

SPŠE Ječná

TEORETICKÁ INFORMATIKA

Jakub Mazuch



Praha, 2022

Jakub Mazuch


Střední průmyslová škola elektrotechnická, Praha 2, Ječná 30

Ječná 30,

121 36 Praha 2

Verze skript: 2.10 [poslední aktualizace: 31.12.2021, 13:15:00]

Skripta jsou zcela nová. Neprošla žádnou revizí a mohou jistě obsahovat chyby. Za upozornění na ně budu velmi vděčen (mazuch@spsejecna.cz).

Kapitoly označené  jsou v přípravě, rozpracované či nedokončené. Jejich obsah si prosím nastudujte z příložených zdrojů.

Základní informace

<u>Vyučující:</u>	Jakub Mazuch
<u>Kontakt:</u>	mazuch@spsejecna.cz
<u>Konzultační hodiny:</u>	po předchozí e-mailové domluvě

Další výukové podklady jsou dostupné prostřednictvím školního e-learningu.
(moodle.spsejecna.cz)

Vyžadované pomůcky

- Sešit (nelinkovaný / čtverečkovaný) \Leftrightarrow poznámky z hodin, črty, ...
- Psací potřeby (propiska, obyčejná tužka)
- Učebnice: MAZUCH Jakub. *Teoretická informatika*. 2019. Praha: SPŠE Ječná, 2019
- Kalkulačka se taky hodí

Klasifikace – hodnocené aktivity

- Písemné testy
- Ústní zkoušení
- Seminární práce
- Projekty
- Povinné domácí úkoly
- Nepovinné domácí úkoly
- Práce v hodině, aktivita, přístup/vztah ke studiu

Hodnocení probíhá v souladu s Pravidly pro hodnocení SPŠE Ječná.

Pokud je celkový průměr známek ve třídě z daného písemného projevu $\geq 3,50$, písemná práce se bude opakovat v nejbližším termínu, a to bez nároku na výmaz poslední známky.

Písemná práce se dopisuje dle uvážení vyučujícího nebo pokud počet chybějících studentů přesáhne 10 % z celkového počtu.

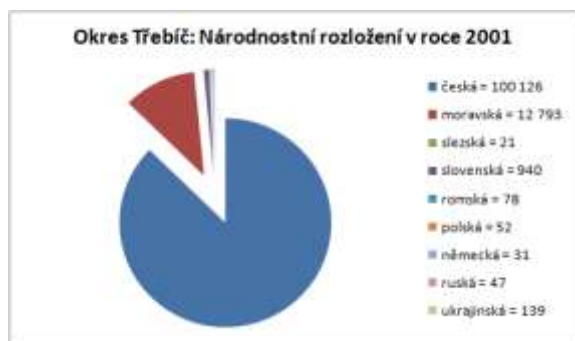
Tematický plán:

1. Activity diagram
2. Algoritmy
 - a. Třídění
 - b. Rekurze
3. UML class diagram a UML object d.
4. Objektové a datové modelování
5. Sekvenční diagram
6. Dynamické datové struktury
7. Základy teorie grafů, stromy
8. Stavový diagram
9. Kódování, elementární kryptografie a hashovací funkce
10. Složitost algoritmu
11. Umělá inteligence

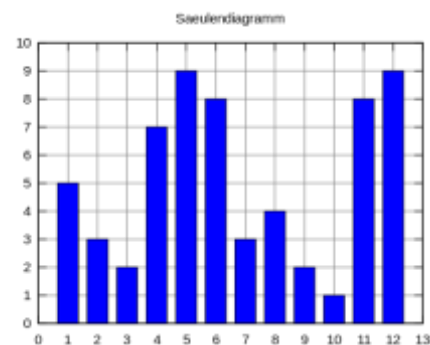
1 Diagram aktivit

1.1 Diagram

Strukturované grafické znázornění pojmů, myšlenek, vztahů, číselných, matematických nebo statistických údajů a podobně.



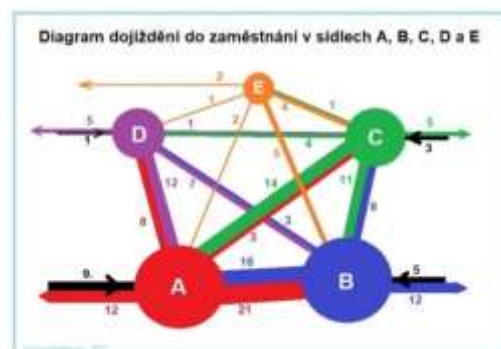
A



B



C



D

OBRÁZEK 1-1: TYPY DIAGRAMŮ (WIKIPEDIE, 2018)

Diagram A Kruhový (koláčový) diagram

Diagram B Sloupcový diagram

Diagram C Svícový diagram

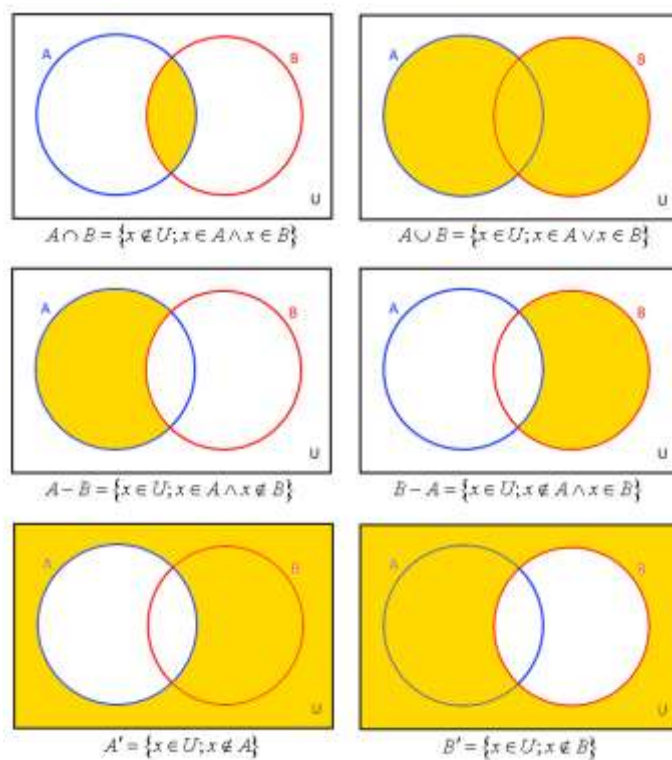
Diagram D Sankeyův diagram

Diagram nebo graf a názvy jejich součástí jsou v mnoha případech také abstrahovanými teoretickými pojmy, jejichž teorie je z grafického znázornění odvozena (například teorie grafů, graf funkce, fázový diagram a trojný bod atd.).

Slovo diagram se do češtiny dostalo přes němčinu (Diagramm) z řeckého slova diagramma, které znamená kresbu nebo obrazec a je odvozené ze slovesa diagrafo (= kreslím, přepisuji).

Diagramy mohou být založeny na různých principech:

- osový diagram: na dvou nebo *více* osách znázorňuje vztah různých veličin, například graf funkce. Osové grafy využívají ve znázornění body, čáry (liniový graf) i plochy, je z nich odvozen i sloupcový graf.
- množinové diagramy, například *Vennův diagram*



OBRÁZEK 1-2: VENNOVY DIAGRAMY (WIKIPEDIE, 2018)

- diagramy znázorňující prostorové uspořádání, například *chemický vzorec*, *mapa*, anatomické schéma, květní diagram
- diagramy znázorňující množstevní poměr, typicky kruhový diagram (například „*koláč sledovanosti*“)
- diagramy popisující změnu ceny v závislosti na čase – *svícový diagram*

- nelineární písma, například notový záznam, Braillovo písmo. Notový záznam nese zároveň rysy osového diagramu (svislá osa = výška tónu, vodorovná osa = čas)
- diagramy znázorňující časovou návaznost procesů a jejich diverzifikaci (větvení možností) – vývojový diagram, různé další procesní diagramy z oblasti lidských činností i přírodních a technických procesů, včetně grafických návodů k použití atd.
- bilanční diagram – Sankeyův diagram
- diagramy odrážející jiné aspekty struktury nebo společných vlastností, například uspořádání periodické soustavy prvků

1.2 UML (Unified Modeling Language)

Grafický standardizovaný jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů.



OBRÁZEK 1-3: LOGO UML (WIKIPEDIE, 2018)

- Soubor grafických standardů
- Využívá se při vývoji softwaru
- V oblasti analýzy – standard → důležité, aby se v něm programátoři orientovali
- Využívám v mnoha materiálech a dokumentacích
- Užitečný nástroj – usnadnění návrhu a vývoje informačního systému.
- Aktuální standardizovaná verze: UML 2.0

1.2.1 Význam a způsoby vyjádření UML

- Komplexní systémy je nejprve potřeba důkladně promyslet a navrhnout (*nelze si sednout a začít psát nějaký kód*)

→

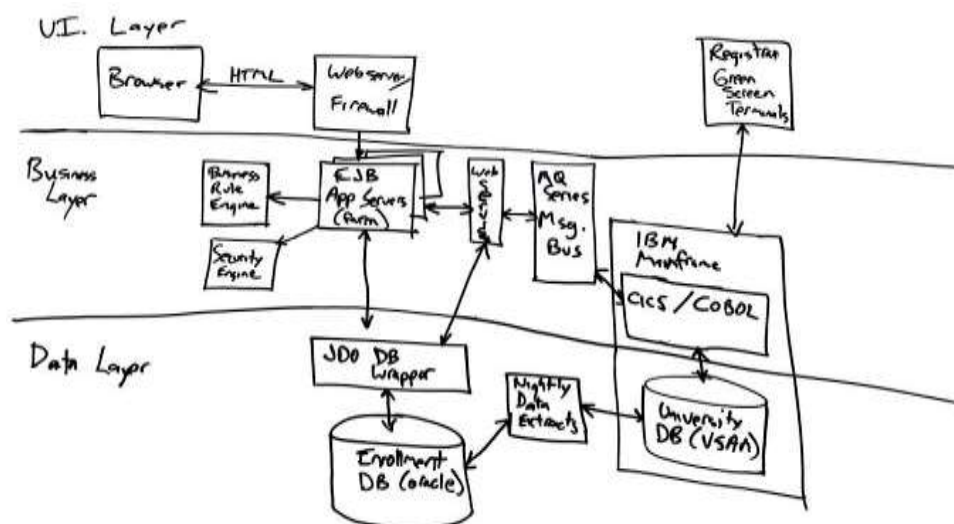
- Při implementaci nutná týmová spolupráce = dobrá komunikace v týmu
- Operativní reakce na změny požadavků klienta
- Obtížný odhad ceny systému v počátečních fázích vývoje IS

UML je nástroj, který vývojáři pomáhá se s výše uvedenými problémy vypořádat (analýza problému, odhad ceny /náklady – výdaje/, operativní změny, ...)

UML lze použít 3 způsoby:

1.2.1.1 UML jako náčrt (sketch)

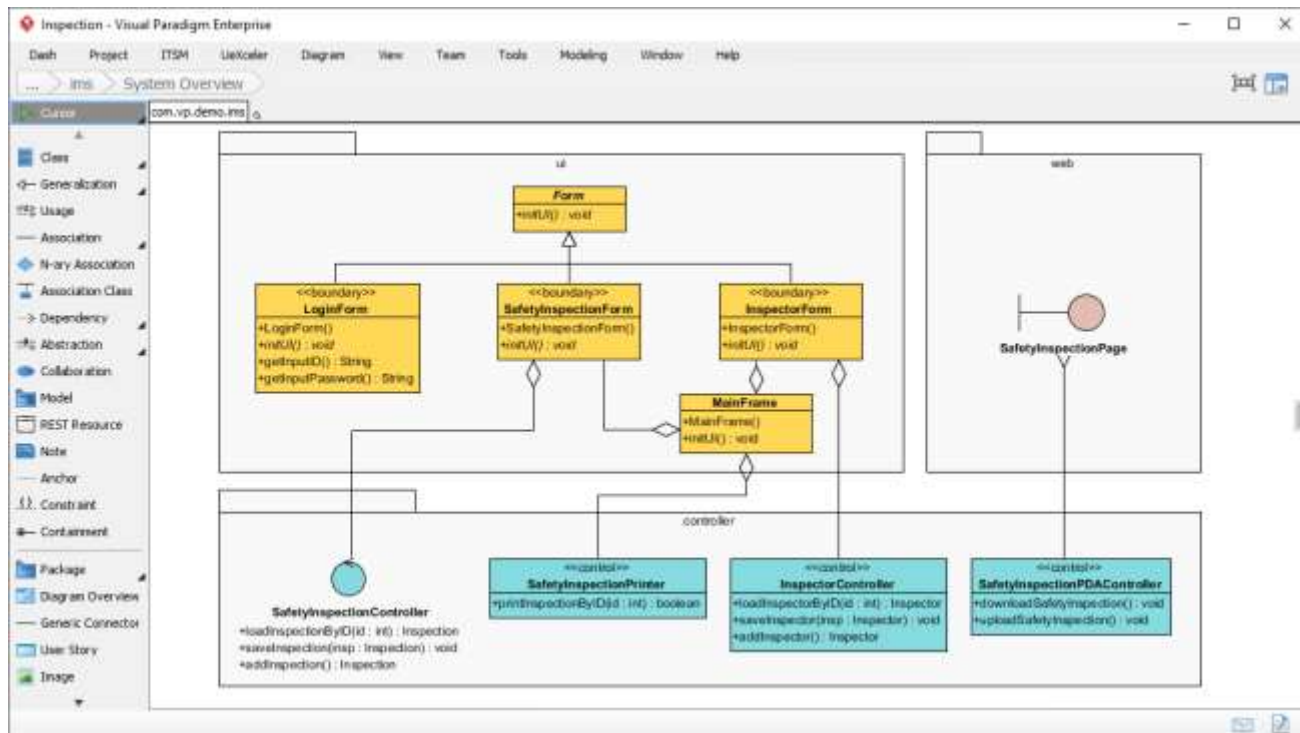
- Používáme v jednoduché podobě jako náčrt (= ručně kreslené diagramy) na tabuli, do sešitu, ...
- Grafická podoba usnadňuje komunikaci s klientem (lépe pochopíme požadavky) – grafická podoba je bližší než text
- Črtáme i průběhu návrhu systému (diskuse v týmu, operativní změny v zadání)
- Vyšší míra Abstrakce – důležitá vlastnost diagramů



OBRÁZEK 1-4:
UML SKETCH

1.2.1.2 UML jako plán (blueprint)

- Mnohonásobně detailnější než sketch
- Diagramy vytvářené v CAD systémech – *plán implementace* pro programátory
- Usnadňují komunikaci v týmu, ulehčují implementaci systému
- Programátoři se v systému lépe orientují (i nezasvěcení)
- Diagramy mohou sloužit i jako dokumentace



OBRÁZEK 1-5: UML BLUEPRINT

1.2.1.3 Programovací jazyk

- Z detailního UML lze vygenerovat šablonu kódu (základ pro implementaci)
- Využití: např. v databázích (vygenerování základacích skriptů)

1.2.2 Diagramy a jejich zařazení

UML se v současné době (verze 2.0) skládá ze 14 diagramů, které lze dále rozčlenit na 2, resp. 3 základní skupiny. Různé zdroje se v členění UML digramů rozchází.

- *Diagramy struktury* (Structure Diagrams) - Popisují strukturu systému, tedy z čeho je složený.
- *Diagramy chování* (Behaviour Diagram) - Popisují chování systému, tedy jak funguje.
- *Diagramy interakce* (Interaction diagrams) - Popisují interakci mezi jednotlivými částmi systému.

Zde je uveden výpis základních diagramů v jednotlivých skupinách

(pozn. zvýrazněné diagramy reflektuje tato učebnice)

Strukturální diagramy

- **diagram tříd**
- diagram nasazení
- diagram komponent
- diagram instancí (objektový diagram)
- diagram profilů
- diagram balíčků

Diagramy chování

- **diagram aktivit**
- diagram užití
- **stavový diagram**

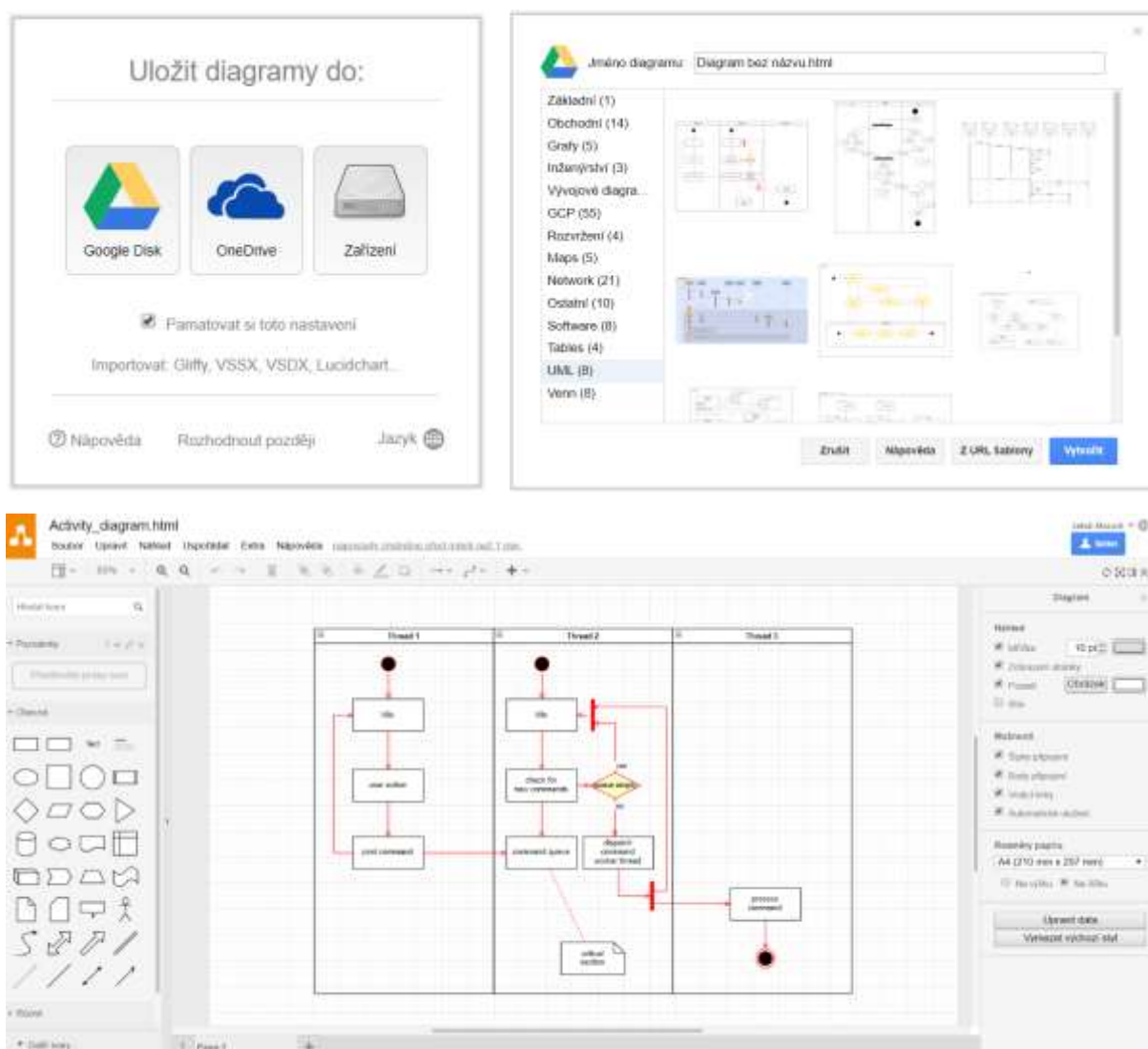
Diagramy interakce

- **sekvenční diagram**
- diagram interakcí

1.2.3 Nástroje

draw.io je zdarma on-line nástroj pro vytváření vývojových a síťových diagramů, plánů budov, myšlenkových map, Vennových diagramů, elektrických obvodů, grafů atd. Tvary pro UML diagramy vyberete v levé liště, v pravé liště se dají následně obarvovat a dostylovat. Nástroj *draw.io* umožňuje ukládat díla jednak do cloudových nástrojů (Google Drive, OneDrive), ale i na lokální disk.

Pokud stále ještě z jakési pohnutky preferujete desktop aplikace, doporučuji <https://www.modelio.org>. I tento nástroj je zdarma.



OBRÁZEK 1-6: UKÁZKA DRAW.IO

1.3 Diagram aktivit

Popis diagramu (Obrázek 1-7: Ukázka konkrétního diagramu aktivit):

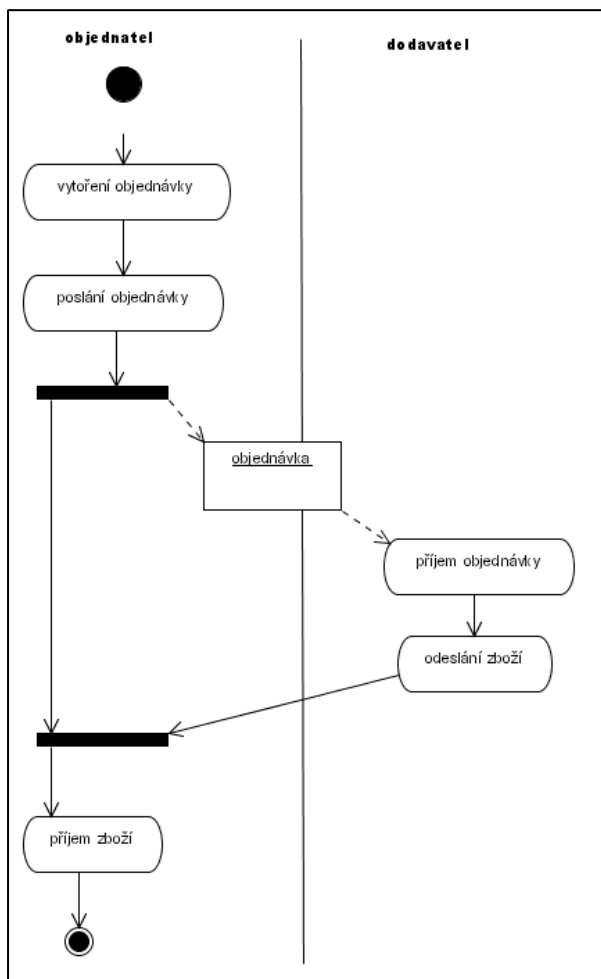


Diagram znázorňuje chování systému, který obsahuje dva objekty: objednatel, dodavatel.

Aktivitu inicializuje (symbol *inicializace*, viz kap. 1.3.2.2, str. 18) objednavatel, který vytváří objednávku (symbol *akce*, viz kap. 1.3.2.1, str. 17) a následně (symbol *hrana/šipka*, viz kap. 1.3.2.4, str. 18) posílá objednávku.

Nyní se tok dělí do více větví (paralelismus, symbol *fork*, viz kap. 1.3.2.8, str. 21).

Vytváří se třída objednávka. Dodavatel přijímá objednávku (*akce*) a následně (*hrana*) odesílá zboží (*akce*). Objednatel čeká (*hrana*) na synchronizaci (dokončení, *join* viz kap. 1.3.2.8, str. 21) aktivit na straně dodavatele.

Po synchronizaci (tj. dokončení všech aktivit na straně dodavatele), tedy odeslání zboží Objednatel přijímá zboží (*akce*) a aktivita končí (*koncový uzel*, viz kap. 1.3.2.3, str. 18).

OBRÁZEK 1-7: UKÁZKA KONKRÉTNÍHO DIAGRAMU AKTIVIT

Diagram aktivit je jeden z UML diagramů, které popisují chování. Tento diagram se používá pro modelování procedurální logiky, procesů a zachycení workflow.

Umožňuje také graficky modelovat jednotlivé případy užití jako posloupnost akcí

Každý proces v diagramu aktivit je reprezentován sekvencí jednotlivých kroků, které jsou v modelu zakresleny jako

- a) akce – atomické dále nedělitelné kroky
- b) vnořené aktivity – volání jiných procesů (aktivit), tyto aktivity mohou být reprezentovány dalším diagramem aktivit.

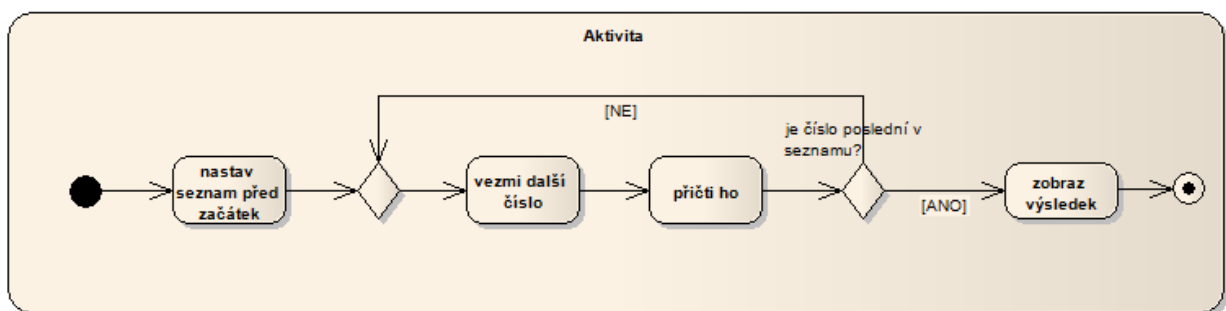
Sekvenci jednotlivých kroků v diagramu aktivit určuje řídicí tok.

1.3.1 Prvky diagramu

Diagram aktivit modeluje procesy jako aktivity, které se skládají z uzlů (nodes), vzájemně propojených hranami (edges).

Aktivita = *něco*, co lze modelovat pomocí diagramu aktivit:

- Business plán
- Workflow (= pracování / technologický postup)
- Procedurální logika (=předepsaný postup, návod jak postupovat)

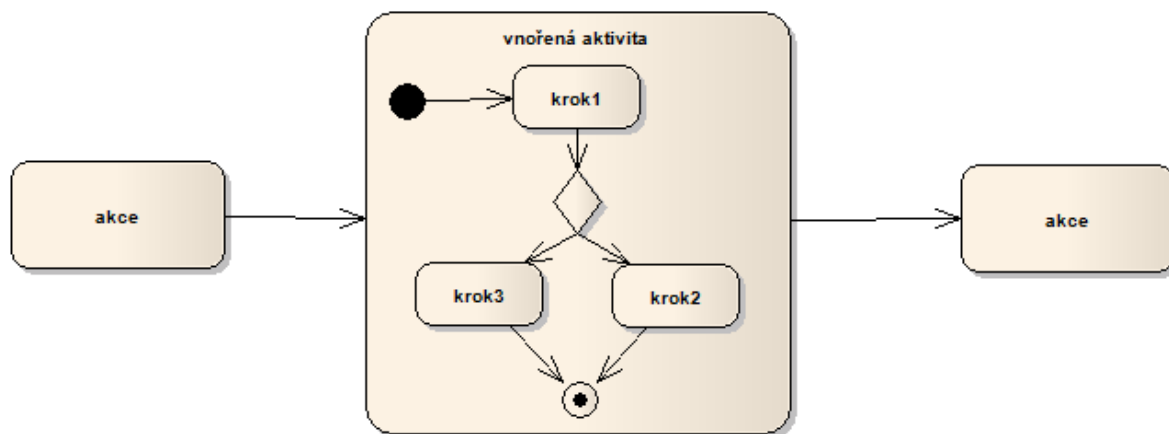


OBRÁZEK 1-8: PŘÍKLAD AKTIVITY (S, 2018)

Parametry

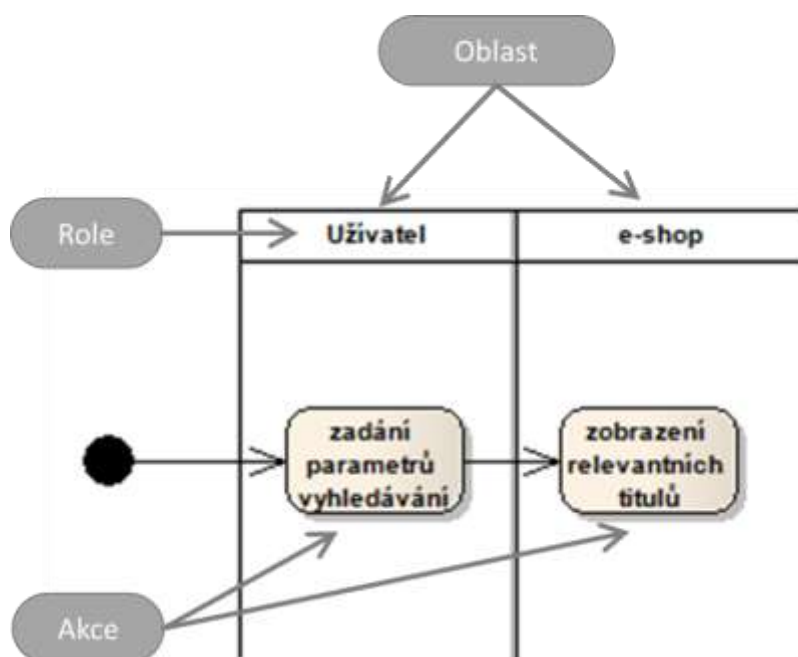
- Vstup = data / objekty jako parametry
- Činnost aktivity (spuštěna jsou-li \forall parametry naplněny)
- Výstup = data / objekty

Akce = činnost, která je aktivně vykonávána uvnitř aktivity (akce může být i *vnořená aktivita*)



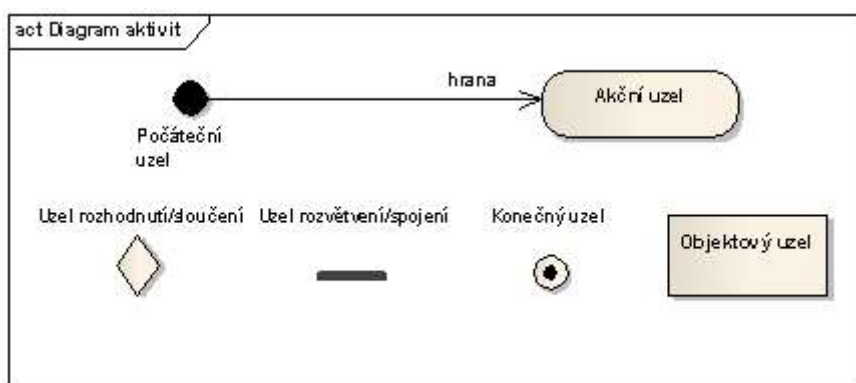
OBRÁZEK 1-9: VNOŘENÁ AKTIVITA I (S, 2018)

- Vykonávány člověkem / systémem
- V diagramu aktivit jsou jednotlivé role znázorněny jako oblasti
- Jednotlivé akce se zakreslují do dané oblasti



OBRÁZEK 1-10: VNOŘENÁ AKTIVITA II (S, 2018)

1.3.2 Tok aktivit

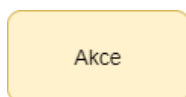


OBRÁZEK 1-11: TOK AKTIVIT (S,2018)

- (1) Akce
- (2) Inicializace aktivity (počáteční uzel)
- (3) Ukončení aktivity (koncový uzel)
- (4) Řídící tok (hrana)
- (5) Uzel rozhodnutí (větvení)
- (6) Vstupní a výstupní podmínky
- (7) Provedení kroku aktivity
- (8) Uzel sloučení (merge)
- (9) Paralelismus (fork, join)

1.3.2.1 Akce

Jednotlivé akce daného procesu znázorňujeme jako obdélník se zakulacenými rohy. Uvnitř je definován název akce (kroku).



OBRÁZEK 1-12: AKCE (ČÁPKA, 2019)

1.3.2.2 Inicializace aktivity / počáteční uzel

Aktivita začíná v bodě, který značíme symbolem inicializace (●)

Inicializačních kroků může být: více → spouští paralelní toky

nebo žádný → startovacím bodem aktivity může být třeba událost (aktivovaná ihned po inicializaci aktivity (nemá vstupní tok))

První akce = inicializovaná externím objektem (parametr aktivity)

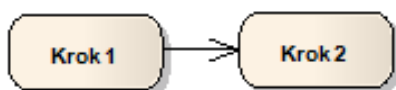
1.3.2.3 Ukončení aktivity / konečný uzel

Celá aktivita je ukončena koncovým bodem (●)

Část aktivity (tj. větev procesu) ukončujeme symbolem konce toku (⊗)

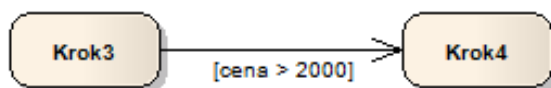
1.3.2.4 Řídící tok (hrana edge)

Posloupnost vykonávání jednotlivých kroků je definována pomocí šipek (označují řídicí tok).



OBRÁZEK 1-13: ŘÍDÍCÍ TOK (S,2018)

Řídící tok lze omezit podmínkou. Následný krok se vykoná právě tehdy když je podmínka splněna.

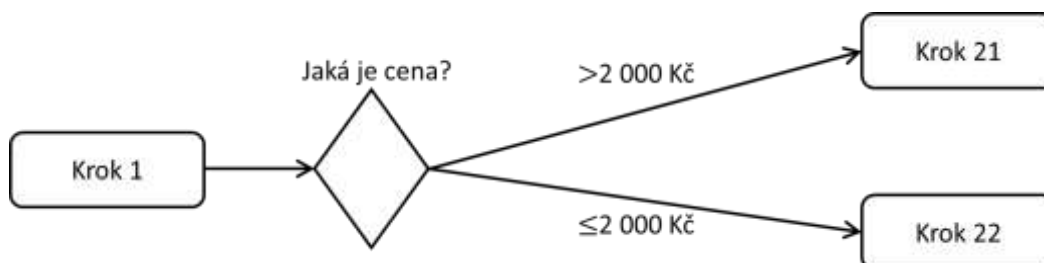


OBRÁZEK 1-14: ŘÍDÍCÍ TOK S PODMÍNKOU (S,2018)

1.3.2.5 Uzel rozhodnutí

Podmíněný tok se typicky používá na výstupech – symbol rozhodnutí (◇)

Z hlediska procesu se nejedná o krok, nýbrž o *informaci*. Tok bude pokračovat právě jednou z větví dle podmínky.

