proměnných

- "Funkce dvou proměnných je předpis, který každému bodu z  $R^2$  (tj. z roviny) přiřazuje jediné reálné číslo"
- funkce, která má na vstupu dvě proměnné
- definiční obor (co můžu dosadit za  $x$  a  $y$  funkce  $f(x,y)$ ) je oblast v rovině

### derivace funkce 2 proměnných

- derivace je směrnice tečny v daném bodě
- Kvůli toho, že graf funkce není v rovině, ale v prostoru, nemůžeme rovnou určovat derivaci v daném bodě. Je potřeba se nejprve omezit na jednu z os ( $x$  nebo  $y$ ) - tím se nám z prostorové zobrazení grafu stane rovinné zobrazení - získám nárys nebo bokorys (podle toho jestli se omezím na  $x$  nebo  $y$ ) grafu funkce

- a až následně mohu řešit derivace - kvůli toho, že jsem se omezil tak se tomu říká parciální derivace ve směru osy x nebo y
- výpočet parciální derivace obecně
  - prostě to derivuju podle jedné z těch proměnných a na tu druhou se dívám jakoby to byla nějaká konstanta

$$f(x,y) = xy + y - x^2 - 3y^2 - 2x$$

$$f_x = y - 2x - 2$$

-  $f_x$  znamená parciální derivace ve směru x

- parciální druhá (třetí atd.) derivace
  - řeší se stejně jako ta první - prostě derivuju zase podle některé z proměnných a na tu druhou se dívám jako na konstantu
- výpočet parciální derivace v bodě

$$f(x,y) = x^2 + y^2 \quad (x_0, y_0) = (x_0, y_0)$$

$$f_x(y_0) = (x^2 + y)^' = 2x \rightarrow f_x(x_0, y_0) = 2x_0$$

$$f_y(x_0) = (1 + y^2)' = 2y \rightarrow f_y(x_0, y_0) = 2y_0$$

- výpočet parciálních derivací funkce  $f(x) = x^2 + y^2$  v bodě (1,2)

- parciální derivace ve směru x
  - 1. do funkce dosadím y hodnotu z bodu (získám  $f_x(x,2) = x^2 + 4$ )
  - 2. získanou funkci zderivuju (získám  $f'_x(x,2) = 2x$ )
  - 3. dosadím hodnotu x bodu (získám hodnotu 2, což je směrnice tečny daného bodu ve směru x)

### Taylorův polynom?

- k approximaci přímky

### gradient

- Gradient v bodě M zadáné funkce  $f(x,y)$  je vektor, který mi říká, kterým směrem se mám vydat z bodu M, abych dostával co nejrychlejší přírůstky hodnot. (pokud je samozřejmě gradient nenulový)
- [Když si představím, že stojím v nějakém kopci, tak kterým směrem se mám vydat, abych šel co nejvíce do kopce a byl tedy co nejrychleji nahoře.]
- výpočet:
  - Gradient je vektor, jehož x souřadnicí je parciální derivace funkce ve směru proměnné x. Y souřadnice je parciální derivace funkce ve směru proměnné y.

$$f(x_0, y_0) = x^2 + 2y^2 \quad (1,1)$$

$$\text{grad } f(x_0, y_0) = (f_x(x_0, y_0), f_y(x_0, y_0))$$

$$f_x = 2x \rightarrow f_x(1,1) = 2 \quad \text{grad } f(1,1) = (2, 4)$$

$$f_y = 4y \rightarrow f_y(1,1) = 4$$

- 1. určím parciální derivace ve směru x a ve směru y

- 2. pak do nich dosadím a mám x a y souřadnici gradientu v bodě
- využívám ho při derivaci funkce o více proměnných v libovolném směru zadaným vektorem
- výpočet derivace funkce v libovolném směru (při parciálních derivacích jsme se omezovali pouze na směr osy x a směr osy y)
  - derivace funkce  $f(x,y)$  v daném směru  $u$  je dána skalárním součinem gradientu této funkce a vektoru  $u$

$$\langle \text{grad } f(x_0) ; \vec{u} \rangle$$

- 
- 1. vypočítám gradient

$$\begin{aligned} f_x &= 2x - 3y & = (2x - 3y) \\ f_y &= 4y - 3x & = -4x \\ \text{grad } f(x_0) &= (2x - 3y; 4y - 3x) \end{aligned}$$

- 2. vypočítám skalární součin gradientu a vektoru  $u$ 
  - tzn. vynásobím x a y souřadnice gradientu a vektoru a tyto součiny pak sečtu

$$\begin{aligned} \langle \text{grad } f(x_0) ; \vec{u} \rangle &= \langle (2x - 3y; 4y - 3x) ; (-1, 1) \rangle = \\ 2x - 3y &= (2x - 3y) \cdot (-1) + (4y - 3x) \cdot 1 = \\ 4y - 3x &= -4x + 6y + 4y - 3x = 10y - 7x \end{aligned}$$

- celý příklad na určení derivace funkce v libovolném směru v daném bodě:

Vypočítejte derivaci funkce  $f(x_0) = x^2 + 2y^2 - 3xy$  v bodě  
 $A = [1; 2]$  ve směru vektoru  $\vec{u} = (-2, 1)$ . (13)

$$\begin{aligned} \langle \text{grad } f(x_0) ; \vec{u} \rangle &= \langle (2x - 3y; 4y - 3x) ; (-1, 1) \rangle = \\ f_x &= 2x - 3y = (2x - 3y) \cdot (-1) + (4y - 3x) \cdot 1 = \\ f_y &= 4y - 3x = -4x + 6y + 4y - 3x = 10y - 7x \\ \text{grad } f(x_0) &= (2x - 3y; 4y - 3x) \end{aligned}$$