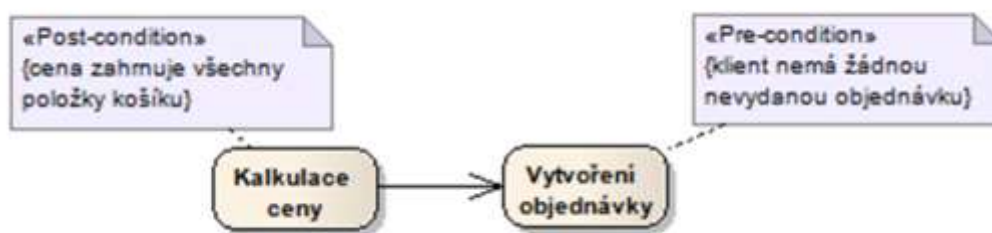


OBRÁZEK 1-15: ROZHODNUTÍ

1.3.2.6 Vstupní a výstupní podmínky

Tok aktivit lze podmínit vstupními (*pre-condition*) a výstupními (*post-condition*) podmínkami:

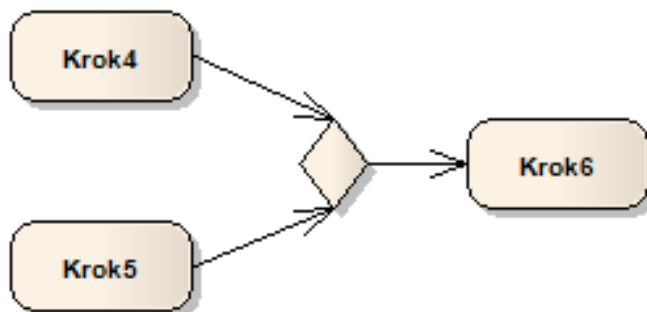
- Není-li splněna vstupní podmínka, nebude krok spuštěn
- Není-li splněna výstupní podmínka, nebude krok ukončen (předáno řízení)



OBRÁZEK 1-16: POST-CONDITION, PRE-CONDITION (S, 2018)

1.3.2.7 Spojení (merge)

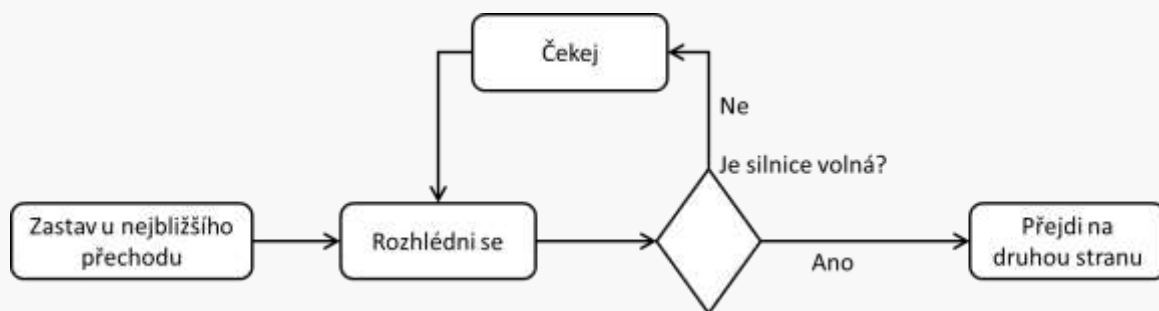
Chceme-li provést krok po provedení kroku předchozího, používáme symbol spojení (merge) (◇).



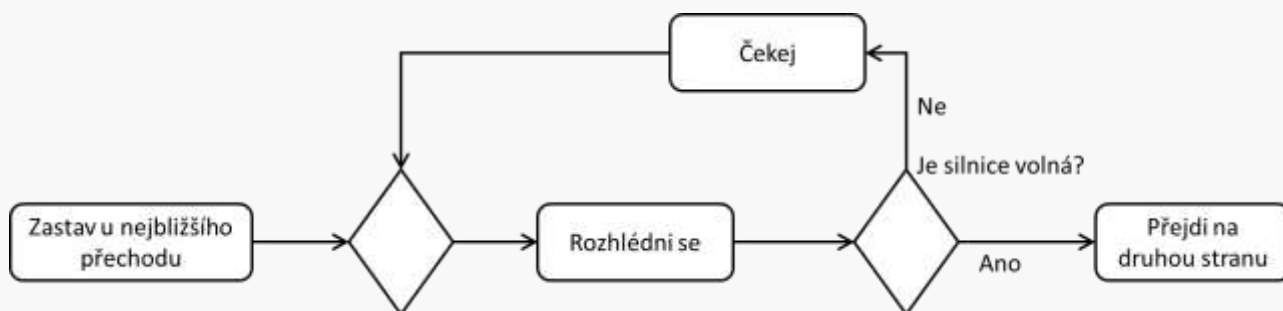
OBRÁZEK 1-17: MERGE (S,2018)

Příklad ([ad 6 Provedení kroku aktivity](#) vs. [ad 7 Merge](#))

Modelujeme jednoduchý proces, který známe z každodenního života: Přecházíme přes frekventovanou komunikaci.



a) Provedení kroku aktivity



b) Spojení (merge)

Správné řešení je vymodelováno na obrázku (b).

Vysvětlení:

Obrázek (a): U kroku „*Rozhlédni se*“ se proces zastaví a čeká se na příchod toku z kroku „*Čekej*“. Krok „*Čekej*“ se však vykoná jen za určité podmínky z kroku „*Rozhlédni se*“, který však čeká na příchod podmínky z kroku „*Čekej*“.

Obrázek (b): Krok „*Rozhlédni se*“ vyvolám buď po ukončení kroku „*Zastav u nejbližšího přechodu*“ nebo po ukončení kroku „*Čekej*“ – má pouze jeden vstupní tok.

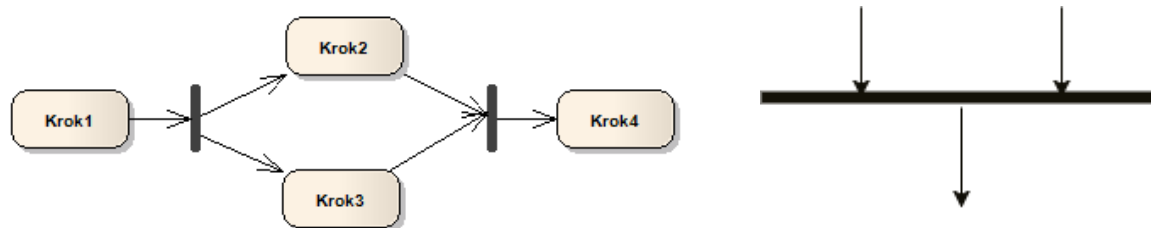
1.3.2.8 Paralelismus (fork, join)

Aktivita diagram podporuje také *parallelizaci*. Tok můžeme rozdělit na více souběžně běžících toků pomocí *fork* a tyto toky poté opět libovolně rozdělovat nebo spojovat pomocí *joinů*. Reálně se může jednat o paralelní zpracování různými vlákny nebo systémy. Forky a joiny zakreslujeme jako plné černé obdélníky, vazby s šipkou do obdélníku se spojují, vazby s šipkou z obdélníku se rozdělují.

Paralelně prováděné akce → rozdělování (fork) / spojovník (join)

Dělí tok do více větví a spouští paralelismus.

Před spojením dochází k synchronizaci (tj. čeká se na dokončení všech větví)



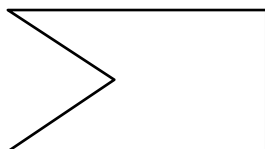
OBRÁZEK 1-18: FORK & JOIN (S, 2018), (ČÁPKA, 2019)

1.3.3 Události (Events)

- (1) Příchozí událost
- (2) Časová událost
- (3) Spuštění události
- (4) Region

1.3.3.1 Příchozí událost

Reakce procesu na externí událost ← proces reaguje na událost zvenčí

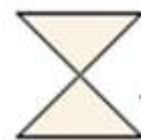


OBRÁZEK 1-19: PŘÍCHOZÍ UDÁLOST

Po každém výskytu události je spuštěn řídicí tok.

1.3.3.2 Časová událost

Reakce na uplynutí daného časového úseku od spuštění akce

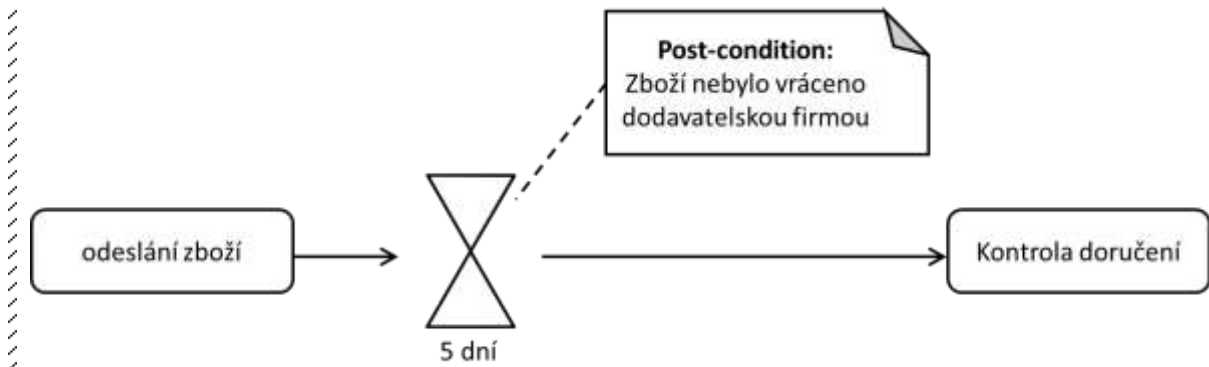


24 hod.

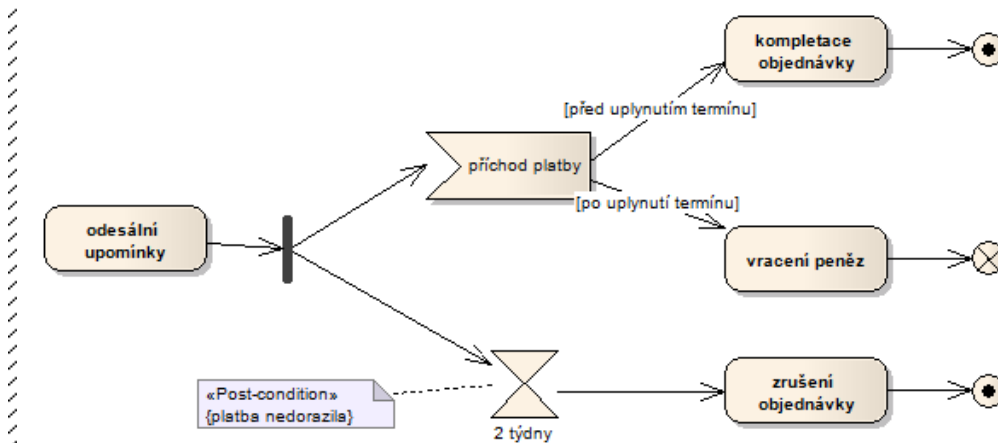
OBRÁZEK 1-20: ČASOVÁ UDÁLOST

Řídicí tok je spuštěn po uplynutí definovaného časového úseku od spuštění akce.

Cvičení: Klientovi je odesláno zboží. V případě, že zboží nebylo vráceno dodavatelskou firmou, pracovník klientského centra pět dní po odeslání volá zákazníkovi pro potvrzení, zda zboží bylo v pořádku doručeno.



Obrázek 1-21: Cvičení - Aktivita diagram I



Obrázek 1-22: Cvičení - Aktivita diagram II (S, 2018)

1.3.3.3 Spuštění události

Aktivita může generovat událost pro jiný proces.



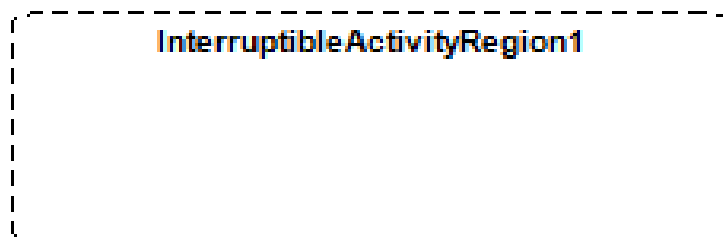
OBRÁZEK 1-23: SPUŠTĚNÍ UDÁLOSTI (S, 2018)

Generování události neovlivňuje aktuální tok aktivity (zvláštní typ akce).

1.3.3.4 Region

Po výskytu události je občas nutno zrušit provádění části procesu ⁽¹⁾ a vykonat jinou činnost (oprava, storno).

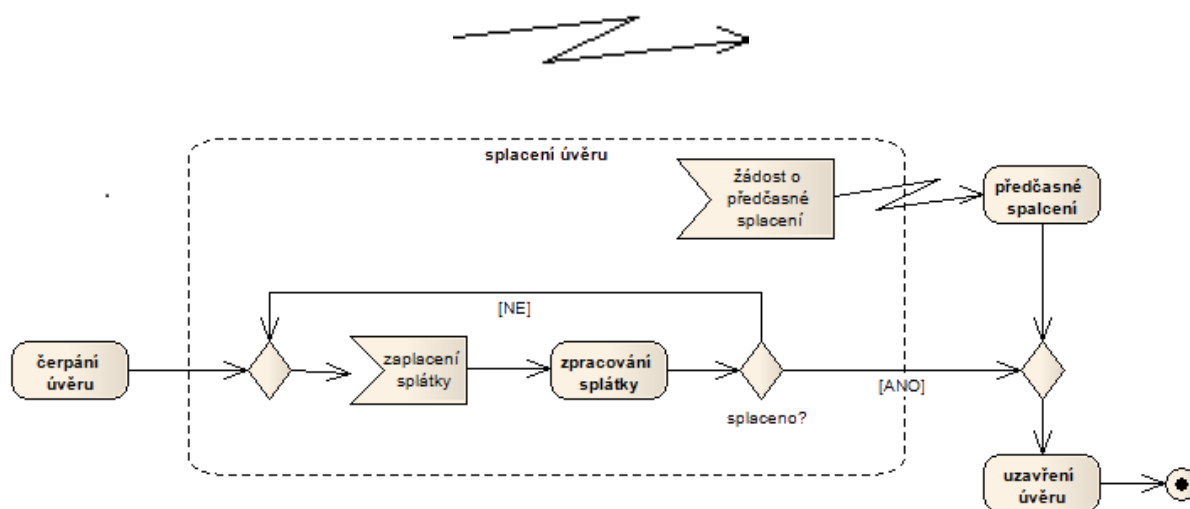
Danou část procesu ⁽¹⁾ lze ohraničit, pomocí regionu (*interruptible activity region*)



OBRÁZEK 1-24: INTERRUPTIBLE ACTIVITY (S,2018)

Událost pomocí přerušovaného toku (*interrupt flow*) spouští aktivitu, která zajišťuje zrušení prováděných akcí.

Původní tok v regionu zaniká.



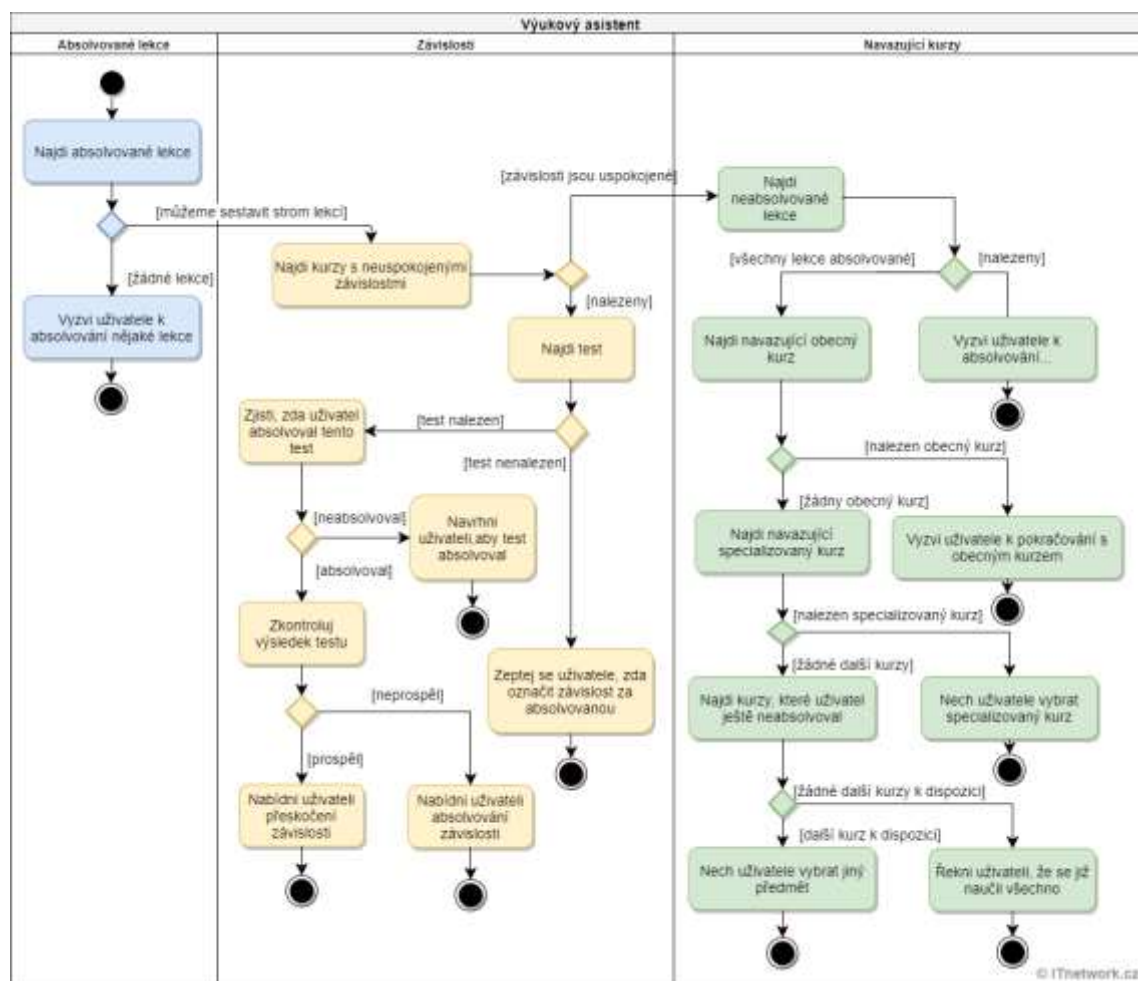
OBRÁZEK 1-25: CVIČENÍ - AKTIVITY DIAGRAM III

1.3.4 Doporučení

Dle UML 2 Activity Diagramming Guidelines (Scott W. Ambler, 2006) platí pro diagramy aktivit následující doporučení:

- Kontrolní podmínky rozhodovacích uzlů by se měly vždy vylučovat a měly by zahrnovat všechny možnosti, které mohou nastat. Nesprávné podmínky jsou například `x <= 9` a `x >= 9`, které se překrývají a není tedy možné jednoznačně určit, co se stane v případě, že `x = 9`.
- Počáteční uzly by měly být umístěny v levém horním rohu diagramu.
- Diagram aktivit by měl vždy obsahovat konečný uzel, pokud se nejedná o opakující se proces.
- Každý akční uzel by měl mít jak vstupní, tak výstupní hranu.
- Diagram by neměl obsahovat více jak pět plaveckých drah.
- Plavecké dráhy by měly být znázorněny vertikálně. Jejich horizontální orientace je nevhodná především při modelování byznys procesů.
- Pokud se v diagramu vyskytuje více identických objektových uzlů (objektů), je vhodné je odlišit vyznačením stavů (např. Objednávka [otevřená], Objednávka [zavřená]).

1.4 Příklad diagramu aktivit + vysvětlení (Čápka, 2019)



OBRÁZEK 1-26: PŘÍKLAD ACTIVITY DIAGRAMU PODLE (ČÁPKA, 2019)

1.4.1 Vysvětlení

První, čeho si všimneme, jsou pravděpodobně tzv. *swimlanes* (plavecké dráhy). Složitější UML diagramy můžeme takto rozdělit do několika svislých nebo vodorovných obdélníků. Barevné odlišení prvků diagramu je jen pro lepší orientaci a není součástí UML notace. Oddělujeme právě pomocí *swimlanes*. Každý obdélník typicky reprezentuje jiné vlákno, jinou část procesu, jiného uživatele apod.

Zde jsme diagram rozdělili do 3 částí:

- Absolvované lekce – V první dráze asistent zjišťuje, zda uživatel již absolvoval alespoň jednu lekci na síti, aby zjistil, co jej zajímá, a mohl mu pomáhat s dalšími kroky jeho studia.
- Závislosti – V prostřední dráze se kontroluje, zda mají všechny absolvované lekce daného uživatele uspokojené tzv. závislosti. To znamená, zda uživatel absolvoval předchozí kurzy, na které lekce navazují. Zkráceně zda mu nic nechybí.
- Navazující kurzy – Poslední dráha se věnuje výběru dalších lekcí a navazujících kurzů.

Absolvované lekce

První dráha je jednoduchá. Program se pokusí nalézt *nějaké* absolvované lekce uživatele. Pokud žádné nenalezne, vyzve jej, aby si na síti nejprve vybral nějaký kurz a absolvoval první lekci. Pokud nějaké lekce našel, je program schopen sestavit tzv. strom lekcí a inteligentně pomáhat dále ve výuce.

Závislosti

Jakmile má program lekce uživatele, namapuje si kurzy, do kterých tyto lekce patří, a podívá se, zda má uživatel absolvované předchozí kurzy. Předchozím kurzům, na které současné kurzy navazují, říkáme závislosti.

+ Pokud vše souhlasí, tok se přesouvá do další dráhy. Pokud je nalezen kurz s neabsolvovaným předchozím kurzem, program se podívá, zda má tento kurz test.

– Pokud ne, zeptá se uživatele, zda ovládá znalosti neabsolvovaného předchozího kurzu a chce jej přeskočit.

Pokud test existuje, podívá se program, zda jej uživatel absolvoval.

Pokud ne, sdělí uživateli, že mu chybí předchozí kurz a měl by se testem ujistit, že danou látku ovládá.

Pokud uživatel test absolvoval, podívá se program na výsledek. Pokud je dobrý, nabídne uživateli přeskočení kurzu. V opačném případě uživateli doporučí absolvování tohoto kurzu.

Navazující kurzy

Když se tok dostane až do dráhy „Navazující kurzy“, znamená to, že je vše v pořádku a má uživatele namotivovat k pokračování ve výuce. Prvně se podívá, zda jsou v některém z kurzů, které uživatel čte, neabsolvované lekce. Pokud ano, jednoduše doporučí, aby pokračoval některou z těchto lekcí. Pokud nejsou v současných kurzech uživatele žádné další lekce k pokračování, pokusí se nalézt navazující obecný kurz. Při úspěchu uživateli doporučí pokračovat právě tímto kurzem. Pokud má uživatel všechny související obecné kurzy absolvované, podívá se po navazujících speciálních kurzech. To jsou kurzy, které dávají uživateli specializaci na určitou technologii a obvykle si vybere jen několik málo těchto kurzů k jeho vybraným jazykům. Pokud je takový specializovaný kurz nalezen, je uživateli doporučen. V opačném případě se skript podívá, jaké kurzy uživatel na ITnetwork ještě neabsolvoval. Pokud nějaké takové kurzy existují, nabídne mu jejich seznam a nechá jej vybrat nový předmět. V opačném případě sdělí uživateli, že se naučit vše na síti. K tomuto poslednímu případu při rozsahu sítě asi nikdy nedojde, ale rozhodovací strom by měl být kompletní.

Všimněte si, že každý tok končí v bodě *ukončení activity*.

Ufff. Můžete se podívat, kolik textu zabralo popsání algoritmu. A nyní se podívejte na obrázek. Je dokázané, že lidský mozek vnímá vizuální nákresy velmi výrazně efektivněji, než text. Diagram je u takto složitého rozhodovacího algoritmu nezbytná pomůcka dobrého programátora. Jen těžko lze ladit kód, o kterém si nejste jistí, co má, v kterém bodě přesně udělat.

Cvičení:

. Sestavte (načrtněte) diagram aktivit na téma: „Jednoduchá platba v obchodě“

. Uvažte pouze možnosti platby hotovostí / kartou / stravenkami

. Využijte znalosti z kapitoly Tok aktivit

. Nepovinný domácí úkol

. Rozšiřte výše uvedený úkol o další části (např. EET)

2 Algoritmy

2.1 Základy teorie algoritmů

Algoritmus je přesný návod či postup, kterým lze vyřešit daný typ úlohy.

- Podrobný popis všech kroků algoritmu → Elementární = skládá se z konečného počtu jednoduchých a snadno srozumitelných kroků – příkazů)

Z lidského pohledu = např. kuchyňský recept;
 návod, na sestavení skříně;
 popis cesty;
 návod, jak ráno vstát z postele.

„Vstaň z postele!“

- Nejedná se o algoritmus – není splněné výše definovaná kritéria

Z pohledu IT / stroje = (neumí myslet)

„Otevři oči → sundej peřinu → posad' se → otoč se → dej nohy na zem → stoupni sť“

Problémy typu seřad' prvky podle velikosti *nebo* vyhledej prvek podle jeho obsahu = 2 základní úlohy – nejčastěji řešené IT technikou →

→ nutnost dokonale promýšlet, optimalizovat, tak aby trvaly co nejkratší dobu.

Další příklad algoritmů = vyřeš kvadratickou rovnici / vyřeš sudoku.

Příklad

Chceme, aby počítač prohodit dvě hodnoty proměnných (x , y).

```
x=y;  
y=x;
```

Tento kód zcela jistě fungovat nebude!

Vstup: $x = 4$ a $y = 10$.

Činnost: V prvním kroku se do proměnné y uloží 4. Druhý krok pak do proměnné x uloží hodnotu y , tedy 4.

Výstup: $x = 4$ a $y = 4$.

Zadání tedy není splněno.

Pomozme si tedy *pomocnou* novou proměnnou w :

```
w=x;  
x=y;  
y=w;
```

Vstup: $x = 4$ a $y = 10$

Činnost:

- V prvním řádku se do nově definované proměnné w se uloží hodnota proměnné x .

Stav po činnosti 1. řádku: $w = 4$ a $x = 4$ a $y = 10$.

- Na druhém řádku se do proměnné x uloží hodnota y .

Stav po činnosti 2. řádku: $w = 4$ a $x = 10$ a $y = 10$.

- Na třetím řádku se do proměnné y uloží hodnota proměnné w , tedy původní hodnoty x .

Stav po činnosti 3. řádku: $w = 4$ a $x = 10$ a $y = 4$.

Výstup: $x = 10$ a $y = 4$ Hodnota w není předmětem zkoumání.

Zadání splněno.



Máme tu tedy první algoritmus na prohození dvou čísel. Všimněte si, že nelze říci „*prohod' čísla*“. Musíme dokonale popsat co má program dělat pomocí soustavy příkazů.

Další důležitá vlastnost: determinovanost

- Po \forall kroku lze určit, zda algoritmus skončil, příp. jak dál jednoznačně pokračuje.
- Algoritmus by měl skončit správným výsledkem, a to v případě zadání jakéhokoliv vstupu.
- Měl by být schopen pokrýt všechny vstupy

Algoritmus musí být obecný:

```
■ 2+1;
```

Nejedná se o algoritmus \Leftrightarrow řešení jednoho konkrétního problému.

Algoritmus řeší celou škálu obdobných problémů a má širokou množinu různých vstupů:

```
■ x+y;
```

za předpokladu, že máme zdefinované proměnné x a y .

Algoritmus musí být ještě konečný. Nemůže si dovolit se někde zacyklit, vždy musí skončit s konečným počtem kroků.

2.1.1 Základní algoritmické konstrukce

Základní stavební kámen = instrukce, příkaz

- Popisuje elementární operaci, která se má vykonat (*např. přičti 2*)

Algoritmus se navrhuje několika možnými způsoby:

- Shora dolů – postup řešení rozkládáme na jednodušší operace, až dospějeme k elementárním krokům.
- Zdola nahoru – z elementárních kroků vytváříme prostředky, které nakonec umožní zvládnout požadovaný problém.
- Kombinace – obvyklý postup shora dolů doplníme „částečným krokem“ zdola nahoru tím, že se například použijí knihovny funkcí, vyšší programovací jazyk nebo systém pro vytváření programů (CASE).

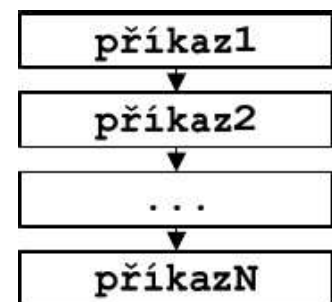
Dále se v algoritmu vyskytují záhadné výrazy, jako např. „jsi-li“, „pokud“, „dokud“, „opakuj“, ...

⇒ tím lze vytvořit z instrukcí složitější struktury:

2.1.1.1 Posloupnost příkazů

Jedná se o posloupnost jednoduchých i strukturovaných příkazů, které se postupně provedou.

```
Start
Prikaz1;
Prikaz2;
...
PrikazN
End
```

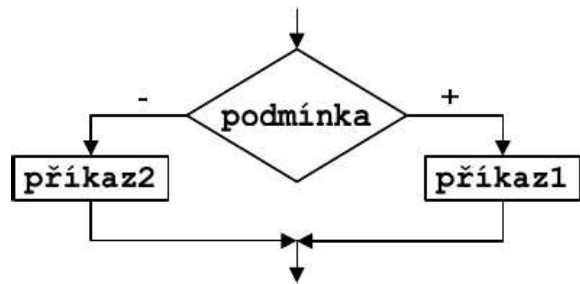


2.1.1.2 Větvení

Jedná se o algoritmický konstrukt, který na základě vyhodnocení určité podmínky dovoluje rozdělit program do dvou větví. Rozlišujeme úplné a neúplné větvení. Zatímco úplné obsahuje dvě disjunktivní větve, neúplné dělení neobsahuje větev pro výsledek FALSE.

Úplná podmínka

```
if (a>b){  
    Prikaz11;  
    Prikaz12  
}  
else {  
    Prikaz21;  
    Prikaz22;  
}
```



Neúplná podmínka

```
if (a>b){  
    Prikaz1;  
    Prikaz2;  
}
```

2.1.1.3 Cykly

Cyklus (též *smyčka*), angl. loop je algoritmický konstrukt, který opakovaně provádí danou posloupnost příkazů. Opakování i ukončení cyklu je regulováno danou podmínkou. Cyklus se skládá z posloupnosti příkazů a podmíněného skoku, pomocí kterého se cyklus ukončuje při splnění podmínky.

Nekonečný cyklus

- za normálních okolností takový cyklus není ukončen

while-do cyklus (cyklus s testem na začátku)

- Když podmínka není na počátku splněna, cyklus nemusí proběhnout ani jednou.
- Tento cyklus testuje podmínku před samotným provedením těla. Z toho vyplývá, že cyklus nemusí proběhnout ani jednou. Pokud není před prvním průchodem splněna podmínka, cyklus se jakoby přeskočí a pokračuje se

následujícím příkazem. Cyklus se používá v případě, pokud nevíme, kolikrát v programu proběhne. Ukončovací podmínka tedy závisí na nějaké akci (příkazu) uvnitř těla cyklu.

```
while (podmínka) {  
    tělo cyklu  
}
```

do-while cyklus (cyklus s testem na konci)

- Tento cyklus musí proběhnout aspoň jednou.
- Využití v případech, kdy víme, že cyklus musí být proveden přinejmenším jednou a zároveň nemusíme znát počet potřebných opakování
- Pokud program dospěje k cyklu, provede bez okolků všechny příkazy v jeho těle a potom narazí na podmínku. Pokud je daná podmínka splněna, program se vrací na začátek těla cyklu a ten se opakuje. V opačném případě je cyklus ukončen.

```
do {  
    tělo cyklu  
} while (podmínka);
```

cyklus s testem podmínky uprostřed posloupnosti příkazů

- řídce užívaná varianta.
- Hodně jazyků umožňuje násilné přerušení cyklu (vyskočení ven z cyklu; např. v C se používá příkaz `break`) => tedy je možné vytvořit například nekonečný cyklus, uvnitř tohoto cyklu testovat nějakou podmínku a ve vhodné situaci cyklus přerušit.

for cyklus (cyklus se známým počtem průchodů)

- speciální případ cyklu s podmínkou na začátku, obvykle užívaný pro výčet prvků z množiny prvků (např. interval celých čísel $\langle 1; 10 \rangle$). V některých jazycích je počet opakování vyhodnocen jednou na začátku a další změna této

podmínky nemá na počet opakování vliv. V ostatních jazycích je for cyklus de facto zvláštním případem while-do cyklu:

- Cyklus for se používá v případech, kdy předem víme, kolikrát proběhne.
- Velmi často se používá při práci s datovou strukturou - polem (array), kdy hodnota opakovacího výrazu určuje, se kterou hodnotou v poli se bude v tomto okamžiku pracovat.

```
for (počáteční výraz; podmínka; iterační výraz) {  
    tělo cyklu  
}
```

Konkrétní zobrazení příklad:

```
for (i = m; i < n; i++) {  
    printf("Ahoj");  
}
```

Cyklus proběhne n krát.

V obecném případě (pro i od m do n) proběhne $(n - m + 1)$ krát, pokud $m > n$ tak neproběhne ani jednou.

2.1.1.4 Složené algoritmické konstrukce

- Jednotlivé základní algoritmické konstrukce lze do sebe vnořovat.
- Místo jednotlivých příkazů v konstrukcích mohou být celé složené příkazy.

2.1.2 Vlastnosti algoritmů – shrnutí:

Podmínky, které algoritmus musí splňovat:

Konečnost

- Každý algoritmus musí skončit v konečném počtu kroků.

Obecnost (hromadnost, masovost, univerzálnost)

- Algoritmus neřeší jeden konkrétní problém, ale obecnou třídu obdobných problémů a má širokou množinu možných vstupů.

Determinovanost

- Každý krok algoritmu musí být jednoznačně a přesně definován; v každé situaci musí být naprosto zřejmé, co a jak se má provést, jak má provádění algoritmu pokračovat.

Výstup (resultativnost)

- Algoritmus má alespoň jeden výstup

Elementárnost

- Algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků.

2.1.3 Způsoby zápisu algoritmu

Poté, co algoritmus vymyslíme, je nezbytné jej zapsat tak, aby mu rozuměl ten kdo jej bude vykonávat (počítač, tým pracovníků, ...).

1. Přirozený jazyk
2. Strukturovaný jazyk
3. Grafické vyjádření
4. Programovací jazyk

2.1.3.1 Přirozený jazyk

S tímto zápisem algoritmu se setkáváme každodenně.

- + Rozumíme mu všichni (babylon)
- Není jednoznačný (spor s definicí algoritmu!) – jazyková úskalí

Př.: Oholím se a půjdu do divadla nebo půjdu na párty

První nejednoznačnost = spojka nebo (vylučovací nebo slučovací?)

- × Existence, resp. neexistence čárky před *nebo* → význam jednoznačně určuje

2.1.3.2 Strukturovaný jazyk

V podstatě přirozený jazyk, v němž platí určitá dohodnutá omezení

- Například smíme používat silně omezené množství slovních výrazů

2.1.3.3 Grafický zápis

Velmi názorný (nehodí se pro vyjádření složitých algoritmů) – vhodné rozdělit do dílčích částí.

Př.: UML diagramy ve způsobu vyjádření sketch a blueprint. (Ekvivalent)

2.1.3.4 Programovací jazyk

Jasně daná množina slov (příkazů) s přesně definovaným významem. U některých programovacích jazyků lze vytvářet nové příkazy.

Cvičení:

1. Je dána kvadratická rovnice tvaru $ax^2 + bx + c = 0$. Vytvoř algoritmus pro řešení a následně jej převed' do Aktivit' diagramu.
2. Vytvoř aktivit' diagram pro algoritmus smažení řízku:
Nápověda
 - a. Jsi-li vegetarián, vytáhni z ledničky sojové maso, jinak vytáhni plátek masa.
 - b. Naklepej a osol surovinu
 - c. Obal ji v mouce
 - d. Vykoupej ji ve vajíčku
 - e. Obal ji ve strouhance
 - f. Rozpal olej a vlož do něj obalenou surovinu
 - g. Smaž, dokud není dozlatova usmažená
3. Vytvoř algoritmus pro řešení BMI (index tělesné hmotnosti) a následně jej převed' do diagramu aktivit – POVINNÝ DŮ (malá známka)
nápověda pro výpočet: $BMI = \frac{\text{hmotnost [kg]}}{(\text{vyska [m]})^2}$
pozn.: nesplněno = nedostatečně

2.2 Třídící algoritmy

= zajišťuje uspořádání dané sady (pole, seznam, soubor) datových záznamů do zadaného pořadí.

V obecném pohledu lze třídící algoritmy rozdělit na:

- Elementární (např.: Selection sort, Bubble sort, Insertion sort)
- Pokročilé (např.: Heap-sort, Quick sort, Merge sort)

Existuje několik různých způsobů třídění dat, např.:

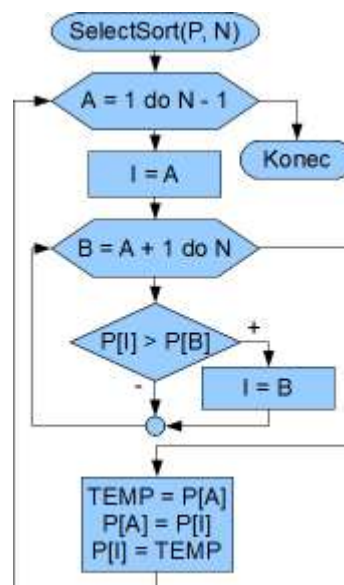
2.2.1 Selection sort (řazení výběrem)

- Jeden z nejjednodušších (ale pomalých) algoritmů
- Myšlenka = nalézá minimum, které přesune na počátek (analogicky maximum)
- První krok = nalezneme nejmenší prvek v datovém záznamu (poli) a ten přesuneme na začátek
- Druhý krok analogický, zanedbáme však 1. prvek (ten nejmenší, najitý v předchozím kroku)

Časová složitost	(n^2)
Stabilita	Ne
Rychlost	Pomalý

viz video na Moodlu

OBRÁZEK 2-1: VÝVOJOVÝ DIAGRAM PRO SELECTION SORT (BENEŠ, 2012) >>>



Příklad zdrojového kódu v Javě

```
// Řazení výběrem (od nejvyššího  
public static void selectionSort(int[] array) {  
    for (int i = 0; i < array.length - 1; i++) {  
        int maxIndex = i;  
        for (int j = i + 1; j < array.length; j++) {  
            if (array[j] > array[maxIndex]) maxIndex = j;  
        }  
        int tmp = array[i];
```

```

        array[i] = array[maxIndex];
        array[maxIndex] = tmp;
    }
}

```

ALGORITMUS 1: SELECTION SORT V JAVĚ (ALGORTIMY.NET, 2019)

2.2.2 Bubble-sort (bublínkové řazení)

- Poměrně hloupý algoritmus (výskyt jen v akademickém světě) =>

=> Didaktické důvody

- Žádné dobré vlastnosti ← zajímavý jen průběhem

- Probíhá ve vlnách

- Při každé vlně propadne „nejtěžší“ prvek na konec, resp.
 - „Nejtěžší“ prvek vybublá nahoru (podle implementace)

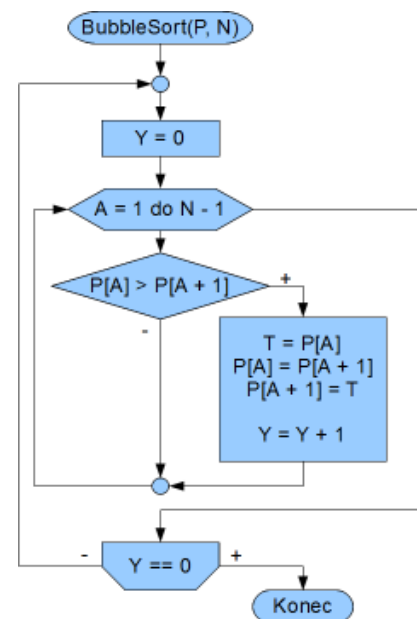
- Vlna porovnává dvojice sousedních prvků (např.

L (levý) a P (pravý)) =>

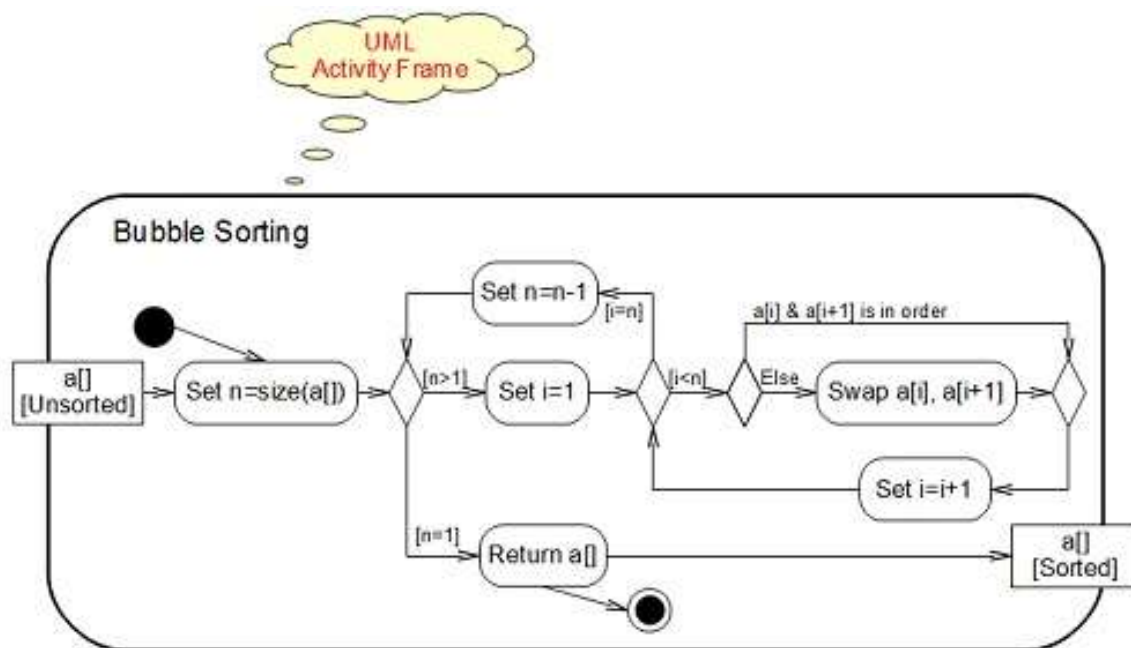
=> v případě, že $L > P$: prvky se prohodí

=> pokud platí $L \leq P$: prvky zůstávají na svých místech

- Stabilní



Časová složitost	$O(n^2)$
Stabilita	Ano
Rychlost	Extrémně pomalý



OBRÁZEK 2-2: ACTIVITY DIAGRAM PRO BUBBLE SORT (YANG, 2014)

Příklad zdrojového kódu v Javě

```

public static void bubbleSort(int[] array){
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = 0; j < array.length - i - 1; j++) {
            if(array[j] < array[j+1]){
                int tmp = array[j];
                array[j] = array[j+1];
                array[j+1] = tmp;
            }
        }
    }
}

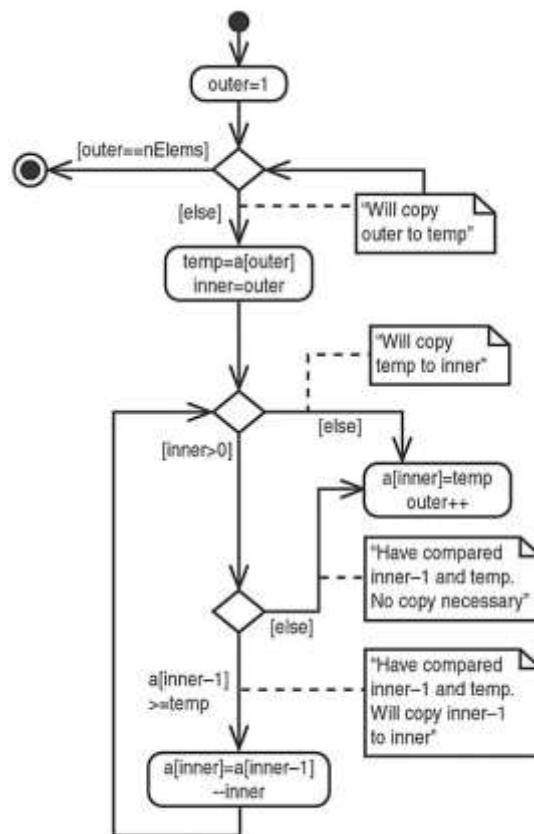
```

ALGORITHMUS 2: BULBBLE SORT (ALGORTIMY.NET, 2019)

2.2.3 Insertion sort (řazení vkládáním)

- ‚Králem‘ mezi elementárními řídicími algoritmy
- Stabilní, jednoduchý, inteligentní
- Insertion sort dělí datový záznam na dvě části:
 - (a) setříděnou
 - (b) nesetříděnou
- Prvky z nesetříděné (b) části postupně *vkládá* je mezi prvky v setříděné (a) části, tak aby zůstala setříděná — vkládá prvky přesně tam, kam patří
 - Nedělá zbytečné kroky jako Bubblesort

Časová složitost	$O(n^2)$
Stabilita	Ano
Rychlost	Vysoká na malých polích



OBRAZEK 2-3: ACTIVITY DIAGRAM PRO INSERTION SORT (LAFORE, 2003)

Příklad zdrojového kódu v Javě

```

public static void insertionSort(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int j = i + 1;
        int tmp = array[j];
        while (j > 0 && tmp > array[j-1]) {
            array[j] = array[j-1];
            j--;
        }
        array[j] = tmp;
    }
}

```

ALGORITHMUS 3: INSERTION SORT (ALGORTIMY.NET, 2019)

2.2.4 Heapsort (řazení haldou)

- Patří mezi chytré rychlejší algoritmy
- Základy na ideách Selection sortu → odtrhává maximum a přesouvá na konec datového záznamu
 - V každém vnějším cyklu se musí projet celý nesetříděný datový záznam a zároveň kontrolovat, zda prvek náhodou není nové maximum

→ nevýhoda

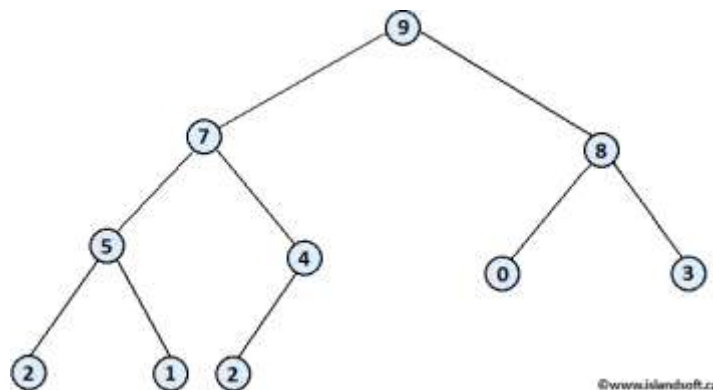
(!) Existence datové struktury, kde můžeme prvky reprezentovat stejně, jako v poli, a zároveň nalézáme maximum v konstantním čase, bez jediného průchodu

→ Halda (ang. Heap)

Halda je binárním stromem s následujícími vlastnostmi:

- (a) Všechna „patra“ haldy až na poslední jsou plně obsazeny prvky (tedy každý vnitřní vrchol má právě 2 syny → (strom je velmi vyvážený)
- (b) Poslední patro haldy je zaplněno zleva (může být i zaplněno celé)
- (c) Pro prvky v haldě platí tzv. speciální vlastnost haldy: oba synové jsou vždy menší nebo rovny otci.

Ze speciální vlastnosti haldy nutně vyplývá, že v kořeni bude vždy uloženo maximum, což se nám velmi hodí.



Příklad haldy

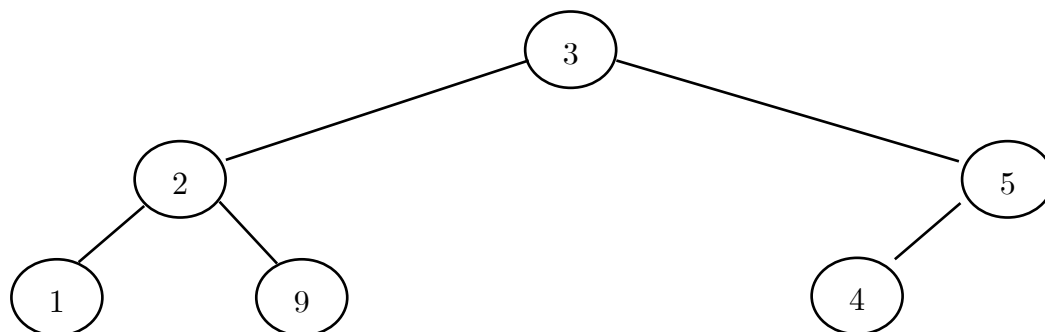
Průběh, praktická ukázka:

- Detailní průběh ve formátu GIF je k dispozici ve studijním prostředí Moodle.

Nechť je dáno pole:

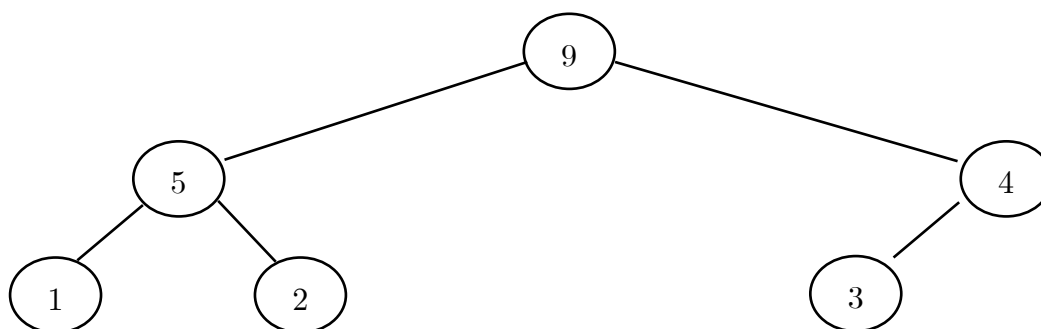
3	2	5	1	9	4
---	---	---	---	---	---

Toto pole reprezentuje následující haldu (zatím rozbitou):



Halda splňuje vlastnosti (a) a (b). Speciální vlastnost není splněna. Haldu tedy opravíme (zhladujeme). \Leftrightarrow použijeme funkci up (nahoru) \rightarrow postupně voláme na prvky od kořene dolů:

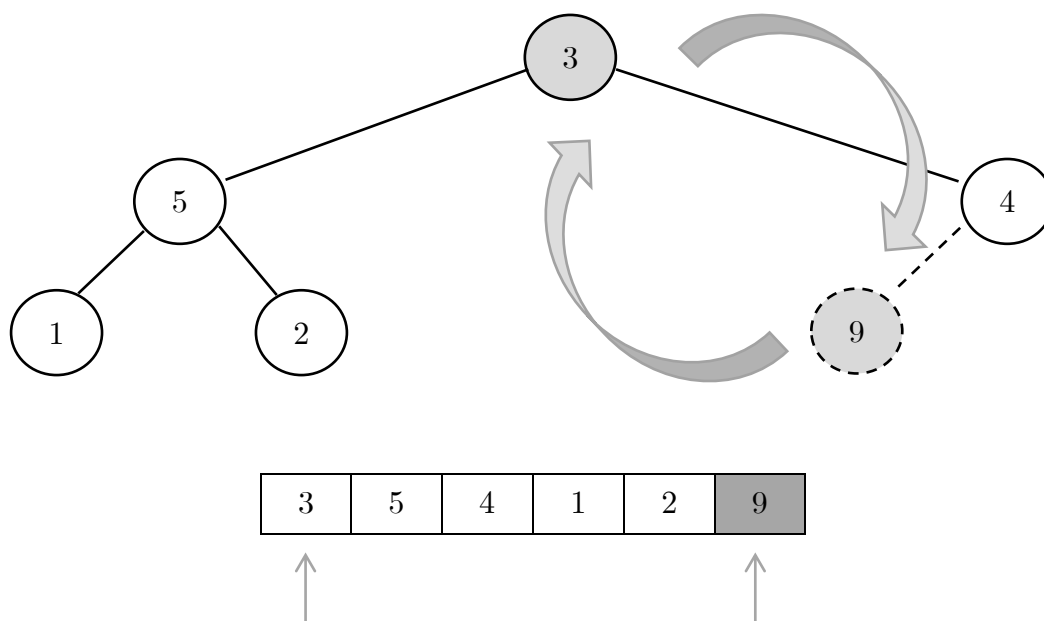
- Funkce up porovná hodnoty otce (O) a syna (S):
 - Pro $S > O$: hodnoty prohodí a pokračuje dál (může se problém přenést o úroveň výše) \Rightarrow funkci opakujeme pro každý prvek dokud speciální vlastnost neplatí pro celou haldu



Nyní již halda splňuje všechny vlastnosti. Pole v tomto kroku vypadá následovně:

9	5	4	1	2	3
---	---	---	---	---	---

Nyní budeme provádět Selection sort v haldě (tedy prohazovat maximum s posledním prvkem, ...). Maximum – vždy na indexu 0 (v kořeni haldy) → prohodíme



(!) přehozením prvků si rozbijeme haldu

Původního maxima (9) si již sice všimnout nebudeme (→ setříděná část pole, které si stejně jako v Selection sortu nevšímáme), ale prvek, který je nyní novým kořenem (3), dost pravděpodobně nebude maximum nové haldy.

Nyní využijeme analogickou funkci k up, tentokrát down (dolů). Tu pustíme na nový kořen:

- Pro $S > 0$: hodnoty prohodí a pokračuje dál (může se problém přenést o úroveň výše) \Rightarrow funkci opakujeme pro každý prvek dokud speciální vlastnost neplatí pro celou haldu

Opakujeme, dokud halda nesplňuje speciální vlastnost.

<i>Časová složitost</i>	$O(n \cdot \log(n))$
<i>Stabilita</i>	Ne
<i>Rychlost</i>	Velmi dobrá

Příklad zdrojového kódu v Javě

```
/**
 * Heapsort - razeni haldou
 * @param array pole k serazeni
 * @param descending true, pokud ma byt pole serazeno sestupne, false pokud
 * vzestupne
 */
public static void heapSort(Comparable[] array, boolean descending) {
    for (int i = array.length / 2 - 1; i >= 0; i--) {
        repairTop(array, array.length - 1, i, descending ? 1 : -1);
    }
    for (int i = array.length - 1; i > 0; i--) {
        swap(array, 0, i);
        repairTop(array, i - 1, 0, descending ? 1 : -1);
    }
}

/**
 * Umisti vrchol haldu na korektni misto v halde (opravi haldu)
 * @param array pole k setrizeni
 * @param bottom posledni index pole, na který se jeste smi sahnout
 * @param topIndex index vršku haldy
 * @param order smer razeni 1 == sestupne, -1 == vzestupne
 */
private static void repairTop(Comparable[] array, int bottom, int topIndex,
int order) {
    Comparable tmp = array[topIndex];
    int succ = topIndex * 2 + 1;
    if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order) {
        succ++;
    }

    while (succ <= bottom && tmp.compareTo(array[succ]) == order) {
        array[topIndex] = array[succ];
        topIndex = succ;
        succ = succ * 2 + 1;
        if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order)
        {
            succ++;
        }
    }
    array[topIndex] = tmp;
}

/**
 * Prohodi prvky haldy
 * @param array pole
 * @param left index prvnioho prvku
 * @param right index druhého prvku
 */
private static void swap(Comparable[] array, int left, int right) {
    Comparable tmp = array[right];
    array[right] = array[left];
    array[left] = tmp;
}
```

ALGORITMUS 4: HEAPSORT V JAVĚ (ALGORTIMY.NET, 2019)

. Cvičení:

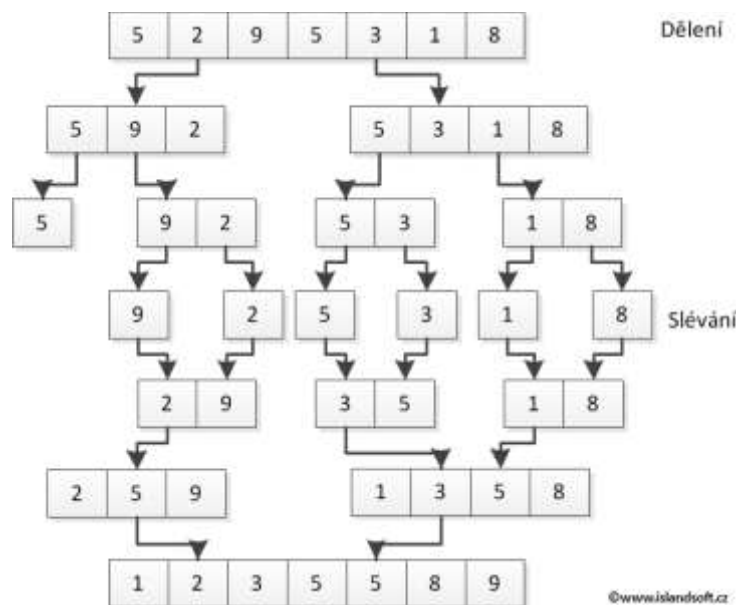
. Vygenerujte si 5–10 náhodných přirozených čísel a tato čísla si vytiskněte. Zkuste si
/ tato čísla seřadit pomocí heapsort. Náповědou, nechť jsou videa na Moodlu.

Zaznamenejte svůj postup do Diagramu aktivit.

2.2.5 Merge sort (řazení slučováním / sléváním)

- Založený na principu rozděl a panuj (divide et impera) \Rightarrow
 \Rightarrow problém nelze triviálně řešit v celku (rozložíme si ho na více menších a jednodušších problémů)
- Totéž lze aplikovat i na menší problémy až se dostaneme na úroveň, kterou lze řešit triviálně

Merge sort operuje s myšlenkou, že dokážeme velmi rychle a v lineárním čase slít (spojit, anglicky merge) dvě již setříděná pole do jednoho tak, aby výsledné pole bylo opět setříděné.

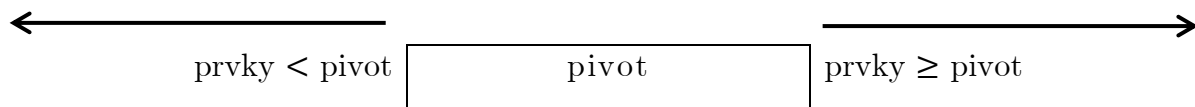


2.2.6 Quicksort (rychlé řazení)

- Rychlý, resp. nejrychlejší algoritmus
- Skutečně se používá v praxi
- Pro malá i velká pole (nenáročný na paměť)
- Založen na principu rozděl a panuj

Quicksort určí jeden prvek v poli jako tzv. pivot (výběr prvku prozatím zanedbatelný – *prozatím* pivot = vždy první prvek v poli)

Funkce divide (rozděl) přeuspořádá (! nikoliv setřídí !) pole tak, aby:



Prvky jsou vzájemně neuspořádané \Leftrightarrow jediné setřídění spočívá v jejich rozdělení pivotem (rychlá operace)

Nyní funkci rekurzivně zavoláme na levou a pravou polovinu (pivota necháme tam kde je) a opakujeme, dokud nemáme jednotkové pole (*pole velikosti 1*)

Výběr pivota je z hlediska efektivity průběhu klíčový. Vybereme-li pivota jako první prvek, příp. náhodně, může se stát, že rozdělení nebo nerovnoměrné.

První příklad

Uvažujme datovou strukturu


3	4	8	2	13	10	5
---	---	---	---	----	----	---

Definujme první prvek jako pivot.

3	4	8	2	13	10	5
---	---	---	---	----	----	---

Nyní použijeme funkci divide. Funkce uloží pivot na konec pole, aby nepřekážel (prohodí pivota s posledním prvkem).

5	4	8	2	13	10	3
---	---	---	---	----	----	---



Postupně funkce projde pole a přeuspořádá prvky:

2	3	5	4	8	13	10
---	---	---	---	---	----	----

Identický postup se aplikuje na prvky nalevo od pivota a následně napravo od pivota, tedy na pole. Jak je vidět levá strana od pivota obsahuje jediný prvek.

Pravá strana obsahuje prvků pět.

Pokud budeme aplikovat tentýž algoritmus na jednotlivé strany (třeba pomocí rekurze), je triviálně zřejmé, že průběh bude časově nevhodný, než pokud bychom vybrali např. hodnotu \bar{x} . Proto může být výhodné zvolit medián \bar{x} dané datové struktury. Medián je prostřední hodnota definována jako

$$\bar{x} = x_{\left(\frac{N+1}{2}\right)}$$

kde X je daný datový soubor hodnot, x_i je i -tý prvek souboru X a N je velikost souboru.

Pro volbu mediánu je však nezbytně nutné mít seřazený datový soubor, což je však cíl úlohy. Pokud se nám podaří zvolit číslo blízké mediánu, tak je daný algoritmus Quicksort skutečně velmi rychlý až $\mathcal{O}(n \log n)$. V extrémních případech se může složitost dostat až na $\mathcal{O}(n^2)$. Proto existuje velké množství alternativních způsobů, které se snaží efektivně vybrat pivot co nejbližší mediánu. Zajímavou metodou může být Metoda mediánu tří (pěti / sedmi / ...) prvků. Pomocí pseudonáhodného algoritmu (také se používají fixní pozice, typicky první, prostřední a poslední) se vybere několik prvků z množiny, ze kterých se vybere medián, a ten je použit jako pivot.

Druhý příklad

Nechť je dána datová struktura


5	2	9	3	5	1	8
---	---	---	---	---	---	---

Definujeme si jeden prvek jako pivot, např. první prvek (5)

5	2	9	3	5	1	8
---	---	---	---	---	---	---

Nyní použijeme funkci divide. Funkce uloží pivot na konec pole, aby nepřekážel (prohodí pivota s posledním prvkem).

8	2	9	3	5	1	5
---	---	---	---	---	---	---



Postupně funkce projde pole a přeuspořádá prvky:

2	3	1	5	5	9	8
---	---	---	---	---	---	---

Identický postup se aplikuje na prvky nalevo od pivotu a následně napravo od pivotu, tedy na pole

2	3	1
---	---	---

a

5	9	8
---	---	---

Příklad zdrojového kódu v Javě

```
/**
 * Razeni slevanim (od nejvyssiho)
 * @param array pole k serazeni
 * @param aux pomocne pole stejne delky jako array
 * @param left prvni index na který se smí sahnout
 * @param right poslední index, na který se smí sahnout
 */
public static void mergeSort(int[] array, int[] aux, int left, int right) {
    if(left == right) return;
    int middleIndex = (left + right)/2;
    mergeSort(array, aux, left, middleIndex);
    mergeSort(array, aux, middleIndex + 1, right);
    merge(array, aux, left, right);

    for (int i = left; i <= right; i++) {
        array[i] = aux[i];
    }
}

/**
 * Slevani pro Merge sort
 * @param array pole k serazeni
 * @param aux pomocne pole (stejne velikosti jako razene)
 * @param left prvni index, na který smím sahnout
 * @param right poslední index, na který smím sahnout
 */
private static void merge(int[] array, int[] aux, int left, int right) {
    int middleIndex = (left + right)/2;
    int leftIndex = left;
    int rightIndex = middleIndex + 1;
    int auxIndex = left;
    while(leftIndex <= middleIndex && rightIndex <= right) {
        if (array[leftIndex] >= array[rightIndex]) {
            aux[auxIndex] = array[leftIndex++];
        }
        else {
            aux[auxIndex] = array[rightIndex++];
        }
        auxIndex++;
    }
    while (leftIndex <= middleIndex) {
        aux[auxIndex] = array[leftIndex++];
        auxIndex++;
    }
}
```

```

while (rightIndex <= right) {
    aux[auxIndex] = array[rightIndex++];
    auxIndex++;
}
}

```

ALGORITMUS 5: QUICKSORT V JAVĚ (ALGORTIMY.NET, 2019)

2.2.7 Další třídící metody

Shell sort (Shellovo řazení, též řazení se snižujícím se přírůstkem)

- Podobný insertion sortu
- 1959, Donald L. Shell (* 1.3.1924 — † 2.11.2015)
- Nestabilní (tj. nezachovává původní pořadí dvou prvků se stejných klíčem)

Comb sort (hřebenové řazení)

Introspektivní řazení

Bucket sort (příhrádkové řazení)

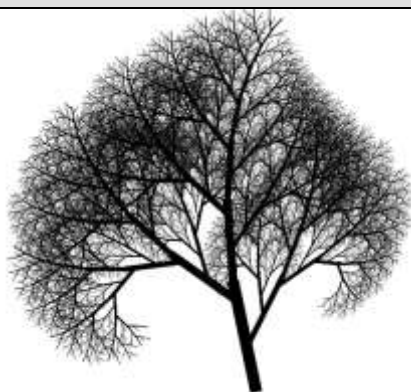
Radix sort (číslicové řazení)

a mnohé jiné další...

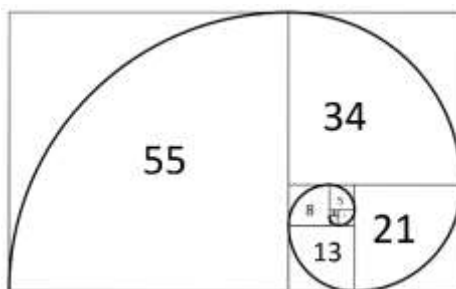
2.3 Rekurze

Programovací technika – určitá procedura / funkce znovu volána dříve, než je dokončeno její předchozí volání

Metoda, která volá sama na sebe se nazývá rekurzivní. Takže, když metoda volá sama na sebe, tomu se říká rekurze. Klíčovou složkou rekurzivní metody je příkaz, který provádí volání na sebe sama.

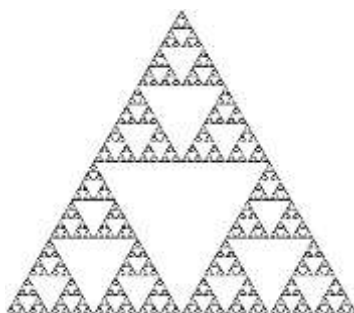


OBRÁZEK 2-4: REKURZIVNÍ KRESLENÍ (LESSNER, 2014)



OBRÁZEK 2-5: REKURZIVNÍ SPIRÁLA (SCHMIDT-CORNELIUS, 2014)

,



OBRÁZEK 2-6: REKURZIVNÍ TROJÚHELNÍK (PIANDWHIPPEDCREAM, 2007)



OBRÁZEK 2-7: REKURZE ANEB GOOGLE VTÍPKY

- Použití rekurze
 - může vést ke stručnému a matematicky elegantními řešení
(ne vždy však nutné optimální!)
- Starší programovací jazyky a překladače rekurzi neumožňují, nebo aspoň vyžadují, aby programátor explicitně uvedl, že daná procedura / funkce je rekurzivní

Rekurzivní chování – různé v podmíněnosti na tom kolik podprogramů se jí účastní.