## extrémy

- extrém je bod, pro který platí, že všechny body z jeho okolí mají větší/menší funkční hodnotu
- výpočet

Najdete lokální extrémy funkce:  
 $f(x_0) = xy + y - x^2 - 3y^2 - 2x$

- 1. určím definiční obor funkce

$$D_f = \mathbb{R}^2$$

- 2. najdu podezřelé (stacionární) body - tj. body ve kterých je funkční hodnota derivace ve směru x i y rovna 0
  - vypočítám parciální derivaci ve směru x a ve směru y

$$f_x = y - 2x - 2$$

$$f_y = x + 1 - 6y$$

- a to co mi vyjde dám rovno nule
- vznikne mi soustava 2 rovnic o 2 neznámých - tuto soustavu vyřeším

$$\begin{array}{l} y - 2x - 2 = 0 \rightarrow y = 2x + 2 = 2(-1) + 2 = 0 \\ x + 1 - 6y = 0 \\ \hline x + 1 - 6(2x + 2) = 0 \\ x + 1 - 12x - 12 = 0 \\ -11x = 11 \quad |:(-11) \\ x = -1 \end{array}$$

- vyřešením této soustavy rovnic získám nulový bod (neboli stacionální nebo podezřelý bod)

$$(x; y) = (-1; 0)$$

- 3. vypočítám determinant D (ten se skládá z druhých derivací zadáné funkce)

$$D = \begin{vmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{vmatrix}$$

$$\begin{array}{l} f_x = y - 2x - 2 \\ f_y = x + 1 - 6y \\ \hline f_{xx} = -2 \\ f_{xy} = 1 \\ f_{yx} = 1 \\ f_{yy} = -6 \end{array}$$

- 4. dosadím do determinantu podezřelý bod a vyčíslím determinant  $\rightarrow D = f_{xx} * f_{yy} - f_{yx} * f_{xy}$

$$\begin{vmatrix} -2 & 1 \\ 1 & -6 \end{vmatrix} = (-2) \cdot (-6) - 1 \cdot 1 = \\ = 12 - 1 = \underline{\underline{11}}$$

- 5. mohou nastat 3 situace
  - $D < 0$  : bod není extrém ( graf funkce vypadá jako koňské sedlo)
  - $D = 0$  : nelze rozhodnout tímto způsobem jestli tento bod extrém
  - $D > 0$  : bod je extrém
    - dosadím bod do  $f_{xx}$  (parciální derivace dle  $x$  a  $x$ )
    - pokud výsledek dosazení do  $f_{xx}$ :
      - je větší než 0 = bod je lokální minimum
      - je menší než 0 = bod je lokální maximum

### dvojné a trojné integrály

- podobně jako u jednoduchých integrálů - hledám takovou funkci  $F(x,y)$ , kterou když zderivuju podle  $x$  a potom podle  $y$  (nebo obráceně, je to jedno) tak dostanu tu původní funkci  $f(x,y)$
- neurčitý integrál
  - při neurčitých integrálech funkce 1 proměnné byly řešením primitivní funkce, kterých bylo nekonečně mnoho s tím, že se lišily pouze o nějakou konstantu  $c$

$$\int f(x) dx = F(x) + C$$

- toto už u integrálů funkcí o více proměnných takto zapsat nelze - proto se neurčité integrály moc nepočítají a počítají se spíše ty určité
- určitý integrál na oblasti
  - výsledkem je nějaké číslo
- dvojný integrál = integrál funkce 2 proměnných
  - výsledkem dvojného integrálu na oblasti je *objem* ( u integrálu pro funkci 1 proměnné to byla plocha)
  - výpočet:
    - 1. rozdělím dvojný integrál na dvojnásobný integrál
      - tzn. určím že nejprve budu integrovat podle  $x$  a potom podle  $y$
    - 2. spočítám vnitřní určitý integrál pro daný interval
      - zbyde mi nějaká funkce obsahující tu proměnnou podle které jsem NEINTEGROVAL
    - 3. následně spočítám určitý integrál pro výraz který mi zbyl z vnitřního integrálu

$$\begin{aligned} \iint (2x - 4y + 5) dx dy &= \int_{-1}^3 \left( \int_{-1}^y (2x - 4y + 5) dx \right) dy = \int_{-1}^3 \left( \left[ x^2 - 4xy + 5x \right]_{-1}^y \right) dy = \\ &= \int_{-1}^3 (4 - 8y + 10 - 4y^2 + 5y) dy = \int_{-1}^3 (18 - 4y^2) dy = \left[ 18y - \frac{4}{3}y^3 \right]_1^3 = \end{aligned}$$

$$\begin{aligned} &= 54 - 54 - 18 + 6 = \\ &= -12 \end{aligned}$$

- trojný integrál = integrál funkce 3 proměnných

**Příklad otázky:** Jako analytik máte navrhnut optimální přiřazení úkolů jednotlivým pracovním týmům. Jakým grafem budete problém modelovat? Jaké aspekty problému charakterizuje hrana,

barvení hran, stupeň vrcholu? Jak při řešení využijete toky v sítích?

Verze okruhů k SZZ z 21.2.2022