- Usnadňuje popis mnoha problémů (snadná definice složitých fraktálových struktur /vyšší matematika/) a mnoha informatických úloh (stromové struktury, viz kap. 7.8 Stromy a lesy, str. 158)
- Rekurzivní algoritmy tvoří klíčový význam pro další algoritmické části učebnice. Bez dobrého porozumění samotné povahy rekurze nelze pochopit mnoho později prezentovaných algoritmů a programovacích metod

2.3.1 Definice rekurze

Rekurze se často klade do protikladu k iterativnímu přístupu čili n -násobnému provádění algoritmů takovým způsobem, aby výsledky získané v předchozích iteracích (průbězích) mohly posloužit jako vstupní data pro následující iterace.

- Iterace = řízeny instrukcemi cyklu (`for`, příp. `while`)
- Rekurze funguje podobně – místo cyklu je založena na tom, že stejná procedura (funkce) opakovaně volá sebe samu s jinými parametry
- V těle rekurzivní procedury můžeme též nalézt klasický cyklus (jen pomocná role)
- Programy zapsané v rekurzivním algoritmy lze převést na klasickou iterativní podobu (někdy ne zcela triviálně)

Hloubka rekurze je počet rekurzivních volání dané funkce.

Ukončení programu by mělo být jasně definováno buď:

- Nalezením hledaného prvku
- Překročením rozsahu pole – opuštěním prohledávané oblasti (pro případ, že $x \notin \text{tab}[n]$).
- Ukončovací podmínkou

V určitém místě musí dojít k takovému dořešení problému, které již nebude rekurzivní.

Velký problém jsme rozložili na elementární problémy, které umíme řešit, a na analogický problém, pouze s nižším stupněm složitosti. Z pole o velikosti n přecházíme na pole o velikosti $n - 1$.

Na základě uvedených postřehů se pokusme určit, jakých dvou základních chyb se programátoři dopouštějí při tvorbě rekurzivních programů

- Špatné vymezení koncové podmínky programu
- Neefektivní dekompozice problému

Rozlišujeme dva základní typy (1,2) dělení:

(1)

- a. Přímá rekurze = program volá sebe samu
- b. Nepřímá (kruhová) rekurze = vzájemné volání podprogramů vytvoří cyklus
funkce A volá funkci B \wedge funkce B volá funkci A

(2)

- a. Lineární rekurze = podprogram volá sama sebe právě jednou
- b. Stromová rekurze = podprogram volá sama sebe vícekrát

2.3.2 Rekurze a cyklus

Cyklus – opakování bloku příkazů za stanovených podmínek

- Podmínka opakování bloku předchází (`for`, `while`)
- Podmínka opakování blok následuje (`do while`)
- Podmínka součástí bloku, tedy uvnitř cyklus

Rekurze

- Opakování bloku, podmínka je součástí bloku
- Nejčastěji využívána u procedur a funkcí (nikoliv samostatně)

Mezi rekurzí a iterací je úzký vztah. Zpravidla platí vzájemná převoditelnost. Rekurze mnohdy bývají efektivnější než iterace. Toto pravidla však neplatí obecně. Pokud je to možné, tak se rekurzi snažíme vyhýbat.

2.3.3 Příklady použití

2.3.3.1 Faktoriál

Odbočka do matematiky – Co je faktoriál?

Faktoriál čísla ***n*** (značeno pomocí vykřičníku: ***n*!**) číslo, rovné součinu všech kladných celých čísel menších nebo rovných ***n***, pro ***n* > 0** a 1 pokud ***n* = 0**. Značení ***n*!** vyslovujeme jako „*n* faktoriál“.

Formální definice:

Předpokládejme

$$\forall n \in \mathbb{N}_0,$$

pak:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 = \prod_{k=1}^n k \text{ pro } n \geq 0$$

$$\text{Např.: } 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$$

Speciální příklad prázdného součinu, platí: $0! = 1$.

Pro $n \geq 1$, lze faktoriál definovat rekurzivně:

$$n! = n \cdot (n-1)! \text{ pro } n \in \mathbb{N}_1$$

Rekurzivní definice – často využívána v informatice (\Rightarrow jednoduchý zápis algoritmu)

- Tento výpočet – zcela nevýhodný (náročný na systémové prostředky (velikost zásobníku))

Příklad zdrojového kódu v Javě

```
/**
 * Rekurzivní výpočet faktoriálu čísla
 */
public static int factorialRek(int number){
    if(number < 0) return -1;
    if(number == 0 || number == 1) return 1;
    return number * factorialRek(number - 1);
}
```

ALGORITMUS 6: REKURZIVNÍ VÝPOČET FAKTORIÁLU ČÍSLA (ALGORTIMY.NET, 2019)

Okno do matematiky

Řešte rovnice v \mathbb{N} .

1. $\frac{n!}{(n-2)!} + \frac{(n-1)!}{(n-3)!} = 8$, pro $\forall n \in \mathbb{N}_3$
2. $\frac{(n-3)! + (n-1)!}{(n-2)!} = 3$, pro $\forall n \in \mathbb{N}_2$
3. $\ln[(x+1)!] - \ln[(x)!] = 1$, pro $\forall n \in \mathbb{N}_0$

Které číslo je větší? $a = 50! + 53!$ nebo $b = 51! + 52!$?

Kombinační číslo $\binom{n}{k}$ a říkáme n nad k je definováno jako:

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & \text{pro } n \geq k \geq 0; \\ 0 & \text{jinak.} \end{cases}$$

Vypočítejte:

1. $\binom{6}{3} =$
2. $\binom{5}{4} =$

***Řešte soustavu rovnic

1. $\binom{x}{2} = 153$
 $\binom{x}{y} = \binom{x}{y+2}$

2.3.3.2 Výpočet x^n

Potřebné znalosti:

- vzorce pro práci s mocninami: $x^a \cdot x^b = x^{a+b}$
- dále je dobré si uvědomit: $n = n + 1 - 1 = 1 + n - 1$ 😊

Výpočet mocniny x^n pro $n \in \mathbb{N}_0$ lze řešit rekurzivně.

$$x^0 = 1, x^n = x \cdot x^{n-1}$$

Jde o stejně neefektivní rekurzi jako v případě faktoriálu. Mocninu lze navíc spočítat i pomocí logaritmů:

$$x^n = e^{n \cdot \ln x}$$

2.3.3.3 Fibonacciho posloupnost

Fibonacciho posloupnost je posloupnost $n \in \mathbb{N}_0$, pro která platí:

- každé číslo je součtem dvou čísel předchozích.

Prvními členy jsou dvě jedničky (někdy se uvádějí také čísla 0 a 1). Nekonečná řada členů pak začíná takto: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Formální definice:

$$F(n) = \begin{cases} 0, & \text{pro } n = 0; \\ 1, & \text{pro } n = 1; \\ F(n-1) + F(n-2) & \text{jinak.} \end{cases}$$

Příklad zdrojového kódu v Javě

```
/**
 * Fibonacciho posloupnost rekurzivne
 */
public static int fibonacciRek(int index){
    if(index == 0) return 0; //zarazka
    else if(index == 1) return 1; //druha zarazka
    else return fibonacciRek(index - 1) + fibonacciRek(index - 2); //rekur-
    zivni volani
}
```

ALGORITMUS 7: REKURZIVNÍ VÝPIS FIBONACCIHO POSLOUPNOSTI (ALGORTIMY.NET, 2019)

Výpočet rekurzí – neefektivní.

2.3.3.4 Třídící algoritmy

Např. Stromové algoritmy či Quicksort (viz kap. 2.2.6 na str. 47)

2.3.3.5 Hanojské věže

Jeden z nejznámějších úkolů, které zahrnutí použití rekurze

Zadání:

Při stvoření světa bylo na jednu ze tří diamantových jehel Velkého chrámu v Benaresu umístěno 64 disků. Od té doby kněží tyto disky přerovnávají na cílovou jehlu, přičemž v daném okamžiku smějí přenést jen jeden disk, nesmí být umístěn větší disk na menším a disky lze přenášet pouze s pomocí jehel. Až se kněží podaří disky přenést, chrám se zboří a svět s třeskem zmizí.

Máme k dispozici tři jehly (zdroj, cíl a pomocnou) a určitý počet disků s otvorem uprostřed. Na začátku jsou všechny disky navlečeny na zdrojové jehle a tvoří věž, která se postupně zužuje. Disky se mají s pomocí třetí jehly přenést na cílovou jehlu. Přitom musí být dodržena zmíněná pravidla, která úlohu poněkud omezují a řešení (přenášení disků) tak chvíli trvá...

Rekurzivní postup je zde velmi názorný a jednoduchý a skládá se ze tří kroků:

1. přenesení n -tého disku ze zdrojové jehly na pomocnou,
2. přenesení jednoho disku ze zdrojové jehly na cílovou,
3. přenesení n -tého disku z pomocné jehly na cílovou.

Poznámka: Úloha též poslouží jako modelová ukázka Stavových prostorů v kap. 8.3
Stavový prostor na str. 188

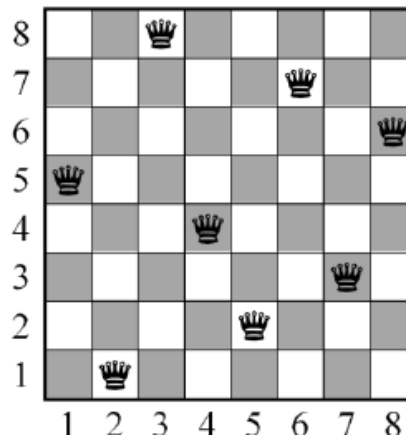
2.3.3.6 Problém osmi dam (Eight queens puzzle)

- Šachový, resp. kombinatorický problém umístit na šachovnici (8×8) osm dam tak, aby se podle pravidel šachu vzájemně neohrožovali

→ vybrat osm polí tak, aby žádná dvě nebyla ve stejné řadě, sloupci, ani diagonální linii

Často se zobecňuje na problém n dam – rozmístění na šachovnici o rozměrech $n \times n$.

OBRÁZEK 2-8: EIGHT QUEEN PUZZLE (MASEHIAN, 2013) >>>>



Příklad zdrojového kódu v Javě

```

/*****
 * Compilation:  javac Queens.java
 * Execution:    java Queens n
 *
 * Solve the 8 queens problem using recursion and
 * backtracing.
 * Prints out all solutions.
 *
 * Limitations: works for n <= 25, but slows down considerably
 * for larger n.
 *
 * Remark: this program implicitly enumerates all n^n possible
 * placements (instead of n!), but the backtracing prunes off
 * most of them, so it's not necessarily worth the extra
 * complication of enumerating only permutations.
 *
 * % java Queens 3
 *
 * % java Queens 4
 * Q * *
 * * * * Q
 * Q * * *
 * * * Q *
 *
 * * * Q *
 * Q * * *
 * * * * Q
 * * Q * *
 *
 * % java Queens 8
 * Q * * * * * *
 * * * * Q * *
 * * * * * * Q
 * * * * * Q *
 * * * Q * * *
 * * * * * * Q
 * * Q * * * *
 * * * Q * * *
 *
 * ...
 */

public class Queens {

/*****
 * Return true if queen placement q[n] does not conflict with

```

```

        * other queens q[0] through q[n-1]
    *****/
    public static boolean isConsistent(int[] q, int n) {
        for (int i = 0; i < n; i++) {
            if (q[i] == q[n])          return false;    // same column
            if ((q[i] - q[n]) == (n - i)) return false; // same major diagonal
            if ((q[n] - q[i]) == (n - i)) return false; // same minor diagonal
        }
        return true;
    }

    /**
     * Prints n-by-n placement of queens from permutation q in ASCII.
     */
    *****/
    public static void printQueens(int[] q) {
        int n = q.length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (q[i] == j) StdOut.print("Q ");
                else           StdOut.print("* ");
            }
            StdOut.println();
        }
        StdOut.println();
    }

    /**
     * Try all permutations using backtracking
     */
    *****/
    public static void enumerate(int n) {
        int[] a = new int[n];
        enumerate(a, 0);
    }

    public static void enumerate(int[] q, int k) {
        int n = q.length;
        if (k == n) printQueens(q);
        else {
            for (int i = 0; i < n; i++) {
                q[k] = i;
                if (isConsistent(q, k)) enumerate(q, k+1);
            }
        }
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        enumerate(n);
    }
}

```

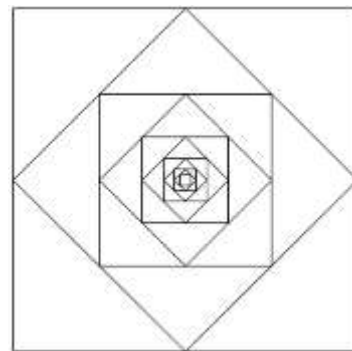
ALGORITHMUS 8: PROBLÉM OSMI DAM NA ŠACHOVNICI V JAVĚ (SEDGEWICK, 2007)

2.3.3.7 Kresba vepsaných čtverců

Nechť máme čtverec o straně d . Nakreslete čtverec, který do daného čtverce bude vepsán. Postup opakujte.

Lze problém řešit prostřednictvím rekurze?

- Úloha je dekomponovatelná na úlohy stejné třídy: kresba čtverců daných vrcholy.
- V dalším kroku rekurze nedochází k navýšení složitosti problému.
- Lze triviálně stanovit podmínku ukončení rekurze: velikost strany menší než ε .

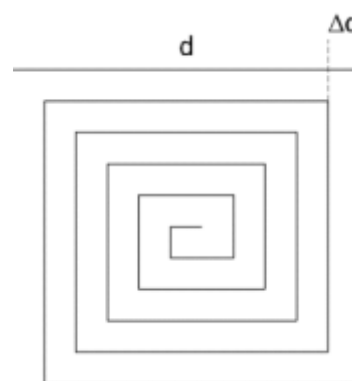


2.3.3.8 Kresba spirál

Zkonstruujeme spirálu, máme-li délku spirály d a zkrácení spirály Δd .

Lze problém řešit prostřednictvím rekurze?

- Úloha je dekomponovatelná na úlohy stejné třídy: opakovaná konstrukce jednoho závitu spirály tvořeného 4 úsečkami.
- V dalším kroku rekurze nedochází k zesložitění problému.
- Podmínka ukončení rekurze: $d - \Delta d < \varepsilon$.



Možné řešení problému: Zkonstruujeme jeden zavit spirály tvořený 4 segmenty. Délky prvního a druhého segmentu d , délka třetího a čtvrtého segmentu $d - \Delta d$. Stejným způsobem rekurzí zkonstruujeme další závity.

<https://web.natur.cuni.cz/~bayertom/images/courses/Prog1/programovani5.pdf>

3 Class diagram, object diagram

3.1 Class diagram

= Diagram tříd

- Třída je jakýsi prototyp objektů. Za třídou si můžeme představit množinu jejích instancí.
- Každý objekt dané třídy má stejnou množinu atributů (proměnných) a operací (metod).
- tzv. abstraktní třída nedefinuje implementaci (algoritmus) jedné nebo více metod. Je-li třída abstraktní, není možno vytvořit její instanci.
- Strukturální UML diagram, znázorňující datové struktury, operace u ,objektů', souvislosti mezi ,objekty'
- Znázorňuje datový model od úrovně konceptu až po implementaci
sketch → blueprint → programovací jazyk
- Datové struktury řadí do tříd a zobrazuje vztahy těchto tříd
- Základní stavební prvek objektové orientovaného modelování

Použití:

- Běžné koncepční modelování
- Detailní modelování
- Převod modelů do programového kódu
- Modelování dat

3.1.1 Význam diagramu

- Návod pro programátora
- Zkušený programátoři a analytici – navrhnou systém (sestaví class diagram)
- Méně zkušený programátor by neměl řešit žádné zásadní otázky → na základě class diagramu by měl sestavit program (rutina) ⇔ levnější
- Donutí přemýšlet v kontextu celého systému (předcházíme tím situacím, kdy *napišeme půlku systému a pak zjistíme, že to takhle fungovat nebude*)

- Složitější informační systému nelze vytvářet bez návrhu
- V budoucnosti může sloužit jako dokumentace

3.1.2 Části diagramu

Class diagram obsahuje třídy (class) a vztahy mezi nimi a příp. mohutnosti.

Každá třída je pak znázorněna obdélníkem, rozděleným na tři části: název třídy, atributy a metody.

3.1.2.1 Třídy

- Základní objekt class diagramu
- Znázorněny boxy → 3 základní části
 1. Název celé třídy, entita (tučné písmo),
pokud víceslovný → CamelCase¹
 2. Atributy třídy
začínají malým písmenem, pokud víceslovný → CamelCase
 - De facto – vlastnosti třídy
 3. Metody a operace třídy
začínají malým písmenem, pokud víceslovný → CamelCase
na konci označeny závorkami
- Mezi název atributu, resp. metody a datový typ zapisujeme dvojtečku
- Většinou se užívá angličtina – univerzální jazyk v IT

¹ *camelCase* neboli *Velbloudí notace* označuje způsob psaní víceslovných frází a nadpisů, v nichž jednotlivá slova nejsou oddělena mezerami, ale každé z nich začíná velkým písmenem – nezávisle na tom, zda by tomu tak podle pravidel pravopisu mělo být.

Př.: iPad, eBay, getSession(), JavaScript, ...

NazevTridy
-privatniAtribut : int +verejnyAtribut : int #protectedAtribut : double ~packageAtribut : char
+verejnaMetoda() : void -privatniMetoda(parametr : String) : int #protectedMetoda() : void ~packageMetoda() : void

Příklad třídy

- Různé zdroje uvádí různé možnosti zápisu

3.1.2.2 Viditelnost

- Symboly/znaky, označující viditelnost jednotlivých částí class diagramu

+	–	#	/	~
public	private	protected	derived	package
veřejný	privátní	chráněný	získaný	atribut viditelný v rámci balíku (package)

TABULKA 1: VIDETELNOST CLASS-DIAG.

Instance je konkrétní datový objekt, odvozený od třídy.

Každá instance má své vlastní atributy a metody podle vzoru (třídy). Atributy definující objekt však nemusí mít nutně pevné hodnoty – nupř. atribut `rodinny_stav` se může měnit v závislosti na tom, zda je objekt `Obcan` před nebo po svatbě, resp. po rozvodu apod.

3.1.3 Vztahy

- Nedílná součást diagramu dle UML
- Propojují jednotlivé instance (třídy)

1.1.1.0 Mohutnost vztahů (multiplicity)

Mohutnost vztahu udává, kolik instancí jedné třídy může být svázáno s instancí třídy druhé

0	0..1	1	0..*	1..*
Žádná instance (zcela výjimečně)	Žádná nebo právě jedna	Právě jedna instance	Žádná nebo více instancí	Jedna nebo více instancí

TABULKA 2: MULTIPLICITY

Vztahy dělíme na:

1. Vztahy na úrovni instance
2. Vztahy na úrovni třídy

3.1.3.1 Vztahy na úrovni instance

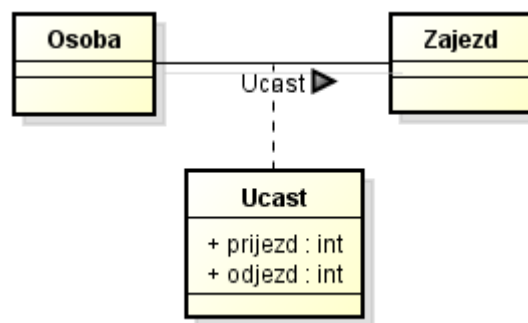
Závislost (Dependency)

= vztah mezi *závislými* a *nezávislými* prvky modelu

- vytvořena mezi dvěma prvky
 - Pokud dojde ke změně definice jednoho prvku (*server* nebo *cíl*) můžou se změny projevit i na straně druhé (dále jen *klient* nebo *zdroj*)
- tento vztah demonstrujete, zda se dané 2 prvky mohou vzájemně ovlivňovat
- asociace – jednosměrná

Asociace (Association)

- zprostředkovává vztah mezi dvěma entitami
- Nejčastější typ asociace = binární asociace (*dva konce*)
 - často reprezentovaná jako čára, spoj. 2 třídy
- Asociace však může propojovat libovolný počet tříd
(na obrázku ↓ je znázorněna asociace, které propojuje 3 třídy, tzv. *ternární asociace*)

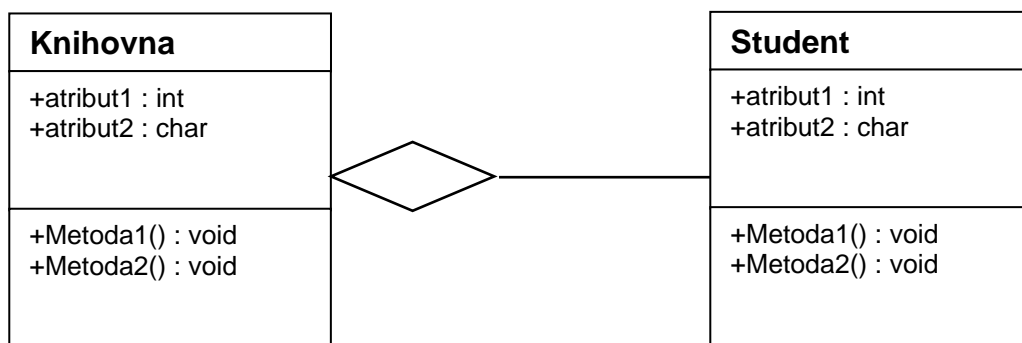


OBRÁZEK 3-1: ASOCIACE (ČÁPKA, 2012)

- Asociace může být v UML vzhledu pojmenována (Ucast), mít přiřazená jména rolí, mohutnost, viditelnost, či jiné vlastnosti
- Exist. 4 různé typy sdružení
 - (1) obousměrný
 - (2) jednosměrný
 - (3) agregace (vč. agregace složení)
 - (4) reflexivní
- Nejčastějším sdružením je *obousměrné* a *jednosměrné*,
např.: třída je spojena s rovinou třídy obousměrně

Agregace (Aggregation)

- Slabší vztah mezi *objektem* a *jeho částmi*
 - Objekt (celek) = agregací (seskupený) objekt
 - Části objektu = konstituční objekty (konstituenty)
- Vlastnosti agregace
 - (a) agregací objekt může existovat bez konstituentů
 - (b) konstituent může být součástí více seskupení (tzn. konstituent může mít více agregací objektů)

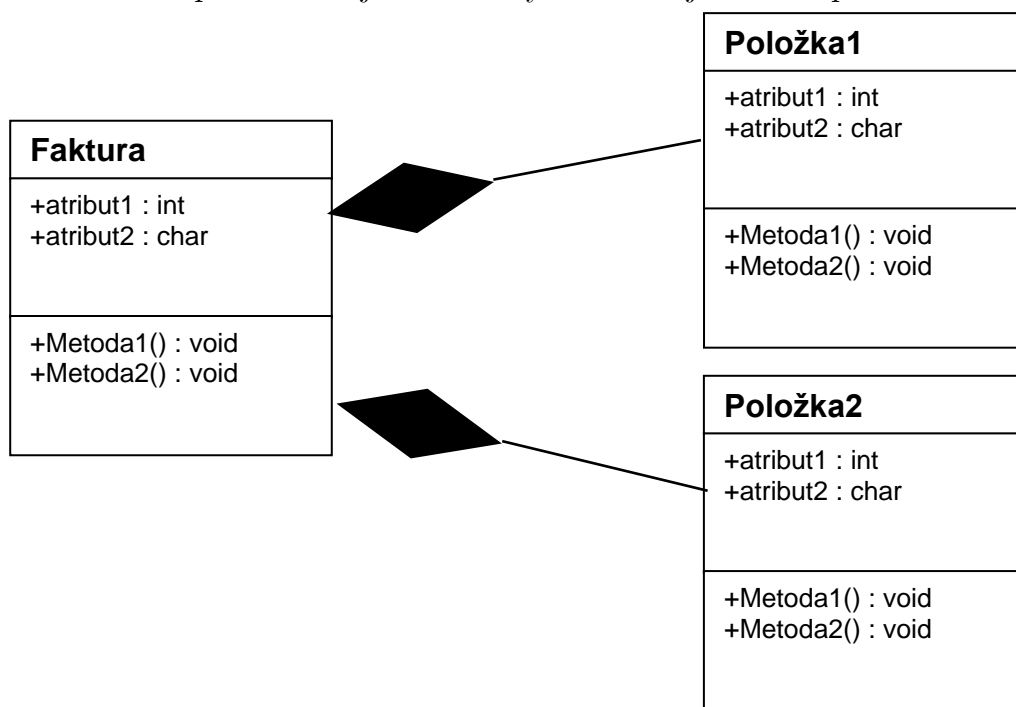


OBRÁZEK 3-2: AGREGACE

Příklad: Knihovna má studenty a knihy. Student však může existovat bez knihovny. Vztah mezi studentem a knihovnou je tedy agregace.

Kompozice, složení (Composition)

- Silnější vztah mezi *objektem a jeho částmi*
 - Objekt (celek) = kompozitní (složený) objekt
 - Části objektu = komponentní (složkové) objekty
- Vlastnosti kompozice
 - Kompozitní (složený) objekt nemůže existovat bez svých komponent
 - Komponentní objekt může být součástí jedné kompozice



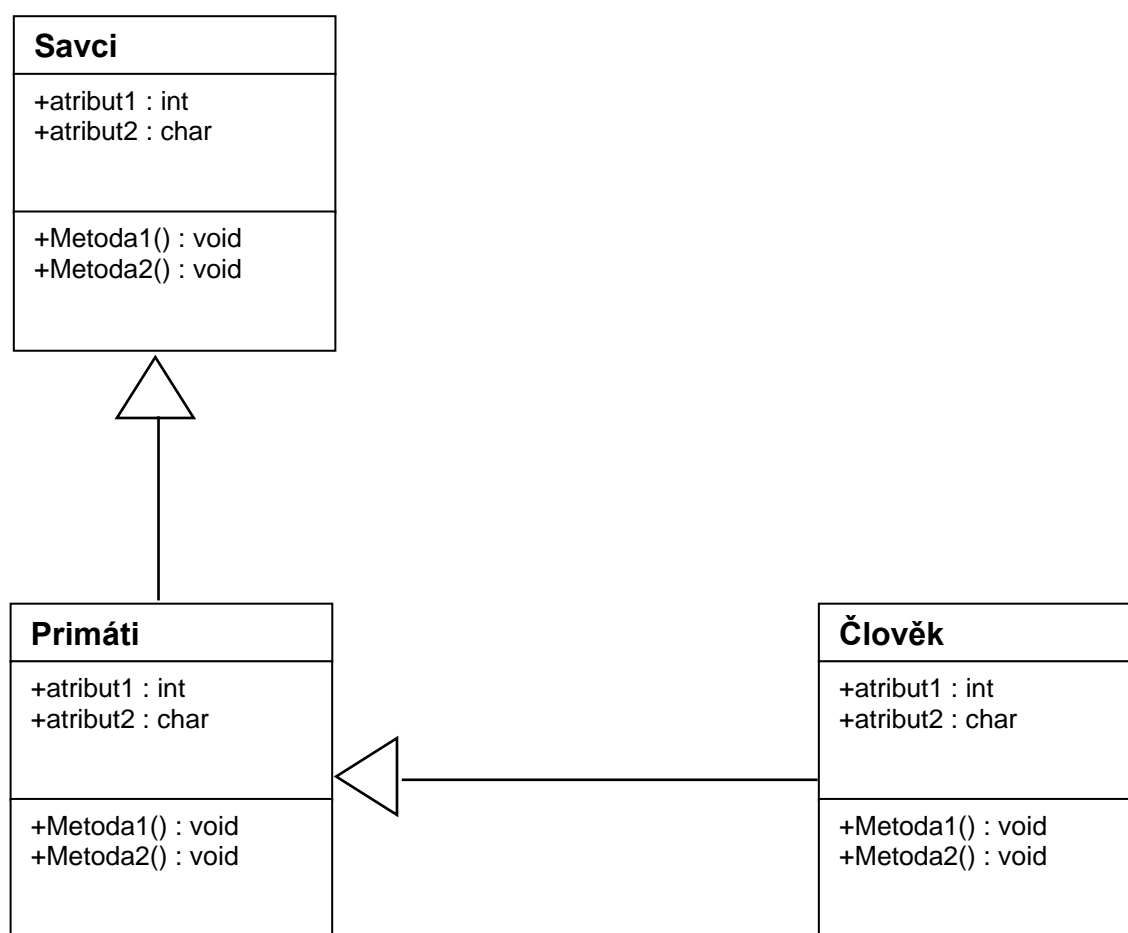
OBRÁZEK 3-3: KOMPOZICE

Příklad: Faktura se skládá z jedné nebo více položek. V okamžiku, kdy faktura z nějakého důvodu zanikne, jsou nám k ničemu i položky pro tuto fakturu.

3.1.3.2 Vztahy na úrovni třídy

Dědičnost (Generalization)

- Hierarchický vztah mezi třídami
- Jeden ze dvou příbuzných tříd (*podtříd*), je považován za specializovaný druh druhého (super typu). V praxi to znamená, že každý případ potomka je také instancí rodičovské třídy.



OBRÁZEK 3-4: DĚDIČNOST

Jednoduchý příklad z biologie: Lidé jsou podtřídou primátů, Ty jsou zase podtřídou savců atd... *To znamená, že všechny opice jsou savci, mají jejich vlastnosti atd. Jsou zvláštní podtřídou savců a mají své specifické atributy, ale ty základní, dědí od své rodičovské třídy, tedy savců.*

Realizace (Realization)

- vztah mezi zdrojovou (mateřskou) třídou a její realizací dceřinou třídou
- nejčastěji = vztah mezi rozhraním a třídou

Ve chvíli, kdy jsou tyto 2 třídy propojeny vztahem *realizace*, označujeme tak fakt, že třída implementuje všechny operace z daného rozhraní.

- také vztah mezi třídami, rozhraními, komponenty a balíčky, které spojuje klientský prvek s prvkem dodavatelským.

Závislost (Dependency)

- Slabší forma vztahu
 - naznačuje, že jedna třída závisí na jiné (používá v určitém časovém okamžiku parametr, resp. proměnnou závislé třídy)
- Někdy je vztah mezi dvěma třídami velmi slabý

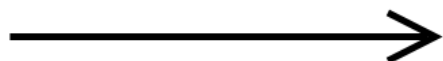
3.1.3.3 Shrnutí - přehled používaných typů šipek:

Vztahy na úrovni instancí

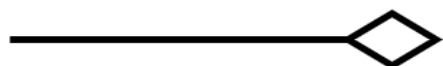
Závislost



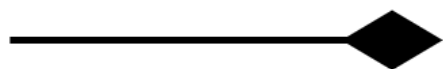
Asociace



Agregace



Kompozice



Vztahy na úrovni třídy:

Dědičnost



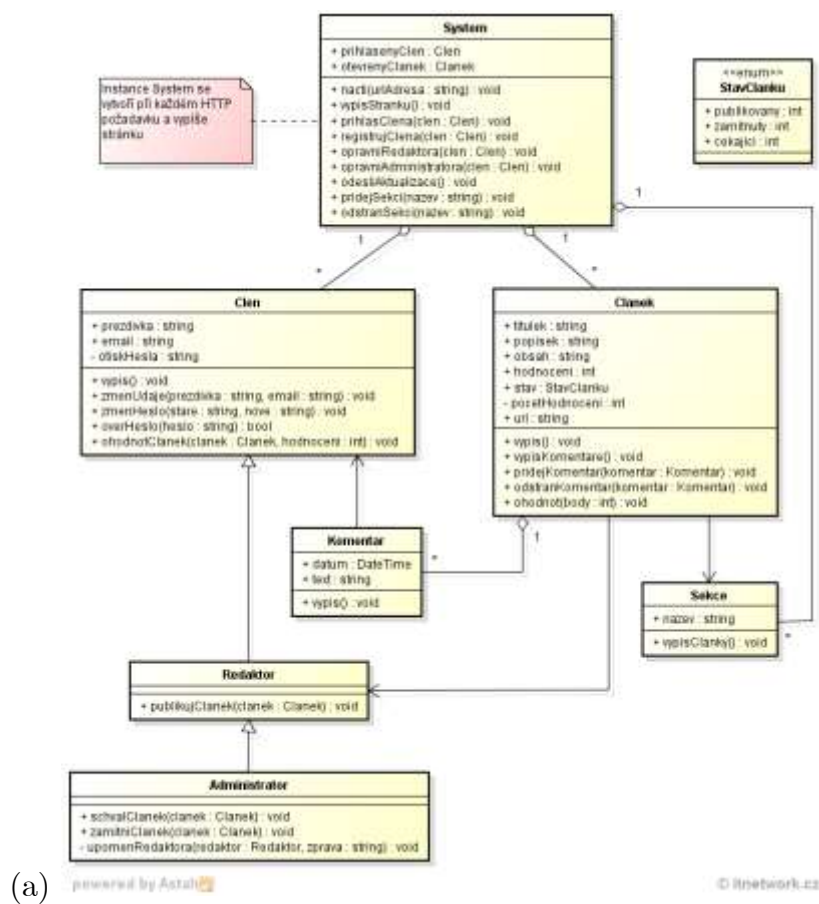
Realizace



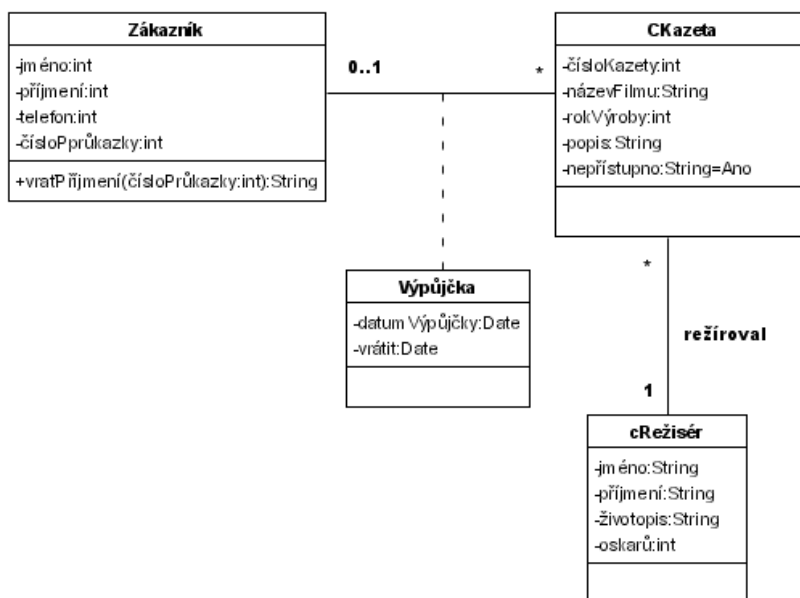
Závislost



Příklady digramů tříd:



OBRÁZEK 3-5: PŘÍKLAD CLASS DIAGRAMU 1 (ČÁPKA, 2012)



OBRÁZEK 3-6: PŘÍKLAD CLASS DIAGRAMU 2 (PAVUS, 2005)

3.2 Object diagram

Objektový diagram zobrazuje objekty a jejich spoje (links) v jednom okamžiku, tj. je to snapshot běžícího systému (či jeho části).

Objekty = instance tříd,

Linky = instance asociací

→ objektový diagram vypadá podobně, jako diagram tříd:

- Objektový diagram
 - Znázorňuje objekty a jejich relace (vztahy) v určitém čase.
 - Jedná se o snímek systému, který zachycuje aktuální objekty a vazby v konkrétním okamžiku
- Objekt
 - Základ objektového diagramu
 - Reprezentuje konkrétní entitu, která existuje v reálném světě (př. konkrétní zákazník, adresa, ...)
 - Neplést s třídou → reprezentuje popis struktury entit reálného světa

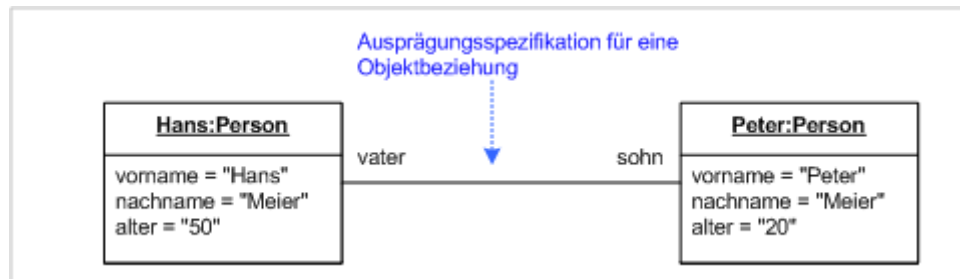
3.2.1 Zápis objektů

`název-objektu : NázevTřídy`

- Název objektu slouží k identifikaci objektu.
- Název třídy slouží k jednoznačnému určení typu objektu (instance různých tříd mohou mít stejné názvy).
- Například: Jan Novák může být instancí třídy Zaměstnanec nebo Zákazník.
- Třída rovněž definuje, jaké atributy objekt může mít.

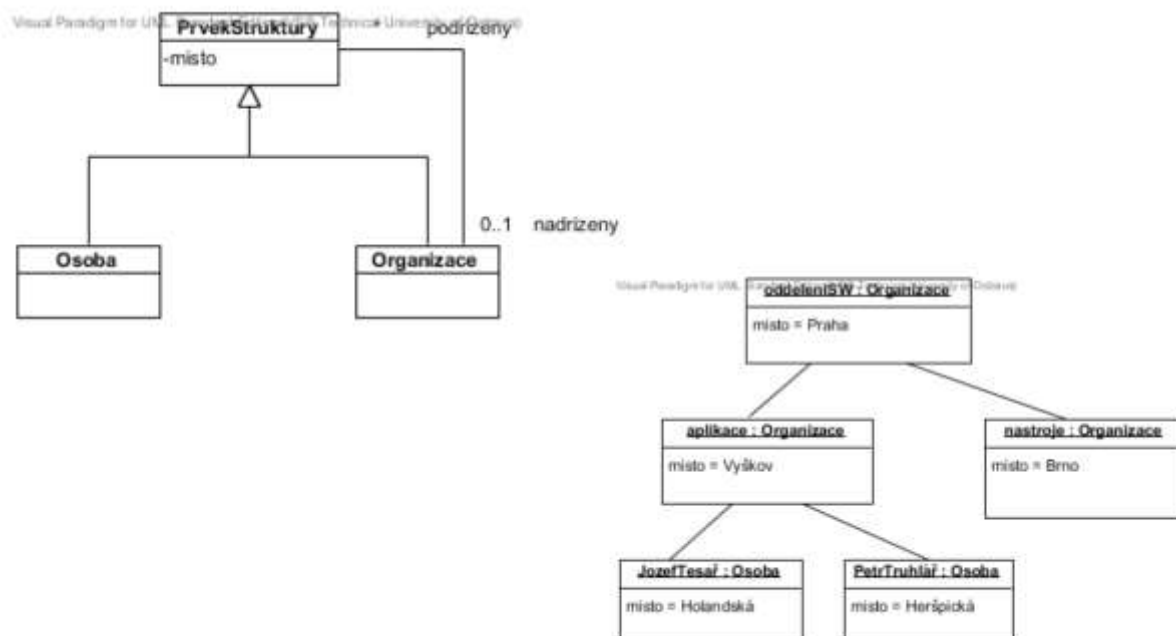
`: Název třídy` (anonymní objekt, bez názvu)

- Objektový diagram v definici atributů neuvádí typ, viditelnost, ...
- Obsahuje název atributu a jeho hodnotu.
- Hodnota atributu může být i prázdná nebo může obsahovat několik hodnot (pole, jiný objekt, seznam, ...).

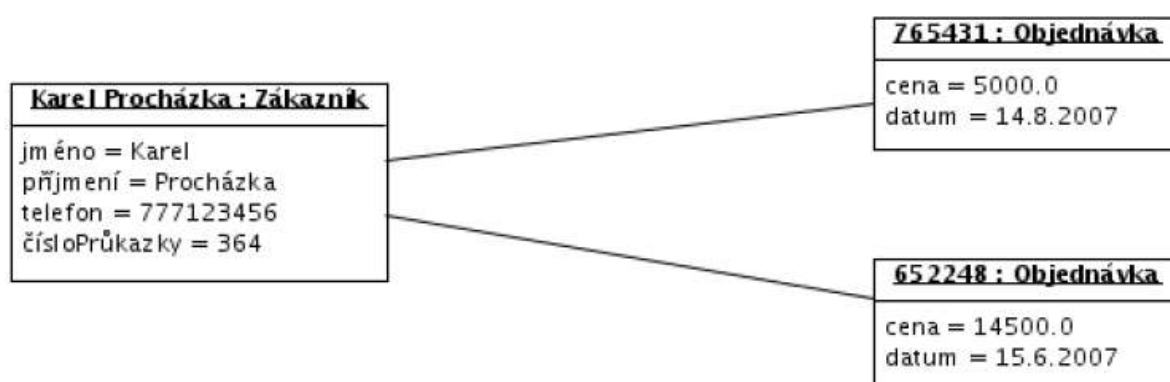
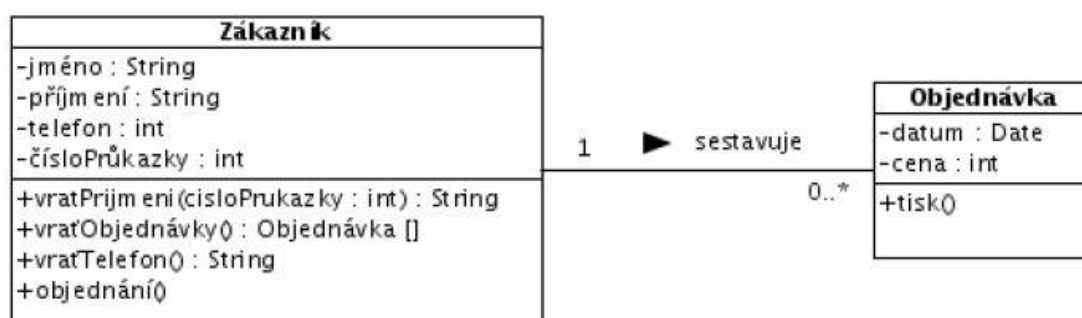


3.2.2 Vztahy mezi objekty

- Účelem objektů je reprezentovat data nebo informace a vazby.
- V diagramu jsou nadefinovány všechny možné vazby mezi třídami (asociace).
- Vztahu mezi objekty se říká propojení (link). Propojení je instance asociace.
- Pokud dva objekty vstupují do vztahu, který je popsán asociací, říkáme, že jsou propojeny.



3.2.3 Rozdíl mezi class diagramem a object diagramem



Class diagram	Object diagram
Tři oblasti (název, atributy, operace)	Dvě oblasti (název, atributy)
Oblast pro název obsahuje pouze název třídy	Oblast pro název objektu obsahuje identifikaci objektu a název třídy (<code>idObjektu : NázevTřídy</code>) nebo pouze název třídy (<code>: Název třídy</code>)
Třída definuje strukturu a typ atributů	Objekt definuje aktuální hodnoty atributů
Definice třídy zahrnuje operace	Objektový diagram operace nezahrnuje
Třídy jsou propojeny pomocí asociací, které obsahují název, role, násobnosti, omezení apod.	Vztah mezi objekty se nazývá spojení a může mít název nebo role (ale nezobrazuje násobnost). všechny spojení mezi objekty jsou 1:1

3.2.4 Použití objektového diagramu

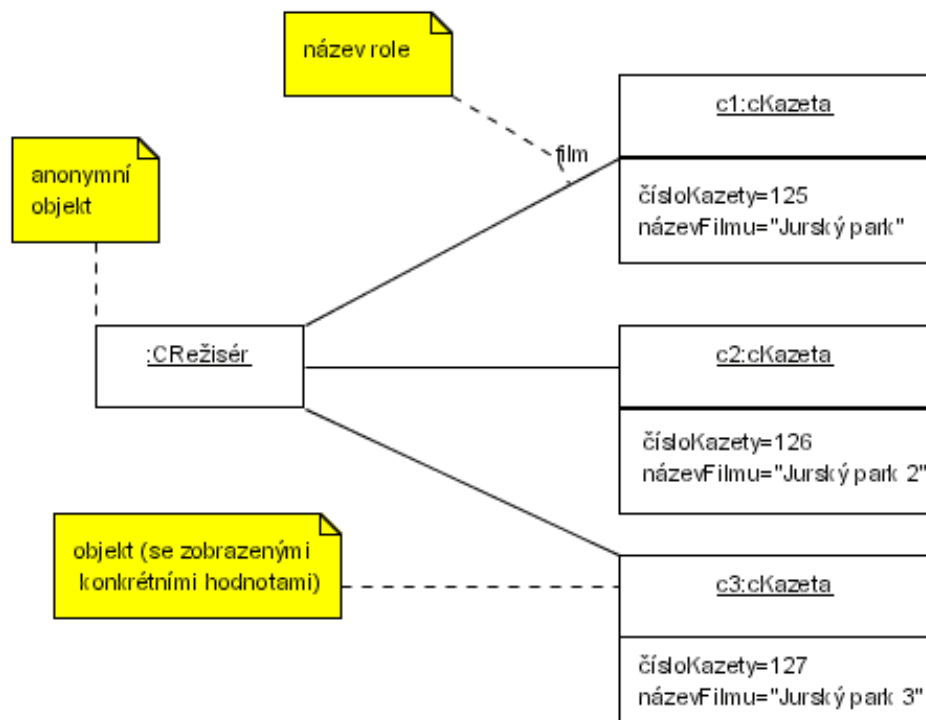
- Znázornění aktuálního stavu systému v konkrétním čase.
- Ověření správnosti diagramu tříd.
- Reverse engineering².
- Lepší pochopení budovaného systému.

3.2.5 Tipy pro tvorbu objektových diagramů

- Objektový diagram zachycuje pouze statickou strukturu objektového systému (nesnažíme se tedy popisovat dynamické chování).
- Některé systémy mohou obsahovat stovky nebo tisíce objektů (není cílem popsat je všechny).
- Dobrý objektový diagram se zaměřuje na konkrétní oblast s cílem dobře ji popsat (je dobré zamyslet se, jaký smysl mají jednotlivé objektové diagramy).
- Při kreslení OD se snažíme minimalizovat křížení linií (zvyšuje to přehlednost).
- Související objekty kreslíme blízko sebe.
- Ke zvýraznění důležitých informací(objektů) lze využít barvy nebo doplňující poznámky.

² Reverzní inženýrství (též zpětné inženýrství nebo zpětná analýza, angl. reverse engineering (RE)) je označení pro proces, jehož cílem je odkrýt princip fungování zkoumaného předmětu (např. počítačového programu), většinou za účelem sestrojení stejné či podobně fungujícího předmětu. Reverzní inženýrství může být v závislosti na situaci a právním systému nelegální (např. jako průmyslová špionáž nebo porušení práv duševního vlastnictví), ne však ve všech státech světa stejně.

3.2.6 Příklad



OBRÁZEK 3-7: OBJEKTOVÝ DIAGRAM (PAVUS, 2005)

Na obrázku je příklad objektového diagramu odvozeného z *class diagramu* půjčovny videokazet:

- objekt je znázorněn podobně jako třída – obdélníkem, ale název objektu je vždy podtržený.
- zápis `c1:CKazeta` značí objekt `c1`, který je instancí třídy `CKazeta` (obdobně je tomu s objekty `c2` a `c3`)
- zápis `:CRežisér` značí tzv. *anonymní objekt*: nevíme, jak se objekt jmenuje, ale víme, ze které třídy byl objekt odvozen
- třetí možnost zápisu by byla `c4`, tj. objekt bez vyznačení třídy: nevíme, ze které třídy je objekt odvozen (nebo to autor věděl, ale chtěl znázornit, že objekt `c4` mohl vzniknout z různých tříd)
- spojení (link) mezi objekty vzniklo jako instance asociace
- objekt `:CRežisér` je znázorněn v minimálním tvaru (jen jeho znázev), objekt `c1:CKazeta` je znázorněn s dalším oddílem (kompartiment) ukazujícím aktuální hodnoty atributů

- u objektu `c1` je vyznačena role - tj. ve vztahu anonymního objektu `:CRežisér` versus `c1:CKazeta` hraje tato kazeta roli filmu (který byl režírován spojeným objektem – režisérem)

Objektový diagram (Object Diagram) je snímkem objektů a jejich vztahů v systému v určitém časovém okamžiku. (Buchalceová, Pavlíček, Pavlíčková, 2007)

Z důvodu, že zobrazuje instance tříd, je též nazýván *diagramem instancí*. Používá se především pro znázornění určité konfigurace objektů či zobrazení vzájemně propojených objektů ve speciálních situacích, kdy je diagram tříd či sekvenční diagram nepostačující. Objektový diagram může být chápán jako speciální případ diagramu tříd vytvářený za účelem zdůraznit vazby mezi instancemi.

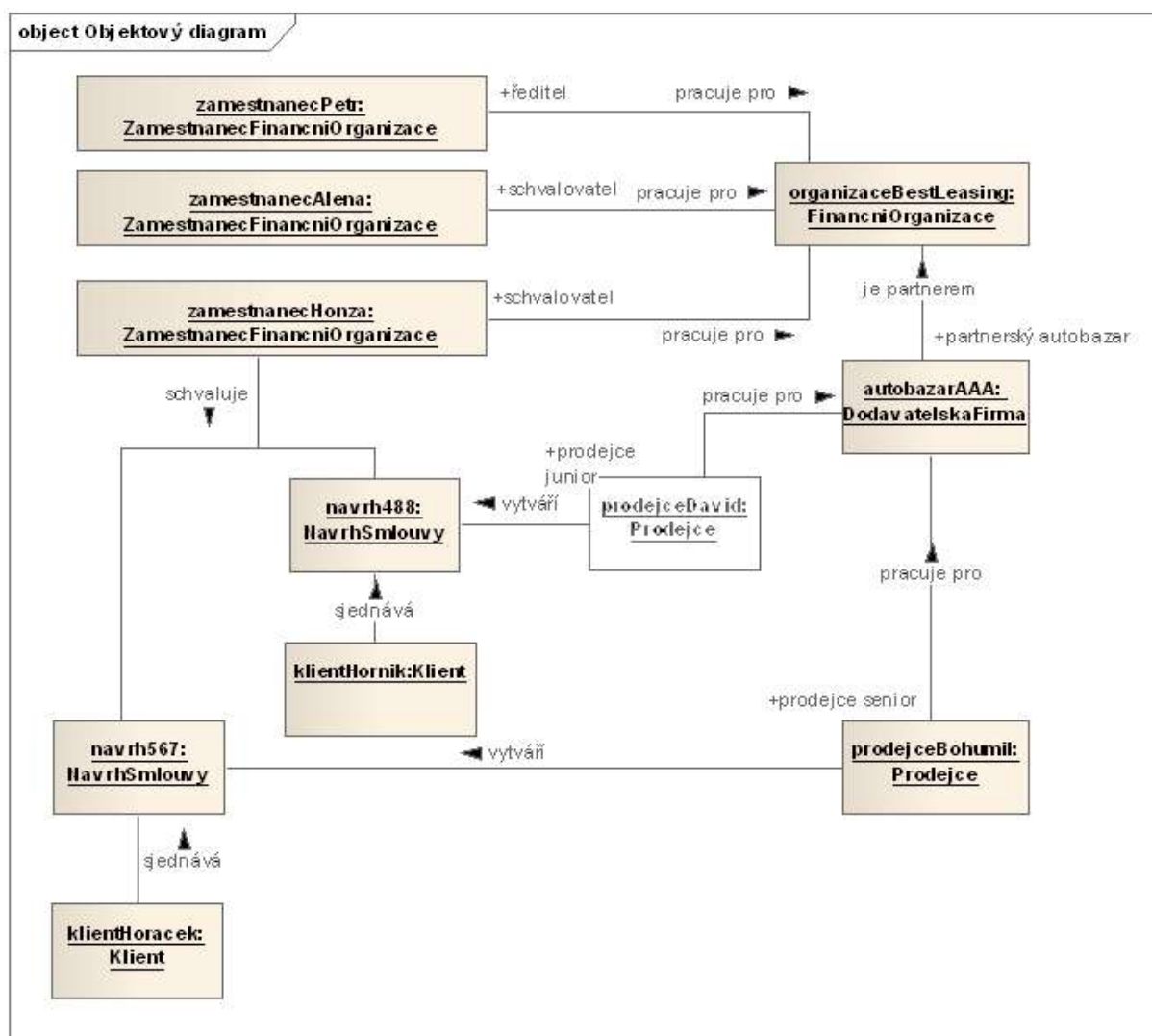
Objektový diagram je užitečný také v počátečních fázích projektu pro modelování ukázek problémové oblasti, které odhalují objekty a jejich vztahy. Často se také používá pro modelování testovacích případů (*test cases*) pro ověření správnosti diagramu tříd.

3.2.7 Prvky diagramu

Velmi podobná notace jako v diagramu tříd (*class diagramu*).

- Objekty (objects)
- Spojení mezi objekty (connections)

Atributy se u objektů vyznačují pouze v případě, že je to nutné pro jejich jednoznačnou identifikaci, metody se neuvádějí vůbec.



OBRÁZEK 3-8: PRVKY DIAGRAMU (ČÁPKA, 2012)

3.2.8 Doporučení

Dle *UML 2 a unifikovaný proces vývoje aplikací : Objektově orientovaná analýza a návrh prakticky* (Arllow, Neustadt, 2008) a *UML Bible* (Pender, 2003) platí pro názvy objektů následující doporučení:

- (1) Pro názvy objektů by měla být používána tzv. camelCase (velbloudíNotace), přičemž názvy by měly začínat malým písmenem.
- (2) Názvy objektů by měly být vždy podtrženy a nejčastěji ve tvaru Objekt:Třída (resp. :Třída).
- (3) Role objektů by měly být uvedeny pouze v případě, že nevyplývají z názvu objektu či spojení.
- (4) Uvádění násobností u spojení není nutné.
- (5) n -ární spojení, která mohou propojit více než dva objekty, by neměla být příliš často používána.

4 Objektové a datové modelování

Základní pojmy

- Program – posloupnost příkazů, které popisují určitou činnost
- Proces – běžící program
- Procesor – zařízení, které dokáže vykonávat příkazy programu
- Data – objekty, údaje, s nimiž pracují procesy
- Zdrojový kód – kód programu, zapsaný v určitém programovacím jazyce
- Cílový kód – binární kód (vytvoření po překladu zdrojového kódu)
 - je spustitelný

Objektové programování

Používá programovací jazyk k napodobení objektů skutečného světa definováním tříd

Zásady objektového programování

- Programovat proti rozhraní, a ne proti implementaci
- Dbát na důsledné zapouzdření a skrývání implementace
- Zapouzdřit a odpoutat části kódu, které by se mohly měnit
- Maximalizovat soudržnost (*cohesion*) entit (balíčků, tříd a metod). Každá entita by měla řešit jen jeden konkrétní úkol
- Koncentrovat zodpovědnost za řešení úkolu na jednu entitu – návrh řízený zodpovědnostmi (*responsibility driven design*)
- Minimalizovat vzájemnou provázanost (*coupling*) entit
- Vyhýbat se duplicitám kódu

Podstata a využití, základní objektové koncepty:

- Abstrakce – separování důležitých rysů od nedůležitých v závislosti na kontextu
- Zapouzdření (encapsulation) – data a operace objektu tvoří nedělitelný celek
- Dědičnost (inheritance) – schopnost objektů dědit vlastnosti a chování předka
- Polymorfismus – jev, kdy operace stejného jména je používána pro více objektů odlišných tříd.

Vlastnost, která umožňuje:

- jednomu objektu volat jednu metodu s různými parametry (parametrický polymorfismus)
 - objektům odvozených z různých tříd volat tutéž metodu se stejným významem v kontextu jejich třídy, často pomocí rozhraní
 - přetěžování operátorů znamená provedení operace v závislosti na typu operandů (operátorový polymorfismus)
- Komunikace (zasílání zpráv = volání metod) – objekty mezi sebou komunikují zasíláním zpráv. Výsledkem přijetí zprávy příjemcem je vykonání nějaké operace

4.1 Objektové a datové modelování

= snaha reálně popsat existující (nebo vznikající) systém v zjednodušené (abstraktnější) podobě

Význam, cíle a použití modelování

- snadné změny s nízkými náklady oproti reálnému systému
- usnadnění komunikace v týmu a se zákazníkem
- přehled o aktuálním stavu projektu
- návrh databází a datových úložišť
- integrace informačních systémů
- správa dat
- vytváření dokumentace

4.2 Princip tří architektur (P3A)

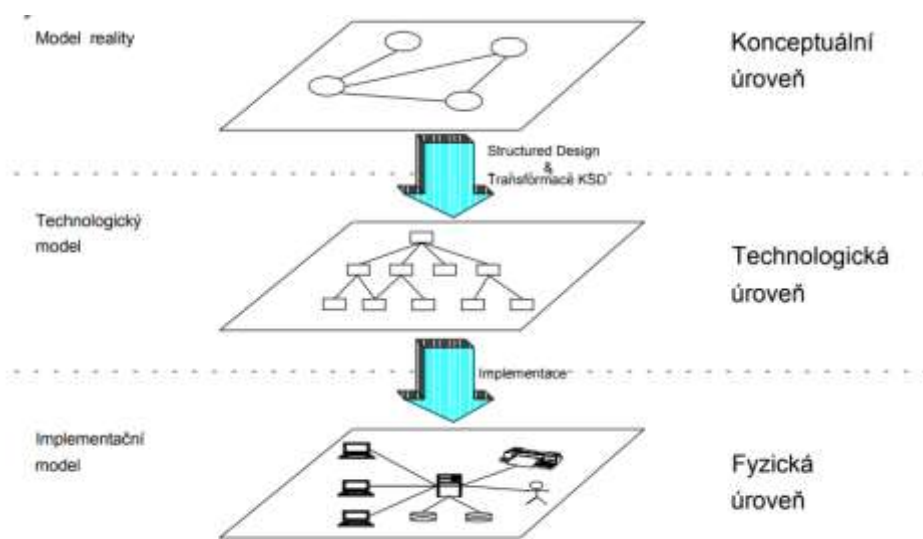
Princip tří architektur (P3A) slouží k popisu procesu vývoje daného modelu.

V souvislosti s architekturou P3A se také často hovoří o třech úrovních návrhu informačního systému:

konceptuální úroveň – obecný model reality, nejvyšší abstrakce, popisuje obsah systému, ne formu (CO je obsahem systému)

technologická úroveň – popis technologie s ohledem na prostředí implementace (JAK je obsah systémů v dané technologii realizován)

implementační úroveň – konkrétní popis detailů implementace v konkrétním prostředí (ČÍM je technologické řešení realizováno)



OBRÁZEK 4-1: ÚROVNĚ P3A (PŘÍSPĚVOVATELÉ WIKIPEDIE, 2016)

4.3 Softwarový proces

Proces = uspořádaná množina kroků, která vede k danému cíli (vyřešení úlohy)

Je-li tím cílem vytvoření nějakého programu (software), pak hovoříme o softwarovém procesu.

- Věda, jejíž náplní je zkoumat problémy související mj. se softwarovým procesem se nazývá softwarové inženýrství.

Vývoj software = novodobá problematika => není exaktně definováno, jak by měl správný softwarový proces vypadat.

Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla.

Existuje mnoho modelů softwarových procesů.

- Modelem rozumíme zjednodušenou realitu.
- Model používáme pro snadnější představu reality a pochopení problému

Příklad modelů:

1. Vodopádový model (*waterfall*)
2. Iterační (Model RUP /*Rational Unified Process*/)
3. Agilní (Scrum / XP)

Statická struktura procesu definuje:

- KDO (role)
 - Role = pracovníci, jejich kompetence a odpovědnost
- CO (artefakty)
 - Artefakty = entity, které jsou v procesu vytvářeny či používány (zdrojový kód, dokumentace, analýzy, ...)
- JAK (aktivity a toky)
 - Aktivita = činnost s cílem vytvořit artefakt

- Tok = workflow, postupnost aktivit
- KDY to má vytvořit

Nutno znát

- Specifikaci – co bude systém dělat
- Architekturu a design – z jakých „kostek“ a jak se bude systém skládat
- Implementaci – vlastní výroba systému
- Validaci – ověření, že systém dělá to, co má
- Další rozvoj – úpravy systému na základě měnících se požadavků

Časté problémy při vývoji softwaru...

- Nepřesné pochopení požadavků koncového uživatele
- Neschopnost pružně reagovat na změny požadavků
- Složitost údržby, nebo rozšíření software
- Pozdní objevení vad
- Nízká kvalita a výkon softwaru
- Nekoordinovaná práce v týmu

... a jejich příčiny

- Nepřesná (nedostatečná) specifikace požadavků a jejich ad hoc správa (management)
- Nejednoznačná a málo precizní komunikace
- Křehká (slabá) architektura
- Obrovská komplexnost
- Nekonzistence v požadavcích, návrhu a implementaci
- Nedostatečné testování
- Selhání při detekci rizik
- Absence managementu změn

Plánovaný vs. Agilní přístup

- Plánovaný softwarový proces (plan-driven)
 - Aktivita jsou plánovány dopředu (UML)
 - Pokrok je měřen a analyzován
 - V případě změn mohou nastat nepříjemnosti
- Agilní softwarový proces
 - Plánování po malých částech
 - Snadné reakce na změny v zadání

4.3.1 Modely

4.3.1.1 Vodopádový model (*waterfall*)

- Složen z několika kroků, která po sobě následují a nemohou začít dříve, než skončí předchozí fáze

Jednotlivé kroky:

1. Analýza požadavku
2. Návrh implementace
3. Implementace
4. Testování
5. *Nasazení
6. *Údržba systému

Nevýhody

- *Dlouhý vývoj* - prodleva mezi zadáním a finálním produktem, během vývoje zákazník nic nevidí a ani vývojáři nedokáží odhadnout kvalitu produktu
- *Přesné zadání* – na začátku je nutné vytvořit přesné a korektní zadání, aby byl výsledný produkt v pořádku.

Modifikace vodopádového modelu, resp. modely vycházející z waterfall modelu

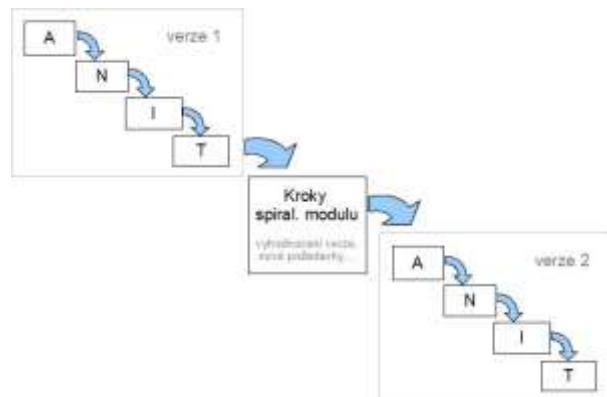


OBRÁZEK 44: VODOPÁDOVÝ MODEL

2. Inkrementální model
3. Spirálový model
4. Průzkumné programování

4.3.1.2 Inkrementální model

- Vychází z modelu Waterfall
- Princip verzování
- Finální produkt – rozdělen na dílčí verze – až poslední verze obsahuje plnohodnotný systém



OBRÁZEK 45: INKREMENTÁLNÍ MODEL

4.3.1.3 Spirálový model

- Vychází z modelu Waterfall
- De facto totéž, co *inkrementální model* s tím že mezi jednotlivé verze byly vloženy další procesy, např. zhodnocení verze z pohledu finálního systému, či přidání nových požadavků zákazníka.

4.3.1.4 Průzkumné programování

- Urputná snaha zjistit od zákazníka co vlastně chce, tím že mu předkládáme hotové systémy.
- Analýza požadavku je odbytá hrubou specifikací a vše směřuje k rychlé realizaci softwaru ke které se zákazník vyjádří.

4.3.1.5 Iterační (tradiční) přístup – model RUP (*Rational Unified Process*)

- Komerční produkt firmy Rational

Disciplinovaný přístup k přiřazování úkolů a zodpovědností v rámci vývojové organizace

Cílem je zajistit vytvoření produktu vysoké kvality požadované zákazníkem v rámci predikovaného rozpočtu a časového rozvrhu

4.3.1.5.1 KLÍČOVÉ AKTIVITY RUP

- Software je vyvíjen iterativně (v cyklech). Na konci každé iterace je spustitelný kód (verze). Produkt je vyvíjen ve verzích, které lze měnit v závislosti na požadavcích klienta
- Využívá existujících komponent, resp. vývoj softwaru se přesouvá do oblasti skládání produktu z již vytvořených komponent
- Model je průběžně vizualizován pomocí UML
- Součástí RUP jsou i metody pro správu požadavků, (postupy reakce na změny v zadání)
- Ve všech aktivitách se neustále ověřuje kvalita softwarového produktu.

4.3.1.5.2 VÝHODY A NEVÝHODY RUP

výhody	nevýhody
<ul style="list-style-type: none">– Obecnost, mohutnost– Iterativní přístup = včasné odhalení závad– Snazší správa změn– Provázanost s notací UML, dokumentace– Existence doplňkových nástrojů	<ul style="list-style-type: none">– Komerční, placený produkt– Rozsáhlost produktu, nevhodné pro malé týmy– Použití vyžaduje hluboké studium

4.3.1.5.3 CYKLUS VÝVOJE (CYKLY, FÁZE, ITERACE)

Každý cyklus vede k vytvoření verze systému, kterou lze předat uživatelům. Cyklus probíhá těmito fázemi:

1. Zadání - původní myšlenka je utřepána do vize koncového produktu
2. Rozpracování - dochází k analýze požadavku, tvorbě podrobných specifikací a návrhu architektury softwaru
3. Tvorba - software je naimplementován a otestován

4. Předání - zhotovená verze systému je předána uživateli do užívání, zahrnuje beta testování a zaškolení

OBRÁZEK 4: CYKLUS VÝVOJE RUP

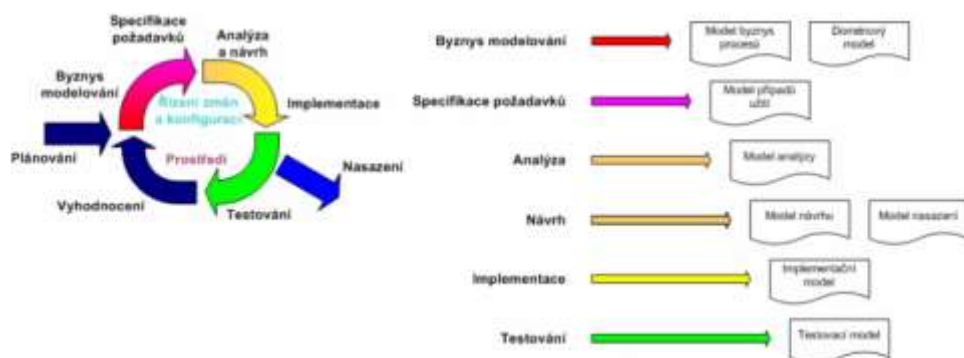
Každá fáze pak může být rozložena do několika iterací.



Iterace je úplná vývojová smyčka vedoucí k vytvoření spustitelné verze systému reprezentující podmnožinu vyvíjeného cílového produktu a která je postupně rozšiřována každou iterací až do výsledné podoby.

V rámci každé iterace proběhnou činnosti, které jsou uspořádány do toků charakteristických svým účelem - tzv. základních toků, jejichž výsledkem je část softwarového produktu (artefakt), a ty jsou:

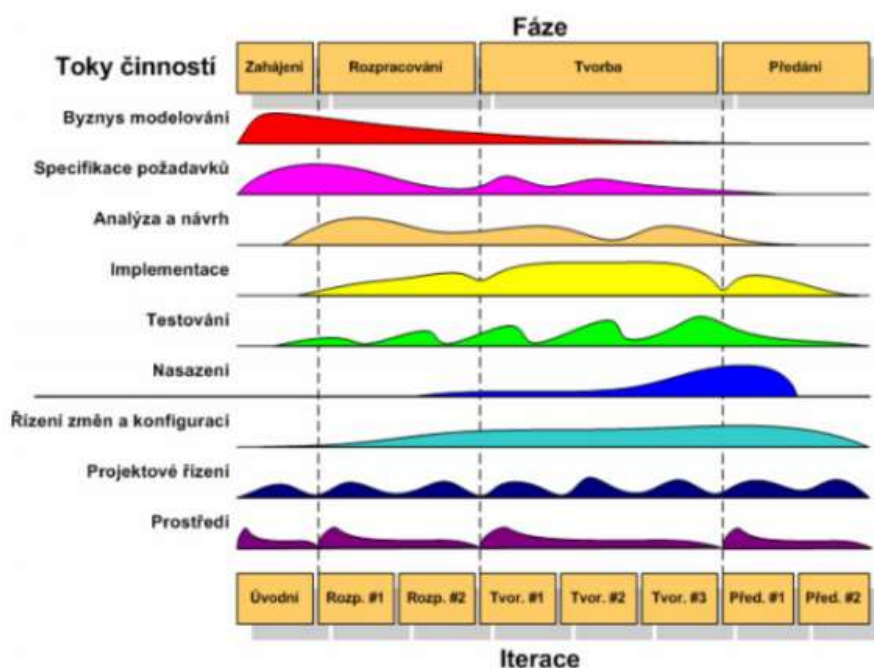
- Byznys modelování - popisující strukturu a dynamiku podniku či organizace.
- Specifikace požadavků - definující prostřednictvím specifikace tzv. případů použití softwarového systému jeho funkcionalitu.
- Analýza a návrh - zaměřené na specifikaci architektury softwarového produktu.
- Implementace - reprezentující vlastní tvorbu softwaru, testování komponent a jejich integraci.
- Testování - zaměřené na činnosti spjaté s ověřením správnosti řešení softwaru v celé jeho složitosti.
- Rozmístění - zabývající se problematikou konfigurace výsledného produktu na cílové počítačové infrastruktuře.



OBRÁZEK 5: TOKY RUP

Mimo základní toky existují i toky podpůrné, které nevytváří hodnotu, ale jsou nutné pro realizaci základních toků. A ty jsou:

- Řízení změn a konfigurací zabývající se problematikou správy jednotlivých verzí vytvářených artefaktů odrážejících vývoj změn požadavků kladených na softwarový systém.
- Projektové řízení zahrnující problematiku koordinace pracovníků, zajištění a dodržení rozpočtu, aktivity plánování a kontroly dosažených výsledků. Nedílnou součástí je tzv. řízení rizik, tedy identifikace problematických situací a jejich řešení.
- Prostředí a jeho správa je tok činností poskytující vývojové organizaci metodu, nástroje a infrastrukturu podporující vývojový tým.



OBRÁZEK 6: SCHÉMATICKÉ ZOBRAZENÍ PROCESU RUP A VŠECH JEHO TOKŮ, JAK JSOU VYVÍJENY V ČASE.

Skupiny metod původně určených pro vyvíjení softwaru založené na iterativním a inkrementálním vývoji nazveme jako tzv. Agilní metodiky. Umožňují rychlý vývoj softwaru a zároveň dokáží reagovat na změnu požadavků v průběhu vývojového cyklu. Podle těchto metodik se správnost systému ověří jediné pomocí rychlého vývoje, předložení zákazníkovi a následných úprav dle zpětné vazby.

Do agilních metodik patří další 2 modely SCRUM a XP.

TABULKA 3: POROVNÁNÍ TRADIČNÍCH - ITERAČNÍCH METOD A METOD AGILNÍCH

Tradiční přístup - RUP	Agilní přístup - SCRUM, XP
Důraz na procesy a nástroje	Komunikace, individualita (kreativita)
Obsáhlá dokumentace	Provozní software
Uzavírání smluv s restrikcemi	Spolupráce se zákazníkem
Striktní plnění plánu	Reakce na změnu

4.3.1.6 Agilní přístup – model Scrum

= česky též mlýn / skrumáž

Jedná se především o metodiku pro týmový vývoj softwaru. Tato metodika má několik zajímavých myšlenek a nápadů (denní meetingy, samostatné přiřazování vývojářů na úkoly, ...), které přispívají k zrychlení a zefektivnění vývojového cyklu.

4.3.1.6.1 KLÍČOVÉ ČÁSTI

- Každodenní setkávání týmu – každý zde referuje o své činnosti a problémech na které narazil
- Iterativní vývoj – období jedné iterace se nazývá Sprint a trvá 2 – 4 týdny
- Demo – výsledek Sprintu, předvedeno zákazníkovi. Ten poskytne feedback, který iniciuje nový Sprint.

4.3.1.6.2 ROLE

V modelu Scrum rozeznáváme tři role:

- Product Owner (PO) – komunikace se zákazníkem
- Scrum Master (SM) – vedoucí vývojového týmu
- Scrum Team Member (STM) – člen vývojového týmu

4.3.1.6.3 VÝVOJOVÝ CYKLUS

1. Komunikace PO se zákazníkem zadání nových požadavků.
 - Ty tvoří tzv. User Stories (požadavky zákazníka).
2. Na *Sprint Planning Meetingu* sejde PO, SM a STM a společně odhadnou zadané *User Stories*.
3. Podle priorit naplánují budoucí *Sprint*, tedy vyberou *user story*, které budou v tomto Sprintu dokončeny.
 - Tyto *user story* jsou poté ve *Scrum Teamu* dále rozepsány do Sprint Backlogu (popis problému na technické úrovni pro programátora) a ty následně do Tasků (samostatný úkol pro člena týmu).
 - Během trvání sprintu (asi 2 - 4 týdny) probíhají každodenní meetingy (Daily Scrum Meetings).
 - Na konci Sprintu, resp. Release je zákazníkovi předvedeno demo vzniklých úprav. Zákazník se k nim může vyjádřit a zhodnotit, zda jsou splněny jeho požadavky.
 - Scrum díky tomu dokáže rychle reagovat na změny zadání od uživatele.

4.3.1.7 Agilní přístup – model XP (Extrémní programování) (x)

4.3.2 Modely z různých pohledů

Pohledy:

- Flexibilita: reakce na změny, rychlost, náklady
- Predikovatelnost: Vím, co a kdy to dostanu a kolik mě to bude stát
- Architektura a design: Dobře navržený systém, konzistentní dodržování principů návrhu
- Implementace: Prostor pro dodání kvalitního díla, požadavky na programátory
- Dokumentace
- Spolupráce se zákazníkem, požadavky na součinnost: Jak moc se musí zákazník podílet na projektu
- Smlouvá na dodávku

Pohled / model	Vodopád	Iterativní	Agilní
Flexibilita	Nepružný, vysoké náklady	Lze zakomponovat změny do další iterace, náklady nižší než u vodopádu	Snadné, změny se očekávají, náklady na změny minimální nebo nízké
Predikovatelnost	Vysoká, máme plán	Vysoká, máme plán	Nízké, plán pouze na krátké období Víme jen co dostaneme během dalšího sprintu
Architektura a design	Vysoká	Vysoká Riziko zanesení problémů při dalších iteracích	Nízká Riziko zanesení problémů při každém sprintu
Implementace	Kvalitní Revize, coding standards	Kvalitní Dost prostoru pro QA Revize, coding standards	Nutný kvalitní tým, může být problematická Riziko nekvalitní práce (není prostor na revize)

	Dost prostoru pro QA ³	Riziko zanesení problému při dalších iteracích	
Dokumentace	OK	Nutno dodržet napříč verzemi	Nízká Obtížné udržovat napříč sprinty
Spolupráce se zákazníkem	V přesně definovaných okamžicích Lze dobře plánovat	V přesně definovaných okamžicích Lze dobře plánovat	V průběhu celého projektu Velké riziko selhání, pokud nebude
Smlouvá na dodávku	Ano	Ano, nutno ověřit rozsah verzí	Ne, požadavky nejsou dopředu známy

4.4 UML

= grafický jazyk pro specifikaci, vizuální popis, tvorbu a dokumentaci jednotlivých součástí softwarového systému. Jazyk pro OO modelování.

Modelování typových úloh – nástrojem je Diagram typových úloh (*Use Case Diagram*), vyjadřuje vztahy aktér × úloha a úloha × úloha. Důležité jsou scénáře.

Modelování tříd – nástrojem je Diagram tříd (*Class Diagram*) (= základní strukturální diagram UML).

- Třída – abstraktní definice množiny objektů (*atributy, metody*)

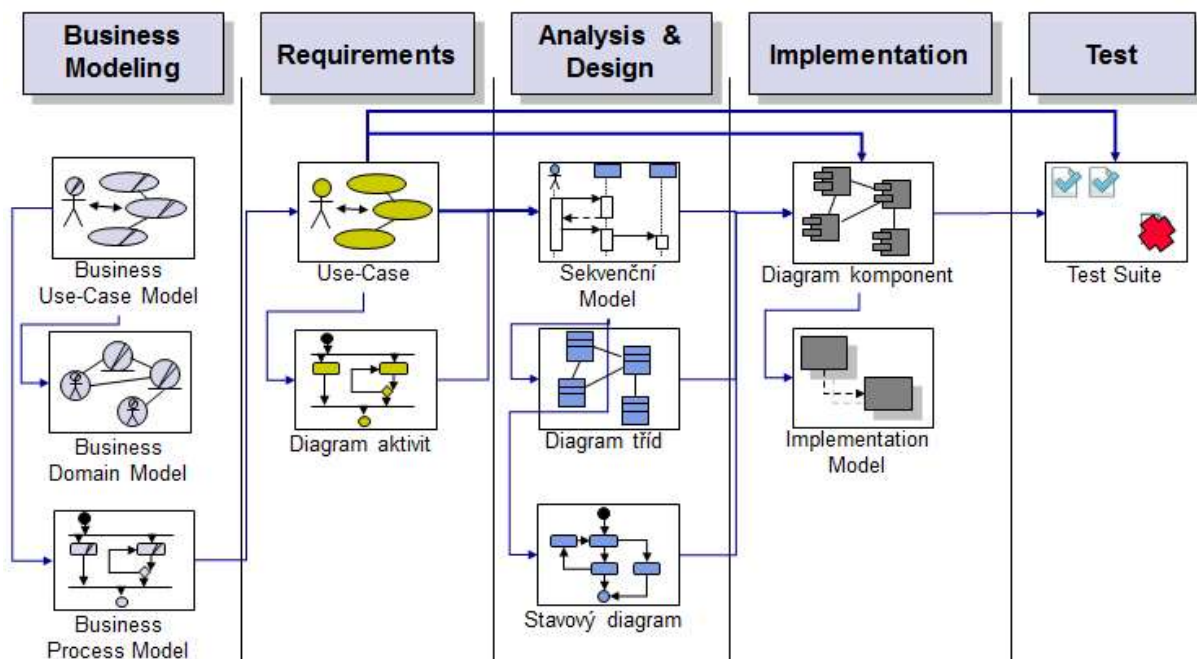
³ Quality assurance (zkráceně QA) se týká obecně všech procesů od návrhu, vývoje, nasazení, údržby až po dokumentaci produktu.

Cílem této aktivity je dohlédnout, že produkt je vytvářen tak, že jeho jednotlivé části budou mít odpovídající kvalitu, která byla určena. Původní anglický termín se odkazuje už na předcházení chybám. Jako součást QA lze označit i řízení kvality (quality control), především pak výstupní kontrola.

- Rozhraní tříd – definuje kontrakt, ke kterému se třídy přihlašují
- Asociace – definují vztahy mezi objekty

Modelování dynamiky systému – dva základní (vzájemně izomorfní⁴) diagramy
OSD a OCD

- sekvenční diagram (*Object Sequence Diagram* – OSD) – zobrazuje interakci objektů s důrazem na časovou posloupnost, mapován k jedné typové úloze
- diagram objektové spolupráce (*Object Collaboration Diagram* – OCD) – pohled na strukturu spolupráce – vztahy mezi objekty
- diagram aktivit – lze použít pro modelování chování, ale není izomorfní s OSD a OCD, modeluje typovou úlohu jako posloupnost aktivit – ne jako interakci uživatele a systému



⁴ Izomorfismus = stejnorodost, shodnost struktury, ekvivalence (viz ABZ.cz)

4.5 Událostí řízené programování (*Event-Driven Programming*)

Událost vzniká buď...:

- ...*jako* výsledek interakce (vzájemného působení) mezi uživatelem a grafickým uživatelským rozhraním (*graphical user interface*, GUI), *nebo*...
- ...*jako* důsledek změny vnitřního stavu aplikace nebo operačního systému

Obsluha událostí

Úsek kódu, který je při vzniku událostí automaticky vyvolán a provádí činnost k události připojenou (*Event Handler*)

Typy událostí

- Klik / dvojklik
- Stisk/uvolnění klávesy/tl. myši
- Změna stavu komponenty (připojení, odpojení)
- Událost systému, nebo zpráva časovače

Cvičení

Navrhni doménový model se základními třídami, atributy a vztahy mezi třídami.

Rozšiř doménový model o datové typy atributů, operace a doplňující třídy (např. výčtové typy).

Vytvoř implementační model z předmětu IT s detailním popisem struktury.

5 Sekvenční diagram

Modelování interakcí mezi objekty

- Modelování zpráv
- Vytváření a ukončování objektů
- Modelování aktivity objektů
- Cykly, alternativy, podmínky

5.1 Interakce mezi objekty

- Cílem je popsat, jak se konkrétní objekty chovají, pokud jsou vzájemně propojeny:
 - *Class diagram* znázorňuje statickou strukturu systému (vazby mezi objekty)
 - *Object diagram* znázorňuje chování instancí tříd v konkr. čase
 - *Sekvenční diagram* znázorňuje vzájemné působení mezi aktivní objekty
- Cílem interakcí je popsat:
 - Jakým způsobem spolu objekty komunikují, aby splnili dané zadání
 - Jak systém reaguje na vnější i vnitřní podněty od uživatele
 - Jaké informace je potřeba přenášet mezi objekty (tj. přechází redundanci)
- Chování systému může být velmi rozsáhlé a komplexní
 - Velké systémy je nutné rozložit na menší části a při popisu se zaměřit na nějakou konkrétní situaci nebo scénář (stejně tak objektové nebo class diagram nemusí modelovat celý systém v jednom diagramu).

Strukturální diagramy

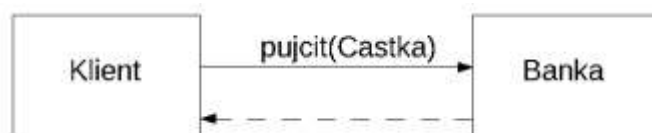
- **Diagram tříd**
- Diagram nasazení
- Diagram komponent
- Diagram instancí (Obejktový diagram)
- Diagram profilů
- Diagram balíčků

Diagramy chování (Behaviorální diagramy)

- **Diagram aktivit**
- Diagram užití
- Stavový diagram

Diagramy interakcí

- Sekvenční diagram
- Diagram interakcí
- Diagram komunikací



Velmi jednoduchý sekvenční diagram

Sekvenční diagram (ang. Object Sequence diagram, zkr. OSD) je jeden z *diagramů interakcí jazyka UML*.

Zachycuje časově uspořádanou posloupnost zasílání zpráv mezi objekty. Sekvenční diagram nejčastěji znázorňuje spolupráci několika vzorových objektů v rámci jednoho případu užití⁵.

Téměř izomorfní s diagramem komunikací (dají se převádět)

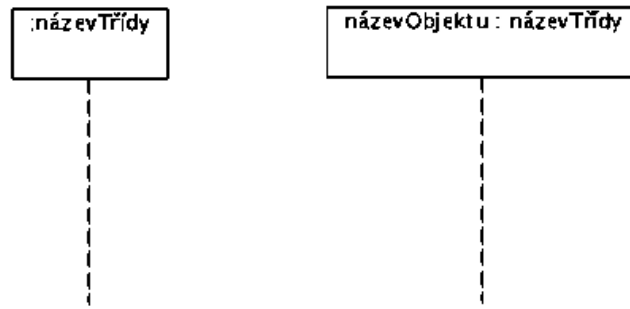
Použití sekvenčního diagramu = důležité časové souvislosti interakcí,
nevidíme v něm zobrazené vztahy mezi objekty

- Diagram zachycuje časovou posloupnost posílání zpráv.
- Slouží k ukázce interakce mezi objekty (modelování zpráv posílaných mezi objekty)
- Vizualizace komunikace se provádí z hlediska času (je zřejmé pořadí zpráv)
- Při tvorbě diagramu je možné objevit dodatečné požadavky na rozhraní objektů, což může vést k vylepšení diagramu tříd.

5.2 Vizualizace

- K vizualizaci se používá symbol pro objekt společně s časovou osou
- Pokud není uveden konkrétní objekt, myslí se tím množina objektů daného typu nebo jakýkoliv objekt daného typu

⁵ V softwarovém a systémovém inženýrství je případ užití (anglicky use case) seznam kroků, který obvykle definuje interakci mezi tzv. rolí (v UML označována jako "actor") a systémem. Za roli může být člověk nebo externí systém.



5.2.1 Zprávy (interakce)

- Zpráva je formálně popsána interakce (komunikace) mezi objekty.
- Poslání zprávy má za následek spuštění metody cílového objektu.
- Zpráva mezi objekty je modelována šipkou (vždy od odesílatele k příjemci):
- Různé šipky mají různý význam
 - Zpráva se modeluje plnou šipkou.
 - Odpověď se modeluje přerušovanou.

Druh chování (zaměřuje se na výměnu informací mezi objekty. Informace se vyměňují pomocí zpráv.

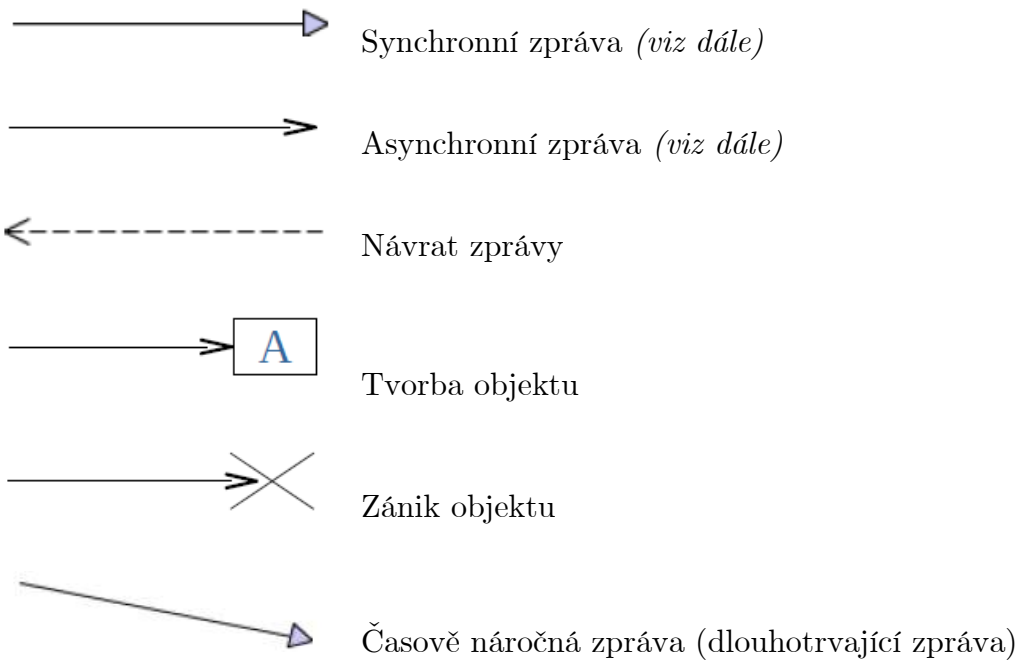
Sémantika⁶ interakce je dána dvěma množinami:

1. Obsahuje platné sekvence poslaných a přijímaných událostí
2. Obsahuje neplatné sekvence

Všechny ostatní sekvence jsou pro interakci nedůležité.

⁶ nauka o významu a změnách významu jazykových jednotek, významosloví

5.2.1.1 Typy zpráv



Pozor na jiná značení šipek od verze UML 1.4.

5.2.1.2 Synchronní, asynchronní a „self“ zprávy

Synchronní zpráva

- Odesílatel čeká na odpověď od příjemce. Poté co ji obdrží, pokračuje v činnosti



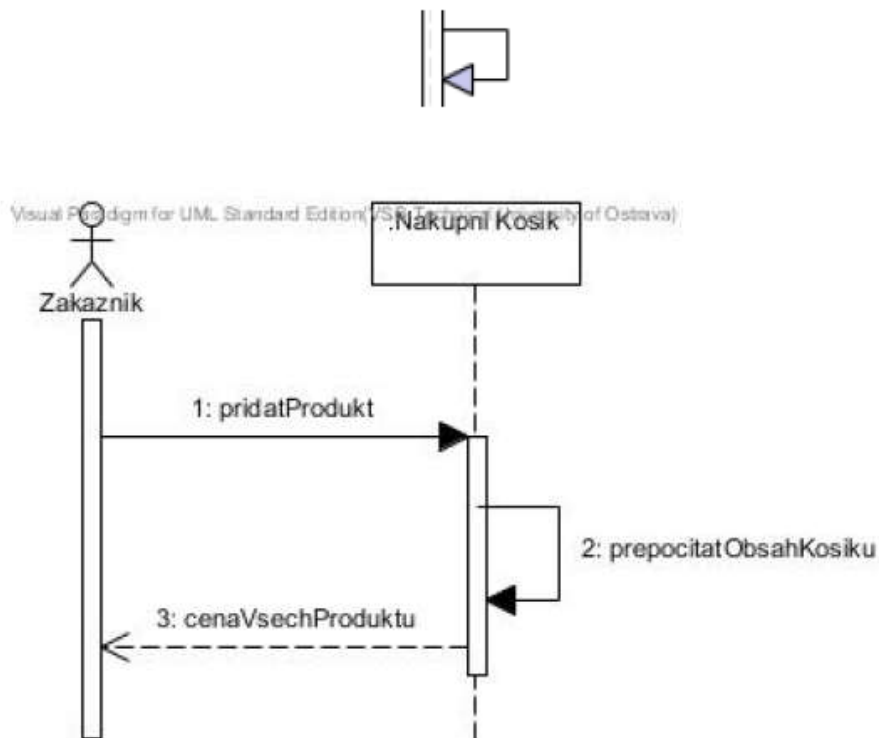
Asynchronní zpráva

- Odesílatel nečeká na odpověď od příjemce. Po odeslání ihned pokračuje ve své činnosti



Zpráva self

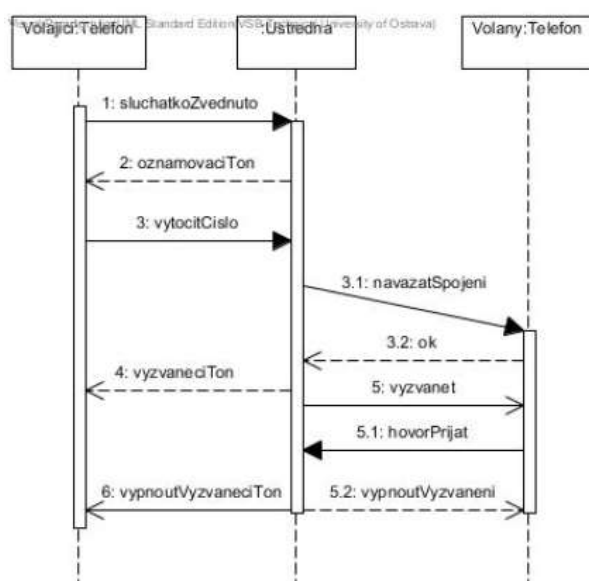
- Zpráva, kterou odesílatel posílá sám sobě



Příklad – zpráva „self“

Časově náročné zprávy (Dlouhotrvající zpráva)

- Používají se v případě, že zaslání zprávy vyvolá operaci, která může trvat delší dobu



Příklad – časově náročná zpráva

5.2.2 Časová osa

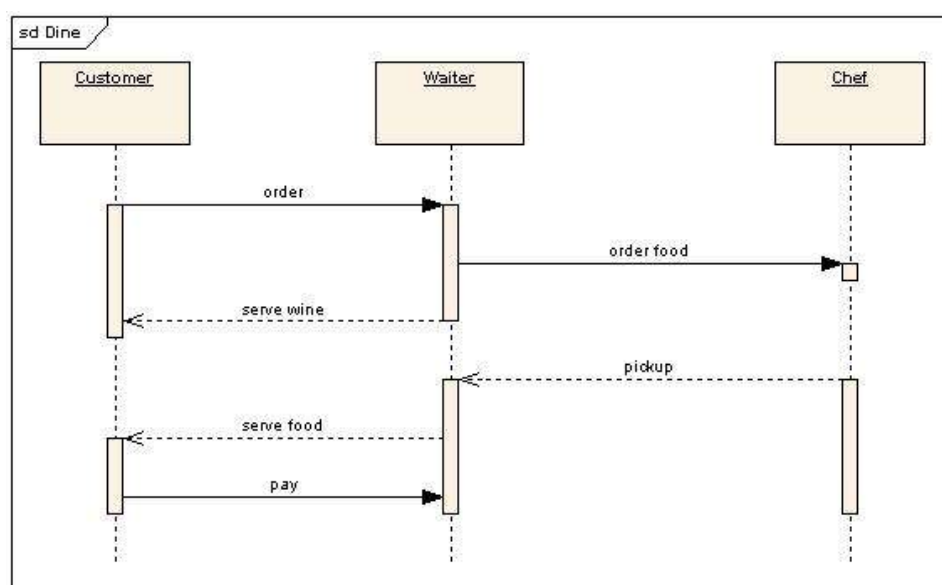
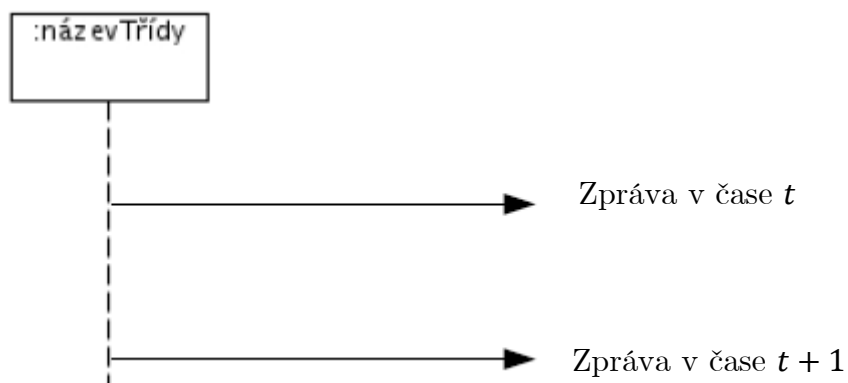
Svislá osa = čas (čas běží shora dolů)
Vodorovná osa = objekty

- Jeden z nejdůležitějších pojmů v sekvenčním diagramu
- Reprezentuje jednotlivé účastníky (participanty) interakce
 - o Znázorňuje, kdy participant žije
- Aktivita objektu (execution specification, focus of control)
 - o Vyjadřuje se zdvojeným úsekem (obdélníkem) na lifeline
 - o Ukazuje periodu, kdy je který objekt aktivní
 - Provádění určité činnosti, chování (vč. podřízeného objektu a dalších objektů)
 - U *aktivního* objektu vyznačuje celý jeho život,
U *pasivního* objektu časový interval, kdy je prováděna operace objektu, včetně čekání na návrat ze zavolané operace; v programátorském světě je to analogické době, po kterou je určitá hodnota ve "stack" - v zásobníku)

Zprávy mezi účastníky interakce jsou zobrazovány pomocí šipek. Typ zprávy určuje notace šipky (viz *kap. 5.2.1 Zprávy (interakce)*, str. 100).

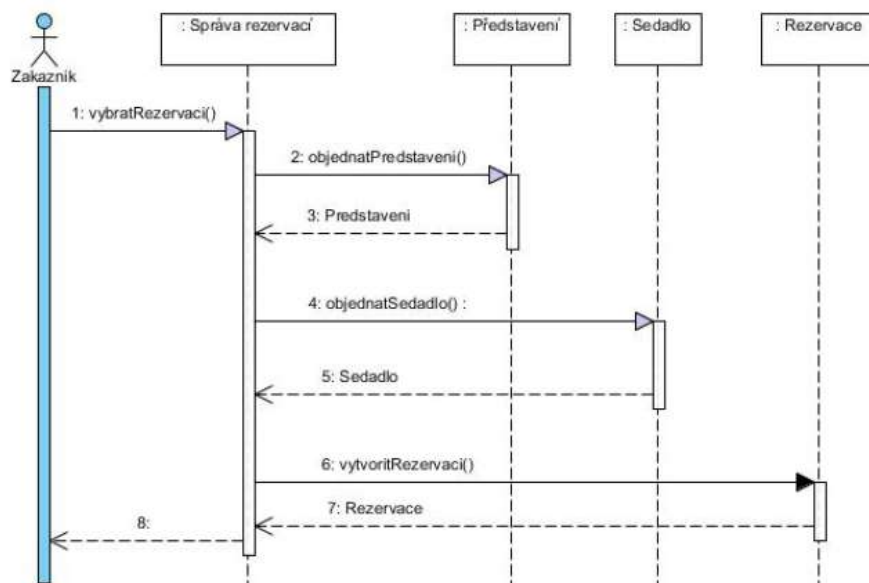
Záhlaví nesmí být prázdné. Časová osa může zobrazit v hlavičce klíčové slovo `self`.

V takovém případě čára života reprezentuje instanci klasifikátoru, kterému interakce náleží.



Příklad – jednoduchý sekvenční diagram

Diagram ukazující sekvenci úkonů v restauraci. Zákazník si objedná jídlo, číšník objednávku nahlásí do kuchyně, poté jídlo vyzvedne a donese zákazníkovi; ten pak zaplatí.



Příklad – jednoduchý sekvenční diagram

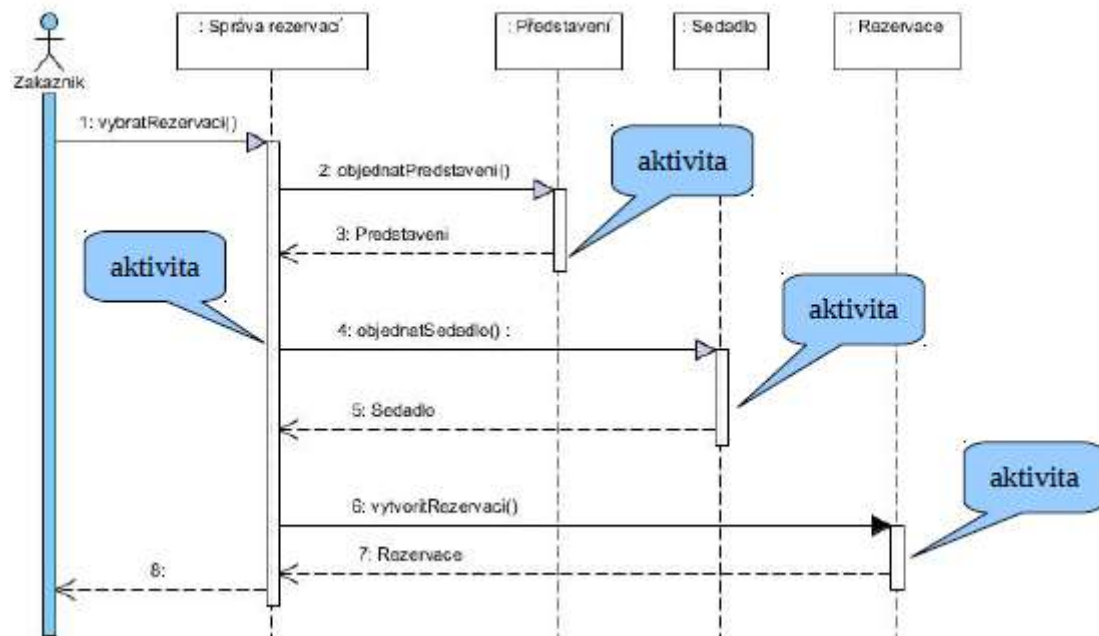
Cvičení

- < Urči pořadí, v jakém se budou posílat zprávy
- < Urči typy jednotlivých zpráv

- Tvorba sekvenčního diagramu je jednodušší, pokud je hotova alespoň první verze případů užití a class diagramu.
- Zde je možné čerpat interakce mezi objekty.
- Jednotlivé zprávy mezi objekty jsou číslovány podle pořadí. (Většina nástrojů provádí číslování automaticky)
- Při tvorbě sekvenčních diagramů se často narazí na nové operace, které je třeba přidat do tříd.

5.2.3 Aktivita objektu

- Používá se ke znázornění, kdy je objekt aktivní (něco dělá) a kdy je neaktivní.
- znázorňuje se obdélníkem podél časové osy objektu

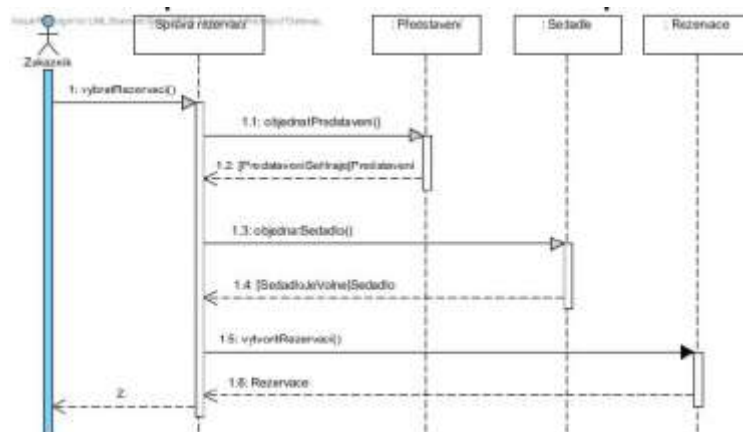


Příklad – aktivace objektu

5.2.4 Podmínky

- Využívají se v případě, že posílání zprávy je omezeno podmínkou – notace:

[podmínka]

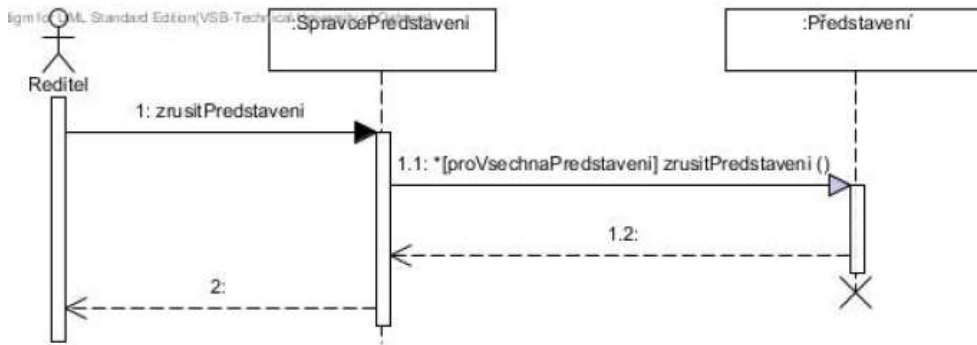


Příklad – podmínky

- Úplná podmínka bude lépe řešitelná pomocí alternativy.

5.2.5 Iterace

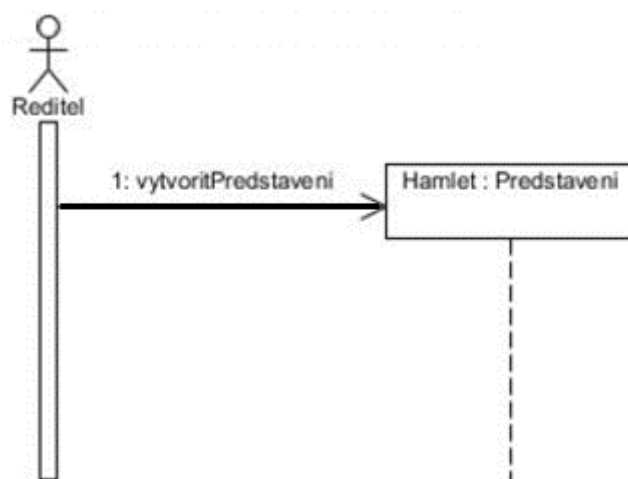
- Používají se v případě, že potřebujeme jednu zprávu poslat vícekrát než jednou.
- označuje se `*[podmínka iterace]`



Příklad – iterace

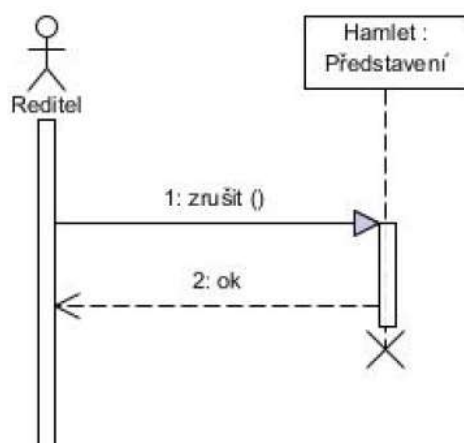
5.2.6 Vytváření objektů

- Objekty mohou vznikat i zanikat i v průběhu scénáře.

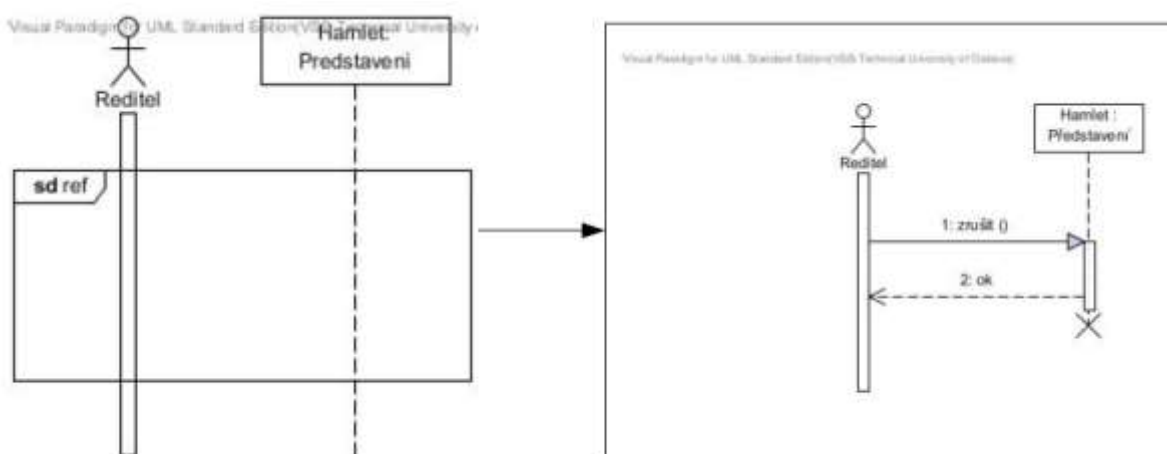


Příklad – vytváření objektu

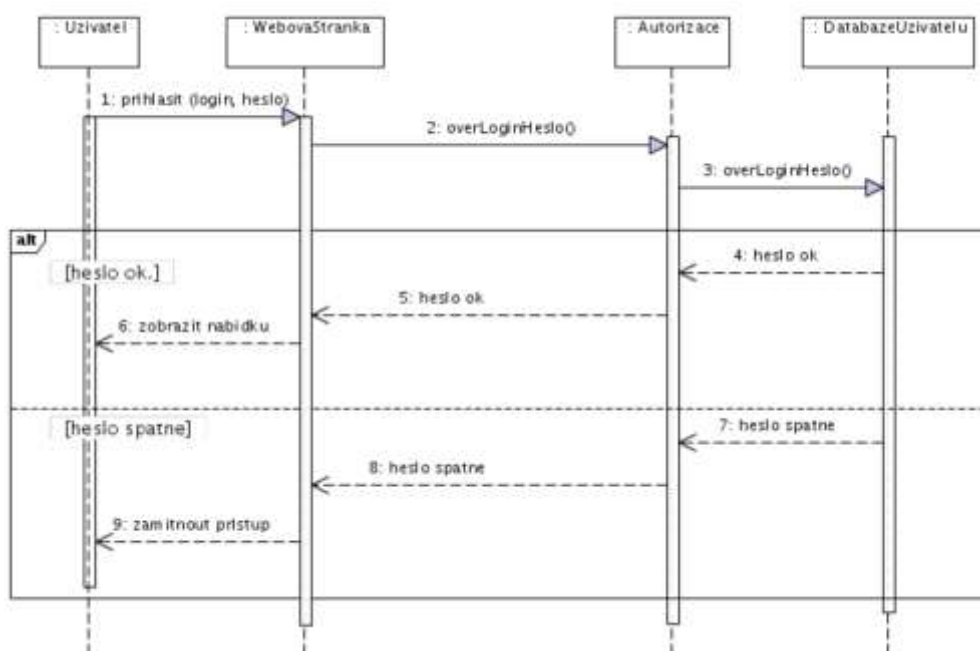
5.2.7 Rušení objektů



5.2.8 Znovupoužití diagramů – reference (odkazy)

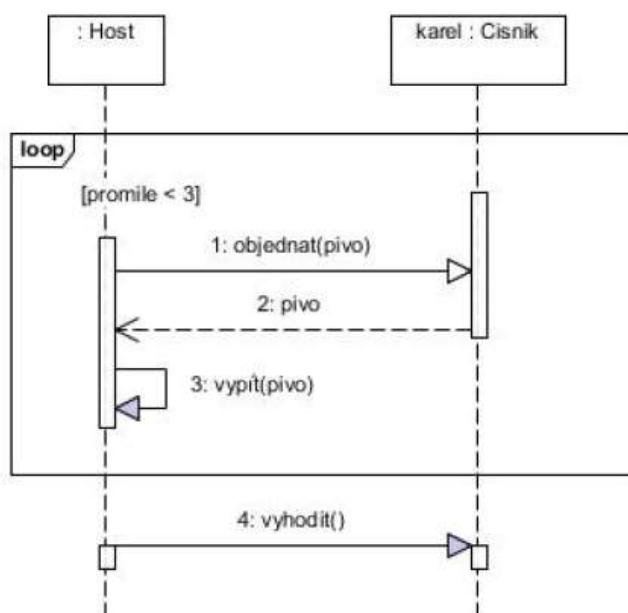


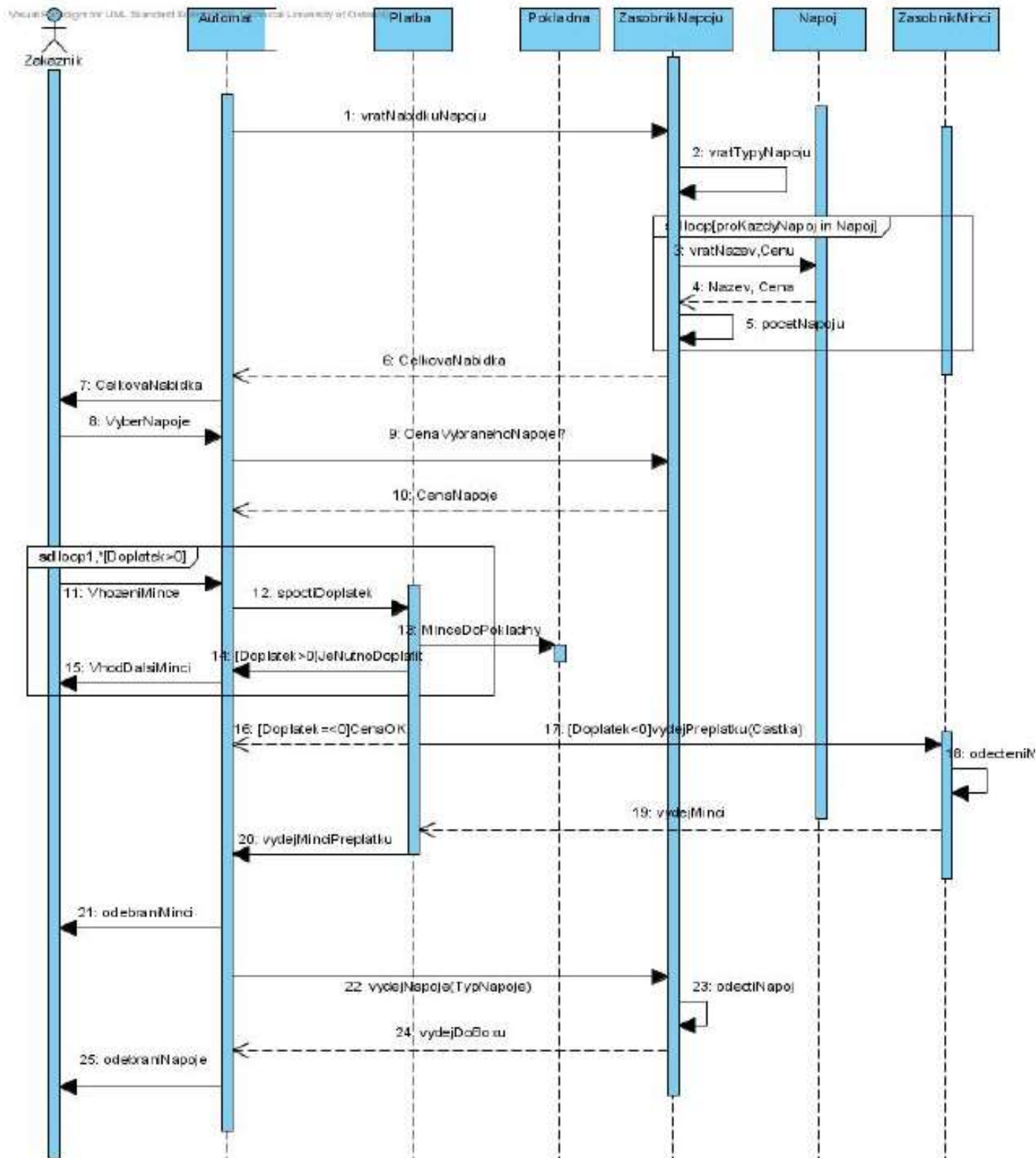
5.2.9 Alternativní scénáře (alternace)

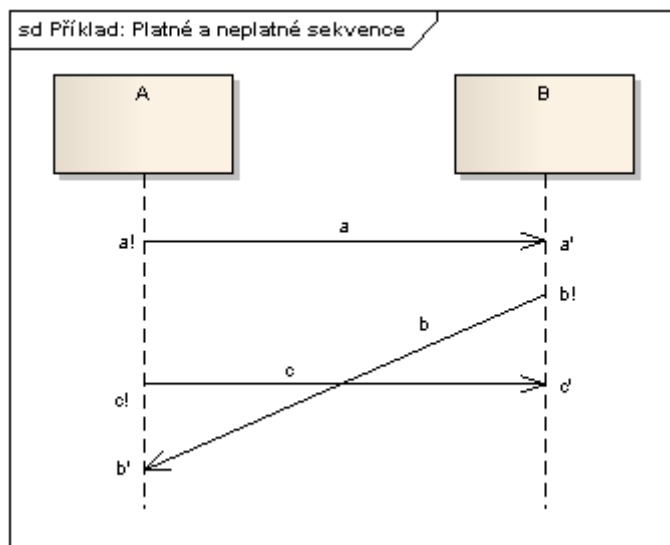


5.2.10 Cyklus

- Slouží pro opakované provedení určité části scénáře, dokud není splněna určitá podmínka.



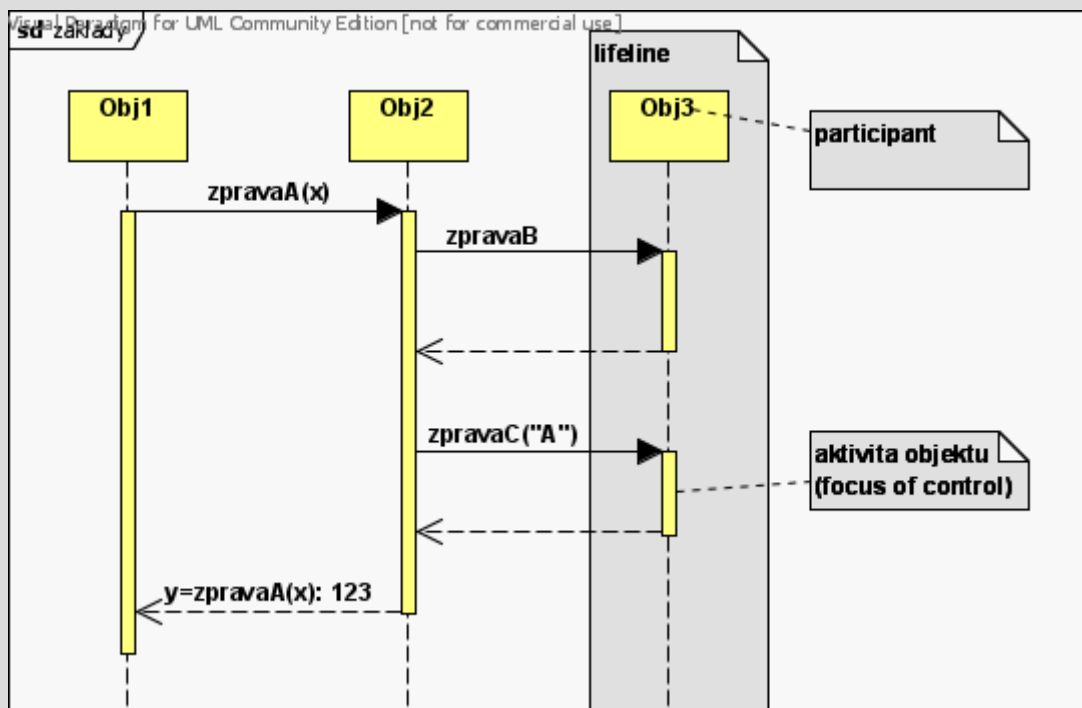




Příklad – platné a neplatné sekvence

Každá sekvence se zapisuje podle

Příklad



Příklad - jednoduchý sekvenční diagram

Lifeline – participant

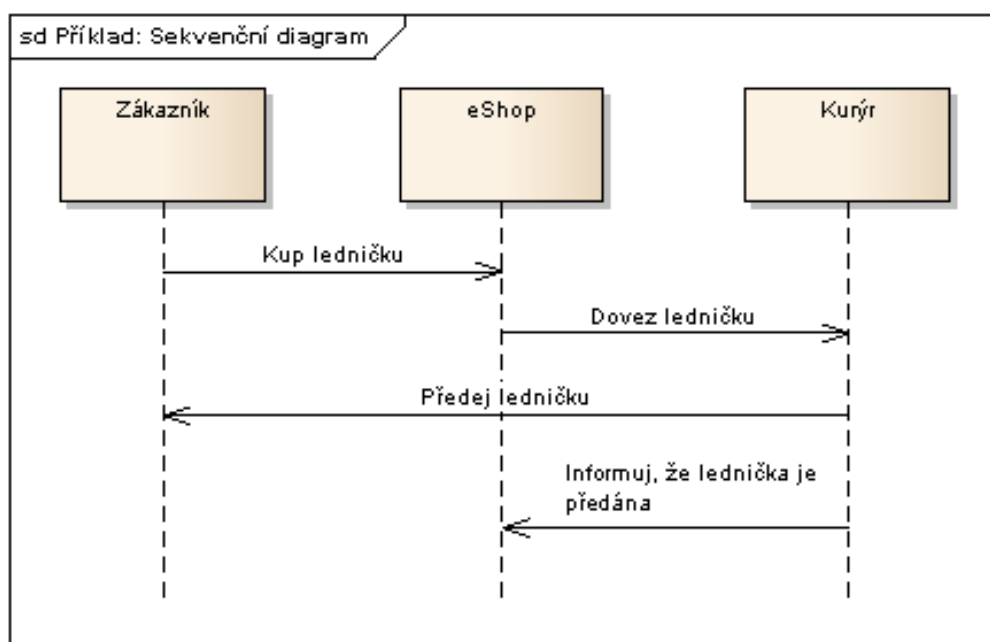
- všechny objekty existovaly už před posláním první zprávy, a dále existují po dokončení sekvenčního diagramu (resp. nevíme, kdy a který objekt zanikne nebo nás to v tuto chvíli nezajímá)

Zprávy

- v našem příkladu máme jen jednoduché obyčejné zprávy: u nich se zpráva zobrazuje plnou čarou a plnou šipkou a v našem příkladu je znázorněno např. ohledně zprávy `zpravaA(x)`: objekt `obj1` posílá (tedy je to sender) zprávu `zpravaA` s jedním argumentem, který je naplněn hodnotou atributu `x`, objektu `obj2` (to je tedy receiver)
- zobrazení návratové zprávy není povinné (vždy si ji lze "domyslet"), ale může přidat na přehlednosti: zobrazuje se šipkou s čárkovanou čarou
- návratová zpráva může, ale nemusí vracet hodnotu: po poslání `zpravaA(x)` a poté, co `obj2` dokončí svou činnost, tak vrátí zpět návratovou hodnotu 123, kterou si `obj1` převezme do svého atributu `y`

V diagramu (Příklad) jsou znázorněny tyto sekvence

1. objekt `obj1` je hned na počátku aktivován (má focus) a poslal zprávu `zpravaA` s parametrem x objektu `obj2`: `obj1` přeruší zpracování (ztratí focus a předá řízení `obj2`) a čeká, až `obj2` odpoví na `zpravaA(x)`
2. `obj2` získává focus a spustí vlastní metodu
3. v jistém bodě zpracování posílá `obj2` zprávu `zpravaB` do `obj3` - `obj2` započne čekat na odpověď od `obj3`
4. `obj3` získává focus (a spouští vlastní metodu)
5. `obj3` dokončí své zpracování, pak vrátí peška (předá řízení zpět) do `obj2`
6. `obj2` pokračuje v práci, a posílá další zprávu do `obj3`
7. až `obj3` dokončí zpracování, tak vrátí řízení zpět do `obj2`
8. `obj2` pokračuje v práci a až skončí, tak vrátí řízení do `obj1`, přitom ale vrátí návratový návratovou hodnotu 123
9. `obj1` si převezme návratovou hodnotu do atributu `y`



5.3 Class diagram vs. Object diagram vs. Object Sequence diagram

- *Class diagram* znázorňuje statickou strukturu systému (vazby mezi objekty)
- *Object diagram* znázorňuje chování instancí tříd v konkr. čase
- *Sekvenční diagram* znázorňuje vzájemné působení mezi aktivní objekty

Kontrola konzistence vizualizačních prvků Class diagramu, Object diagramu a Object Sequence diagramu

- Názvy objektů
- Názvy operací tříd = názvy zpráv (interakcí) mezi objekty.
- *Směrování zpráv*: příjemcem zprávy je objekt, u něž bude spouštěna požadovaná metoda
- Existence operací u třídy, kam je zasílána zpráva v sekvenčním diagramu.
- Existence zprávy v některém sekvenčním diagramu, která odpovídá operaci v class diagramu.

5.4 Shrnutí

- Účelem sekvenčního diagramu je modelování interakcí mezi objekty.
- Vychází z class diagramu a případů užití
- Slouží pro ujasnění, jak objekty fungují a vzájemně spolupracují ke splnění cíle.
- Zaměřuje se na klíčové scénáře, není nutné (ani možné) modelovat celý systém najednou.
- Sekvenční diagramy mají velmi blízko k implementaci.

Spolupráce mezi diagramy

- Diagramy dávají vývojářům dávají komplexní (celistvý) pohled na systém a nastiňují (různé) možnosti strategie tvorby systému.
- Na základě dat *OSD* se zpětně upravuje *class diagram*, a tedy i *object diagram*.

Cvičení

1) Pro příklad operačního systému počítače vytvořte sekvenční diagram představující stisk klávesy v textovém editoru, který způsobí zobrazení tohoto znaku na obrazovce. Předpokládejte interakce mezi těmito objekty:

GUI (grafické uživatelské rozhraní), OS (operační systém), CPU (procesor), Grafická karta, Monitor

2) Vytvořte sekvenční diagram pro praní prádla v pračce. Předpokládejte interakce mezi těmito objekty:

přívod vody, buben, odtok

3)

3a) Vytvořte sekvenční diagram pro nákup nápoje v automatu. Předpokládejte, že se automat skládá z těchto čtyř objektů:

Zákazník, Ovládací panel, Pokladna, Výdejník

3b) Pro případ nápojového automatu uvažujte scénář, v němž zákazník nemá správný obnos.

3c) Pro případ nápojového automatu uvažujte scénář, kdy je požadovaný druh nápoje vyprodán.

Řešení cvičení – viz Moodle

6 Dynamické datové struktury

= též odvozené datové struktury

- Reprezentace tabulek, seznamů, grafů, matic, stromů, ...

Dynamické datové struktury mají širokospektrální využití při řešení různých informatických problémů: např. rychlé třídění, rychlé vyhledávání, překlady výrazů, kompresní algoritmy, optimalizační úlohy, aj.

Dynamické struktury jsou takové datové struktury, u nichž se mění počet prvků v průběhu algoritmu. Naopak statické datové struktury (např. pole) má předem definovaný datový typ i rozměr.

- Seznam (*list*)
- Fronta (*queue*)
- Prioritní fronta (*priority queue*)
- Zásobník (*stack*)
- Strom (*tree*)
- Tabulka (*table*)

6.1 Spojový seznam (*list*)

- Dynamická datová struktura (vzdáleně podobná poli)
 - umožňuje uchovat velké množství hodnot ale jiným způsobem než pole
- Obsahuje jednu a více datových struktur stejného typu (lineárně provázány vzájemnými odkazy pomocí ukazatelů/referencí)
- Neexistují cykly ve vzájemných odkazech (\Leftrightarrow zachování lineárnosti seznamu)
- Datovou strukturu tvoří posloupnost položek
- Každá položka obsahuje odkaz na další položku
- Patří mezi tzv. rekurzivní algoritmy – odkazují na položky stejného typu

Vlastnosti seznamu

- Rychlé přidávání prvků na počátek/konec seznamu ($O(1)$).
- Vhodný pro procházení prvků sekvenčně.
- Nevhodný pro přímý přístup k prvkům.

6.1.1 Základní pojmy

Implementace

- Prostřednictvím pole
- Zřetěžený seznam

Procházení seznamem

- Procházení jedním směrem: Jednosměrný seznam
- Procházení oběma směry: Obousměrný seznam

Hlava (*head*): první prvek seznamu

Ohon (*tail*): poslední prvek seznamu

6.1.2 Typy seznamů:

Jednosměrný seznam (*Single-linked list*)

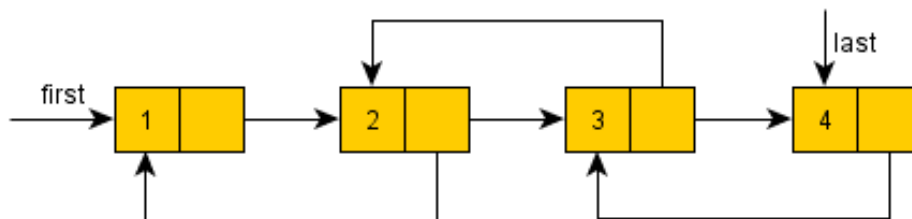
- Každá položka odkazuje na položku následující



Jednocestný spojový seznam. Každý prvek seznamu kromě své hodnoty obsahuje i odkaz (pointer, referenci, ...) na následující prvek v seznamu. Poslední prvek odkazuje „nikam“.

Obousměrný seznam (*Double-linked list*)

- Položky odkazují na následující i předcházející položky

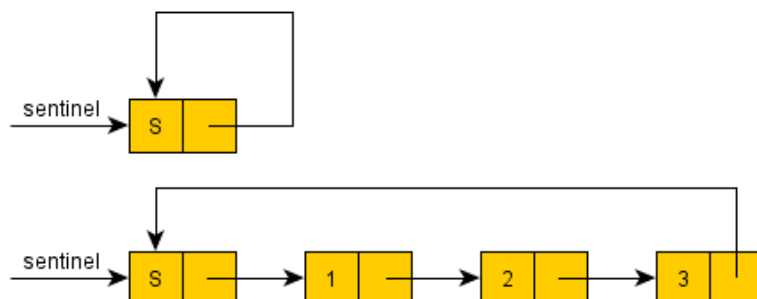


Dvoucestný/dvousměrný spojový seznam. Každý prvek seznamu obsahuje, kromě své hodnoty, odkaz na následující i předchozí prvky seznamu.

Kruhový seznam (*Circular list*)

- Jednoduchý ,trik‘ → zjednodušuje implementaci spojového seznamu
- Hlídka (sentinel)
 - o Speciální uzel, který slouží zároveň jako první a poslední prvek
- Hlava (head) = Ohon (tail)
- o V případě prázdného seznamu ukazuje sám na sebe
- Tímto trikem dojde ke ztotožnění operací vložení na začátek, konec a mezi prvky spojového seznamu.
- Nevýhoda = dodatečné paměťové nároky na objekt hlídky.

Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o kruhový seznam.



Jednosměrný kruhový seznam. Poslední prvek seznamu odkazuje opět na začátek.

6.1.3 Výhody a nevýhody

Výhody:

- kapacita je teoreticky neomezená
- velikost obsazené paměti je přímo závislá jen na počtu prvků, není zde žádné plýtvání
- rychlost přidávání i odebírání prvků je vždy stejně vysoká

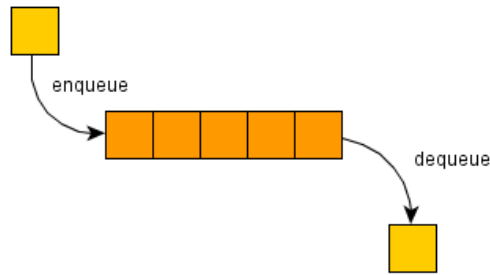
Nevýhody:

- pomalý přístup k prvkům na zadaném indexu i (random access)
- uložené hodnoty nejsou v paměti uspořádány za sebou a změna struktury seznamu může způsobit fragmentaci volného místa v paměti
- pomalejší procházení (při každém posunu je nutná dereference ukazatele a skok na místo v paměti)
- stejné množství dat zabírá více paměti než stejné prvky uložené v poli (kvůli ukazatelům navíc)

6.2 Fronta (*Queue*)

Fronta = slouží k ukládání dat, které vystupují ze struktury ve stejném pořadí, jako vstupují.

- Odborně se nazývá FIFO – *First In First Out* (někdy také jako tunel).
 - Lze si ji představit jako frontu lidí, kteří postupují k pokladně. Jak se řadí do fronty, tak i odcházejí od pokladny.
=> pracoval mohu pouze s prvkem, který je na čele fronty
- Jeden ze základních datových typů (slouží k ukládání a výběru dat takovým způsobem, aby prvek, který byl uložen jako první, byl také jako první vybrán).



OBRÁZEK 6-1: FIFO (NECKÁŘ, 2015)

Implementace

- Polem
- Spojovým seznamem

Složitost operací $O(1)$.

Čelo fronty: Prvek na první pozici, do fronty přidán jako první.

Konec fronty: Prvek na poslední pozici ve frontě, do fronty přidám jako poslední.

6.2.1 Operace

`addLast` (*enqueue*) – Vloží prvek do fronty.

`deleteFirst` (*poll*, *dequeue*) – Získá a odstraní čelo fronty.

`getFirst` (*peek*) – Získá čelo fronty.

`isEmpty` – Dotaz na prázdnotu fronty.

`size` – Vrátí počet obsažených prvků.

6.2.2 Využití

Fronta má v ICT rozšířené využití, zde jsou některé příklady:

- Synchronizační primitivum
- Operátor roura (pipe) - komunikace mezi procesy v operačních systémech
- Kruhový buffer – vyrovnávací paměť pro datové toky
- Heapsort

Příklad v programovacím jazyce JAVA

```
/**
 * Fronta - implementovana jako spojovy seznam
 */
public class Queue {
    private Node first;
    private Node last;
    private int size;
    public Queue() {
        this.size = 0;
    }

    /**
     * Prida na konec fronty prvek - asymptoticka slozitimost O(1)
     * @param i prvek k vlozeni
     */
    public void addLast(int i) {
        Node n = new Node(i);
        if(getSize() == 0) {
            this.first = n;
            this.last = n;
        } else {
            this.last.next = n;
            this.last = n;
        }
        size++;
    }

    /**
     * Odebere z fronty prvni prvek - asymptoticka slozitimost O(1)
     * @return prvni prvek
     */
    public int deteteFirst() {
        if(getSize() == 0) throw new IllegalStateException("Fronta je prazdna");
        int value = first.value;
        first = first.next;
        size--;
        return value;
    }

    /**
     * Vraci prvni prvek fronty
     * @return prvni prvek
     */
    public int getFirst() {
        if(getSize() == 0) throw new IllegalStateException("Fronta je prazdna");
        return first.value;
    }

    /**
     * Getter na delku
     * @return delka fronty
     */
    public int getSize() {
        return size;
    }

    /**
     * Dotaz pra prazdnost
     * @return true, pokud je fronta prazdna
     */
    public boolean isEmpty() {
        return this.size == 0;
    }
}
```

```

    * Klasická toString metoda, vrací textovou reprezentaci objektu
    * @return textová reprezentace objektu
    */
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        Node curr = first;
        for(int i = 0; i < this.size; i++) {
            builder.append(curr.value).append(" ");
            curr = curr.next;
        }
        return builder.toString();
    }

    /**
     * Vnitřní reprezentace prvku
     */
    private class Node {
        private int value;
        private Node next;
        private Node(int value) {
            this.value = value;
        }
    }
}

```

ALGORITMUS 9: FIFO (ALGORTIMY.NET, 2019)

6.3 Prioritní fronta (*Priority queue*)

Někdy též známa jako *fronta s předbíráním*

- Tzv. zobecněná datová struktura – kombinuje funkce zásobníku a fronty
- Pracuje s prvky s rozdílnými váhami
- Reprezentuje strukturu uspořádaných dvojic $[a, b]$, kde a ...priorita (klíč) a b ...položka. Tyto dvojice jsou přeuspořádány podle priority.
- Hodnota priority je určena zhodnocovací funkcí – udává význam prvku
- Zásadní rozdíl oproti frontě = prvek s vyšší hodnotou priority předbíhá prvky s nižší hodnotou priority.

Princip

1. Prvky ukládány v pořadí, v jakém jsou přidávány na konci fronty.
2. Prvky jsou přeuspořádány podle priority.
3. Prvky postupně odebírány na základě hodnoty priority.

Implementace

- Lineární seznam
- Binární vyhledávací strom (BST – *binary search tree*)
- Halda

Velké rozdíly ve výkonnostních charakteristikách!

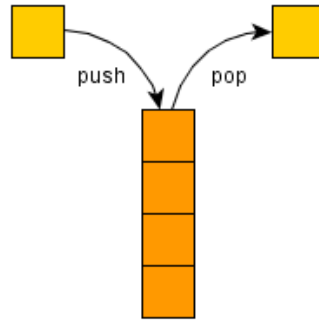
6.3.1 Operace nad prioritní frontou

- Vytvoření fronty
- Přidání položky: `push`
- Smazání položky s největší prioritou: `pop`
- Změna priority položky.
- Výmaz libovolné položky
- Smazání fronty.

6.4 Zásobník (*Stack*)

Zásobník = naopak, první data vystupují ze zásobníku jako poslední a naopak.

- Odborně LIFO – *Last In First Out*.
- Zároveň se nazývá i jako *studna* nebo *díra*, protože když něco uložíme do díry jako první, můžeme to vyndat až jako poslední (resp. až odstraníme všechno ostatní, co následně na první prvek položíme).
- Využívá zvláště pro dočasné ukládání dat v průběhu výpočtu.
- Poslední vložený prvek jde na výstup jako první, předposlední jako druhý a tak dále.



OBRÁZEK 6-2: LIFO (NECKÁŘ, 2015)

Implementace zásobníku

- Spojovým seznamem
- Polem

Složitost operací $O(1)$

Dno zásobníku: Nejspodnější prvek zásobníku, přidán jako první

Vrchol zásobníku: Nejvrchnější prvek zásobníku, přidán jako poslední

6.4.1 Základní operace

`push` - vloží prvek na vrch zásobníku

`pop` - odstraní vrchol zásobníku

`top` - dotaz na vrchol zásobníku

`isEmpty` - dotaz na prázdnotu zásobníku (size - dotaz na velikost zásobníku)

6.4.2 Stack overflow! (přetečení zásobníku)

Problém rekurze (kapitola 2.3 Rekurze, str. 51. in (Mazuch, 2019) 2 Algoritmy)

pokus uložení do zásobníku volání více dat, než kolik se tam vejde.

Velikost zásobníku = dána při startu programu v závislosti na několika indicích
(architektura systému, překladač, množství volné paměti, ...)

Pokud se program pokusí posunout vrchol zásobníku mimo určenou paměť, mluvíme o přetečení zásobníku. To má obvykle za následek pád programu.

Důvodem přetečení je nejčastěji jedna z následujících programátorských chyb:

- Nekonečná rekurze
- Příliš objemné proměnné na zásobníku

6.4.3 Využití

Zásobník se v IT používá zejména pro:

- ukládání stavu algoritmů a programů
- využit v Tarjanově algoritmu
- prohledávání do hloubky (DFS, *depth-first search*)
- implicitně ve všech rekurzivních algoritmech.

Na zásobníkové architektuře jsou postaveny virtuální stroje pro jazyky Java a Lisp.

6.5 Grafy, stromy

Mezi další dynamické datové struktury patří grafy a stromy, kterým se – s ohledem k jejich rozlehlosti – budeme věnovat ve speciální kapitole.

7 Základy teorie grafů

<http://www.graphtheorysoftware.com> – Graph Theory software

7.1 Pojem grafu

Teorie grafů = jedna z nejmladších disciplín matematiky (konkr. diskrétní matematiky)

Existuje mnoho různých typů grafů, např.: statistické grafy (sloupcový, koláčový, ...), grafy funkcí, aj.

Graf je určitým zjednodušením skutečnosti (studovaný problém lze znázornit pomocí bodů a linií, které je spojují) → popisují vlastnosti daného problému.

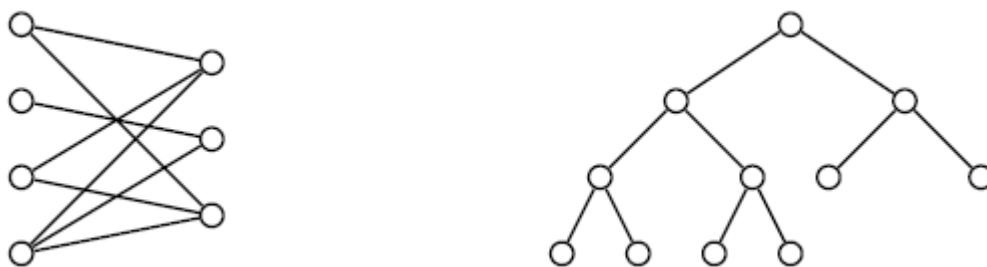
Grafem rozumíme datovou strukturu, která popisuje vztahy mezi objekty.

Grafy z oblasti teorie grafů mají široké využití v mnoha oblastech, např. úlohy o dopravním spojení, logistické problémy, optimální spojení, propustnost sítě, přenos energie, komprese dat, ...

7.1.1 Části grafu

Každý graf obsahuje:

- Body = vrcholy grafu
- Linie = hrany grafu



– OBRÁZEK 7-1: UKÁZKY JEDNODUCHÝCH GRAFŮ (KOVÁŘ, 2012)

7.1.1.1 Vrchol grafu (uzel)

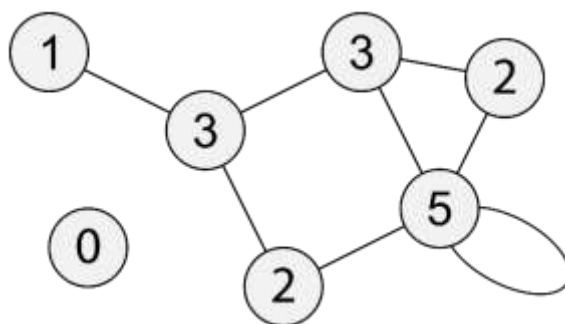
Vrchol též uzel je jedním z prvků množiny definující graf. Graficky se znázorňuje jako bod či malý kruh (např. s názvem vrcholu uvnitř). Z vrcholu mohou vést hrany.

Izolovaný vrchol

Izolovaným vrcholem je nazýván vrchol, který neinciduje žádná hrana grafu. Viz graf na Obrázek 7-2, vrchol 0.

Stupeň vrcholu

Stupeň vrcholu (též valence vrcholu) označuje počet hran, které do daného vrcholu zasahují. Stupeň vrcholu V se značí $\deg(V)$.



OBRÁZEK 7-2: PŘÍKLAD GRAFU S IZOLOVANÝM VRCHOLEM (WIKIPEDIE, 2018)

Pro izolovaný vrchol V platí: $\deg(V) = 0$.

7.1.1.2 Hrana

Hranou rozumíme uspořádanou nebo neuspořádanou dvojici vrcholů grafu. Graficky se znázorňuje jako linie (přímka / oblouk) mezi vrcholy, které váže.

Typy hran

- orientovaná hrana – uspořádaná dvojice vrcholů; má vyznačen směr průchodu, hranou lze procházet pouze ve vyznačeném směru.

Orientovaný graf je takový graf, jehož všechny hrany jsou orientované.

- neorientovaná hrana – neuspořádaná dvojice; bez vyznačení směru průchodu, hranou lze procházet oběma směry

Neorientovaný graf je takový graf, jehož všechny hrany jsou neorientované.

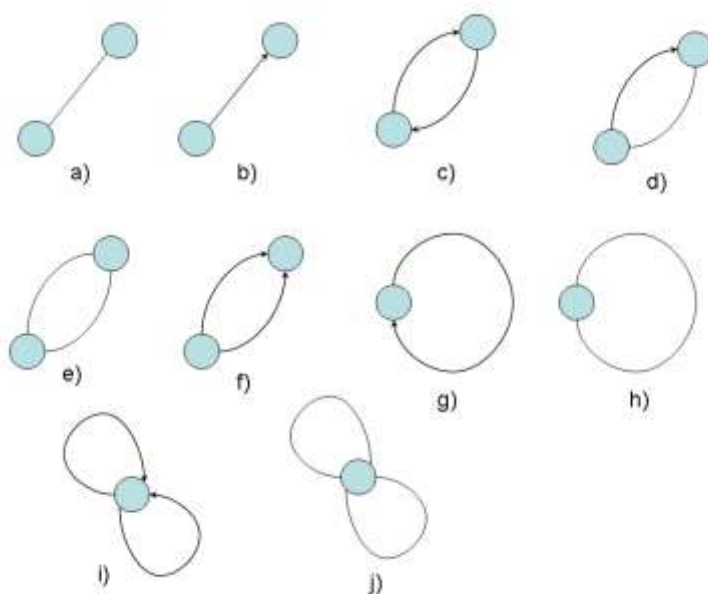
- násobné hrany – více hran spojujících stejné vrcholy
- most – hrana, jejímž odebráním se zvýší počet komponent grafu
- smyčka – hrana vedoucí z vrcholu do něj samotného

Hraně je vhodné přidružit *ohodnocení*. Ohodnocení hrany vyjadřuje kvalitu nebo kvantitu vztahu mezi dvěma vrcholy (např. vzdálenost, průchodnost apod.).

Zařazení grafů

Výskyt různých typů hran má vliv na označení grafu:

- jednoduchý (obyčejný) graf – neobsahuje smyčky ani násobné hrany
- multigraf – obsahuje násobné hrany
- prostý graf – neobsahuje násobné hrany
- pseudograf – obsahuje smyčky



OBRÁZEK 7-3: HRANY GRAFU (WIKIPEDIE, 2018)

- a) neorientovaná hrana,
- b) přímá orientovaná hrana,
- c) a d) násobné hrany,
- e) a f) rovnoběžné hrany,
- g) orientovaná smyčka,
- h) neorientovaná smyčka,
- i) a j) násobné hrany se smyčkou

7.1.2 Využití

7.1.2.1 Hledání nejkratší cesty

Představme si, že chceme zjistit nejkratší cestu mezi čtyřmi (pro jednoduchost) vybranými městy.

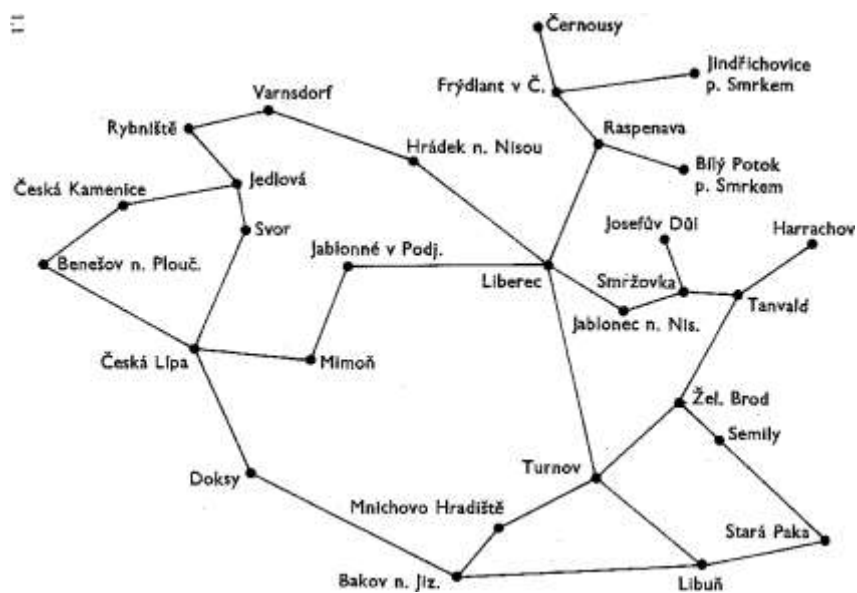
Jediná vlastnost, která nás bude zajímat, je vzdálenost mezi městy

- (vzdálenost (km) = délka cesty, kterou bychom museli ujet po silnici)
- vrcholy tedy použijeme města; hrany pak budou popisovat, že mezi městy existuje cesta a jak je dlouhá

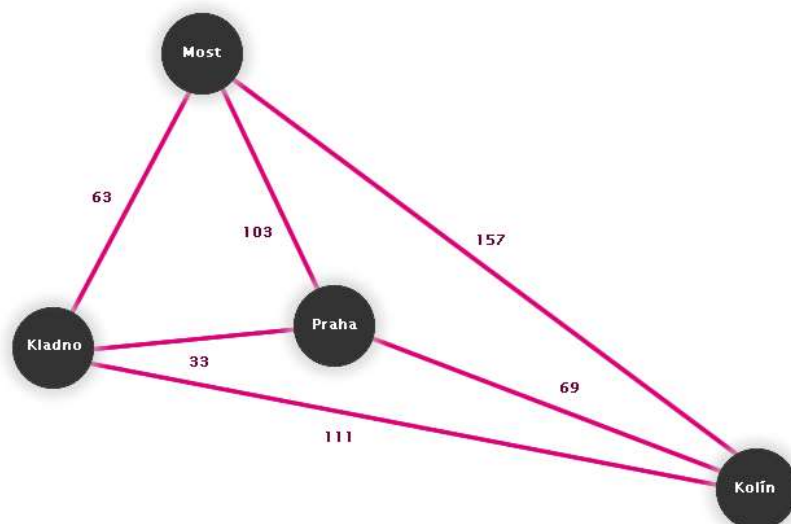
Zanedbáváme další (v tuto chvíli zbytečné informace):

- z kolika různých silnic se cesta skládá
- jakého jsou tvaru
- zeměpisná poloha měst (sever, jih...).

Jediné informace, které pak algoritmus využije, budou čísla na hranách spojujících města (vzdálenosti mezi městy).



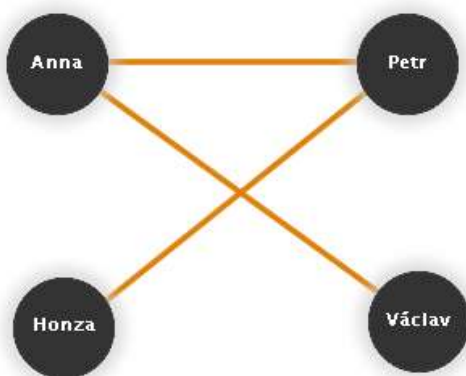
OBRÁZEK 4: VYUŽITÍ GRAFŮ V HLEDÁNÍ NEJKRATŠÍ CESTY (HOUSKA, 1987)



OBRÁZEK 7-5: ZNÁZORNĚNÍ VZDÁLENOSTÍ MEZI MĚSTY POMOCÍ GRAFU (JIROVSKÝ, 2008)

7.1.2.2 Popis rozlišných situací

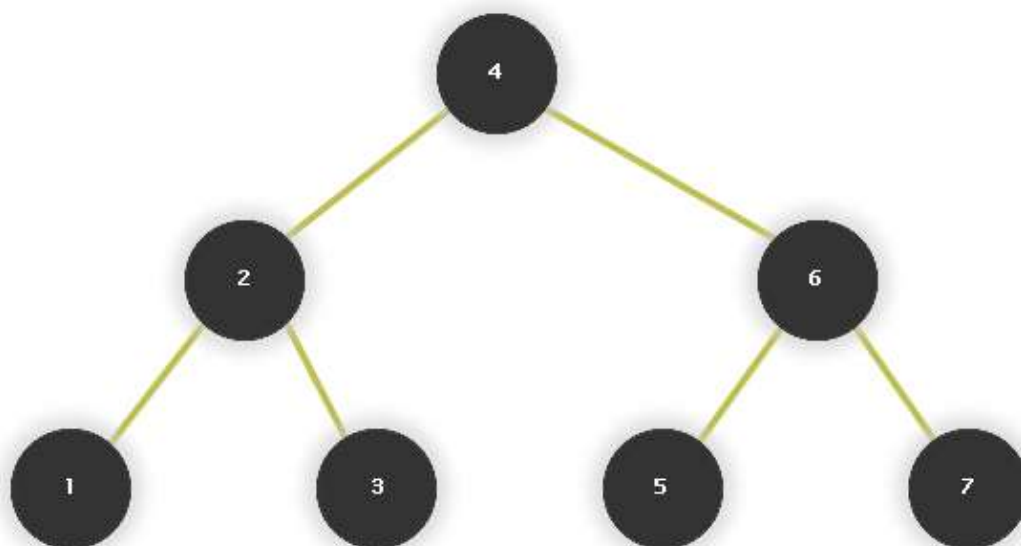
Nechť existují čtyři kamarádi: Anna, Petr, Jan a Václav. Pomocí grafu znázorníme, kdo koho zná (kdo s kým kamarádí). Znalostí grafů lze pak jednodušeji dokazovat různé důsledky (sezení u stolu).



OBRÁZEK 7-6: KAMARÁDI (JIROVSKÝ, 2008)

7.1.2.3 Stromy

Stromy jsou zvláštním typem grafů často používaným v informatice (více *dále*). Pomocí vhodně navrženého stromu lze např. jednodušeji vyhledávat či třídit čísla (heap sort).



OBRÁZEK 7-7: STROM (HALDA) (JIROVSKÝ, 2008)

Algebraická definice grafu:

Graf \mathbf{G} (též *jednoduchý* graf nebo *obyčejný* graf) je uspořádaná dvojice $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, kde \mathbf{V} je neprázdná množina *vrcholů* a \mathbf{E} je množina *hran* – množina (některých) dvouprvkových podmnožin množiny \mathbf{V} .

Říkáme, že vrcholy u a v jsou *sousední* v grafu G , jestliže hrana $uv \in E(G)$. Pokud množina E hranu uv neobsahuje, tak vrcholy u, v jsou *nesousední*. Naopak pracujeme-li s hranou uv , tak vrcholy u a v nazýváme *koncové* vrcholy hrany uv .

Značení:

V množina vrcholů

$|V|$ velikost množiny vrcholů, tedy počet vrcholů (číslo)

E množina hran

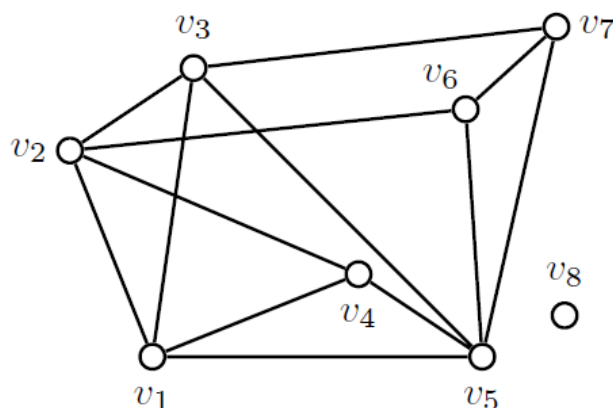
$|E|$ velikost množiny hran, tedy počet hran (číslo)

7.1.3 Znázornění grafu

Graf lze nakreslit, resp. zadat vhodným obrázkem (*viz níže*).

Nechť $G_1 = (V, E)$, kde množina vrcholů je $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$ a množina hran (dvouprvkových podmnožin množiny V) je

$E = \{v_1v_2, v_1v_3, v_1v_4, v_1v_5, v_2v_3, v_2v_4, v_2v_6, v_3v_5, v_3v_7, v_4v_5, v_5v_6, v_5v_7, v_6v_7\}$.



OBRÁZEK 7-8: NAKRESLENÍ GRAFU G_1 (KOVÁŘ, 2012)

Vysvětlení:

- Vrcholy grafu $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$ jsou znázorněny jako:
 - Body v grafickém znázornění
 - Jednotlivé prvky množiny V . Lze též zapsat jako $V = \{v_1, v_2, v_3, \dots\}$
 $|V| = 8$
- Hrany grafu $E = \{v_1v_2, v_1v_3, v_1v_4, v_1v_5, v_2v_3, v_2v_4, \dots\}$ jsou znázorněny jako:
 - Linie v grafu, např.: prvek množiny E , zapsaný jako v_1v_3 spojuje vrchol v_1 s vrcholem v_3 . $|E| = 13$

Aby náčrt grafu byl korektní a přehledný, je praktické požadovat:

- aby každý vrchol byl zakreslen do jiného bodu,
- aby každá hrana procházela jen jejími dvěma koncovými vrcholy (a žádným jiným vrcholem),
- aby se dvě různé hrany v nakreslení protínaly nejvýše jednou,
- aby žádná hrana neprotínala sama sebe,
- aby se v nakreslení křížilo co možná nejméně hran.

Cvičení:

- 1) Jsou vrcholy v_1 a v_5 v grafu G_1 sousední?
- 2) Jsou vrcholy v_4 a v_7 v grafu G_1 sousední?
- 3) Urči všechny nesousední vrcholy vrcholu v_8 v grafu G_1
- 4) Devět kamarádů si na Vánoce dalo dárky. Každý dal dárky třem svým kamarádům. Ukažte (znázorněte grafem), že není možné, aby každý dostal dárky právě od těch tří kamarádů, kterým dárky sám dal.

Grafy, kde jsou každé dva vrcholy spojené nanejvýš jednou hranou. V případě, že jsou dva vrcholy spojené větším množstvím hran, hovoříme o takzvaných násobných hranách. Takovým grafům pak říkáme multigrafy.



OBRÁZEK 7-9: ELEMENTÁRNÍ MULTIGRAF (JIROVSKÝ, 2008)

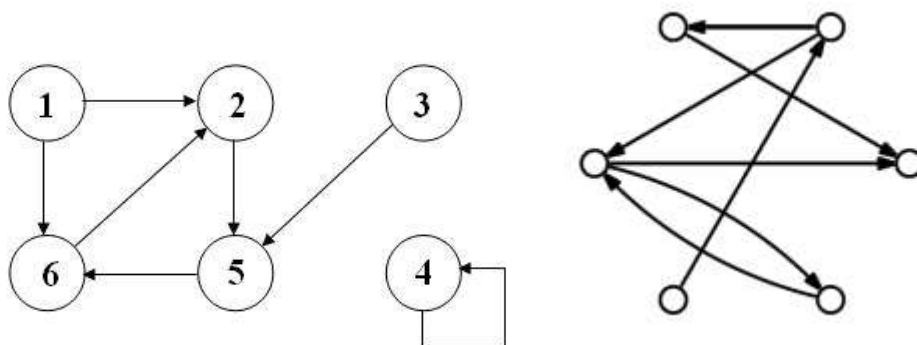
Grafy též mohou obsahovat smyčky



OBRÁZEK 7-10: GRAF SE SMYČKOU (JIROVSKÝ, 2008)

7.1.4 Orientovaný graf a neorientovaný graf

Orientovaný graf



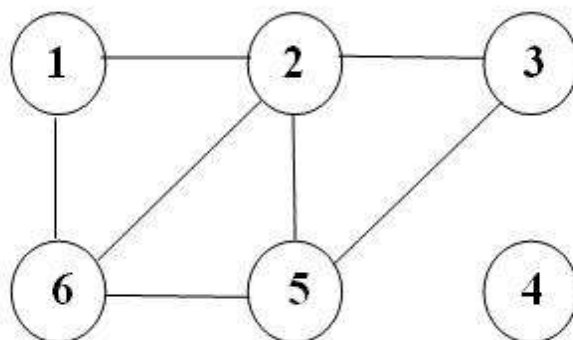
OBRÁZEK 7-11: ORIENTOVANÝ GRAF (JANČAŘÍK, 2014), (KOVÁŘ, 2012)

Orientovaným grafem rozumíme uspořádanou dvojici $G = (V, E)$, kde V je množina *vrcholů* a množina *orientovaných hran* je $E \subseteq V \times V$.

Orientovaná hrana uv pak není dvouprvková podmnožina $\{u, v\}$, ale uspořádaná dvojice (u, v) dvou prvků $u, v \in V$. Vrchol u je počáteční a vrchol v koncový \overrightarrow{uv} .

V nakreslení znázorníme orientované hrany šipkami

Neorientovaný graf



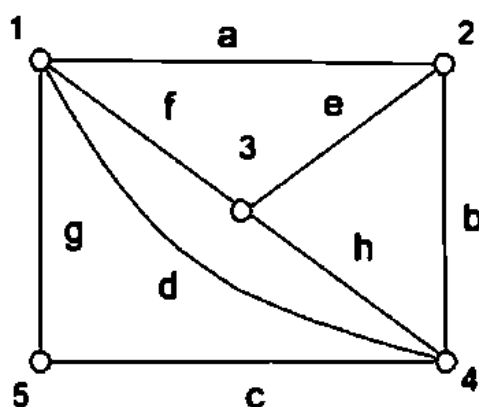
OBRÁZEK 7-12: NEORIENTOVANÝ GRAF (JANČAŘÍK, 2014)

- Obsahuje vrcholy a hrany.
- Hrany jsou neorientované dvojice vrcholů.

Uspořádaná trojice disjunktních množin $G = \langle V, E, \rho \rangle$, kde V – uzly, E – hrany a ρ – incidenci grafu G .

Incidence $\rho: E \rightarrow \{[V_1, V_2]\}$, kde $V_1, V_2 \in V$ přiřazuje každé hraně E grafu G neprázdnou množinu dvojic uzlů z množiny V .

Příklad:



OBRÁZEK 13: GRAF $G = (V; E)$

Množina vrcholů: $V = \{1, 2, 3, 4, 5\}$

Množina hran: $E = \{a, b, c, d, e, f, g, h\}$

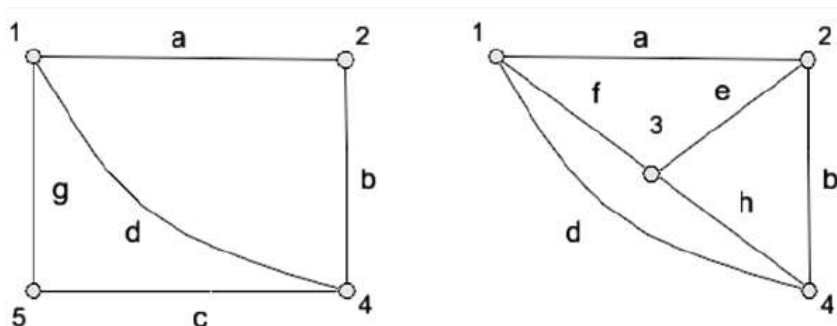
Incidence:

- $\rho(a) = \{1, 2\}$
- $\rho(b) = \{2, 4\}$
- $\rho(c) = \{4, 5\}$
- $\rho(d) = \{1, 4\}$
- $\rho(e) = \{2, 3\}$
- $\rho(f) = \{1, 3\}$
- $\rho(g) = \{1, 5\}$
- $\rho(h) = \{3, 4\}$

7.1.5 Podgraf

Podgraf $G' \subset G$

$G' = (V', E', \rho')$ je podgrafem $G = (V, E, \rho)$ právě tehdy, když platí $E' \subset E$ a $\rho' \subset \rho$ a pro každé $e \in E'$ je $\rho'(v) \subset \rho(v)$



OBRÁZEK 14: PODGRAF G' (VLEVO) A PODGRAF G'' (VPRAVO)

Poznámka: Podgrafy G' a G'' na obrázku Obrázek 14 na straně 136 jsou podgrafy grafu G na Obrázek 13 na straně 135.

7.1.6 Základní třídy grafů

Kromě algebraického popisu grafu (viz str. 132) nebo obrázku lze graf jednoznačně zadat popisem vlastností, nebo jen jeho jménem. Řada struktur se vyskytuje natolik často, že se ustálilo jméno takového grafu nebo celé skupiny grafů. (Jirovský, 2008)

Graf, který obsahuje jediný vrchol (a žádnou hranu), se nazývá *triviální graf* a lze jej značit jako K_1 .



OBRÁZEK 7-15: TRIVIÁLNÍ KOMPLETNÍ GRAF K_1 (KOVÁŘ, 2012)

7.1.6.1 Kompletní graf

Jestliže graf na daném počtu vrcholů $n \in \mathbb{N}$ obsahuje všechny možné hrany, říkáme, že je *kompletní*.

Graf na n vrcholech, kde $n \in \mathbb{N}$, který obsahuje všech $\binom{n}{2}$ hran se nazývá *úplný* nebo též *kompletní* a značíme jej K_n .

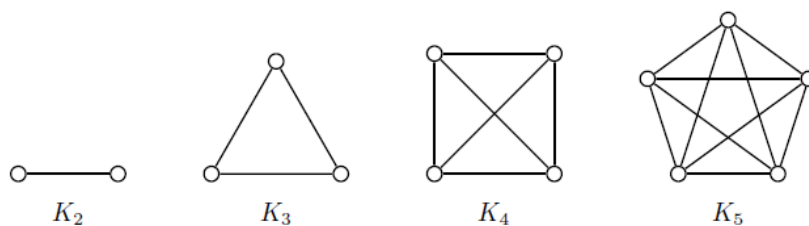
Počet hran je kombinací 2 třídy z n -prvků bez opakování.

$$c_2(n) = \binom{n}{2} = \frac{n!}{2 \cdot (n-2)!} = \frac{n \cdot (n-1) \cdot \cancel{(n-2)!}}{2 \cdot \cancel{(n-2)!}} = \frac{n(n-1)}{2}$$

Algebraicky lze kompletní (úplný) graf K_n popsat následujícím způsobem:

$$K_n = (V, E): \quad V = \{1, 2, \dots, n\}, \quad E = \{ij : i, j = 1, 2, 3, \dots, n \wedge i \neq j\}$$

Příklady náčrtů kompletních grafů pro $n = \{2, 3, 4, 5\}$:



OBRÁZEK 7-16: KOMPLETNÍ GRAFY (KOVÁŘ, 2012)

7.1.6.2 Cyklus (kružnice)

Graf, který má n vrcholů spojených postupně n hranami „dokola“, nazýváme cyklus, nebo též graf cyklický. Graf, který cyklus neobsahuje nazýváme grafem acyklickým.

Graf na n vrcholech (kde $n \geq 3$), kde jsou spojeny po řadě n hranami tak, že každý vrchol je spojen s následujícím vrcholem a poslední vrchol je navíc spojen s prvním vrcholem, se nazývá cyklus na n vrcholech a značí se C_n . Číslo n je délka cyklu C_n .

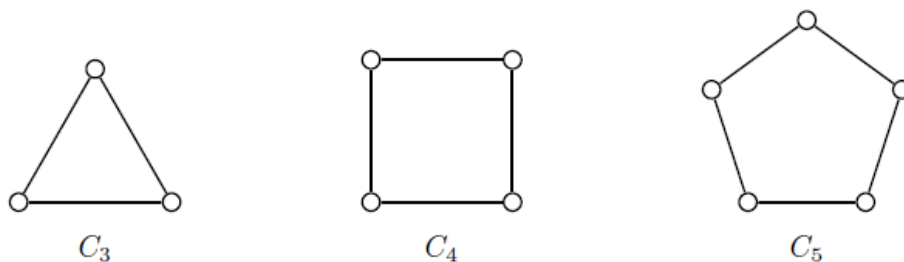
Algebraicky lze cyklus C_n popsat následujícím způsobem:

$$C_n = (V, E): \quad V = \{1, 2, \dots, n\}, \quad E = \{i(i+1) : i = 1, 2, 3, \dots, n-1\} \cup \{1n\}, \quad n \geq 3$$

Zápis $i(i+1)$ je zkrácený zápis množiny $\{i, i+1\}$.

Cyklus C_3 označujeme též jako *trojúhelník*. Cyklus C_4 označujeme též jako *čtverec*.

Příklady náčrtů cyklů:



OBRÁZEK 7-17: CYKLY DÉLKY 3, 4 A 5 (JIROVSKÝ, 2008)

7.2 Reprezentace grafu

Graf lze reprezentovat mnoha způsoby: náčrtem, maticemi, spojovým seznamem.

Nelze říct jaká reprezentace je nejlepší (každá má své využití). Nejčastěji je graf reprezentován pomocí:

- Matice sousednosti

- Matice incidence⁷

7.2.1 Matice sousednosti

Využívá se u:

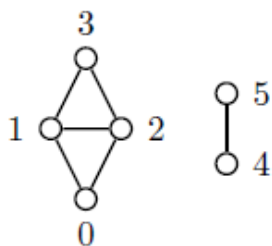
- grafů s velkým počtem hran a
- grafů, které se v průběhu algoritmu často mění.

Matice sousednosti grafu G je čtvercová matice $A = (a_{ij})$, řádu n , ve které:

- prvek $(a_{ij}) = 1$ právě tehdy, když jsou vrcholy i a j sousední.
- prvek $(a_{ij}) = 0$ právě tehdy, když jsou vrcholy i a j nesousední.

Formální definice:

$$a_{ij} = \begin{cases} 1 & \text{je-li } v_i v_j \in E(G) \\ 0 & \text{jinak.} \end{cases}$$

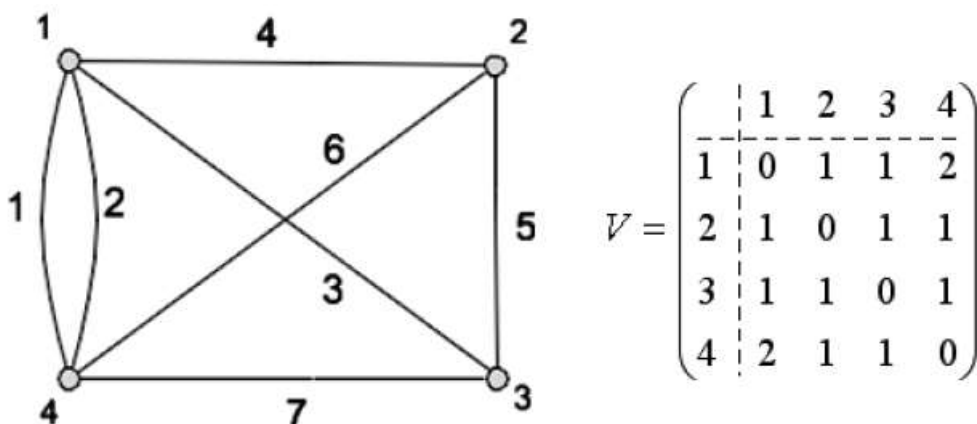


$$A = \begin{matrix} & v \backslash v & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{matrix}$$

OBRÁZEK 7-18: GRAF G SE ŠESTI VRCHOLY $0,1,2,\dots,5$
(Kovář, 2012)

OBRÁZEK 7-19: MATICE SOUSEDNOSTI GRAFU G

⁷ Vzájemná poloha dvou geometrických útvarů majících společnou část. www.slovník-cizich-slov.abz.cz



7.2.2 Matice incidence

Využívá se u:

- Grafů s menším počtem hran a
- Grafů, u kterých se v průběhu algoritmu nemění počet hran.

Incidenční matice B grafu G je matice obdélníková s n řádky a $m = |E(G)|$ sloupce.

Pro \forall – vrchol grafu G platí, že mu odpovídá jeden řádek matice B

- hranu grafu G platí, že ji odpovídá jeden sloupec matice B .

Formální definice:

Nechť $G = (V, E, \rho)$ je neorientovaný graf, kde

množina vrcholů je $V = \{v_1, v_2, \dots, v_n\}$,

množina hran je $E = \{e_1, e_2, \dots, e_m\}$ a

incidence je $\rho: E \rightarrow \{[V_1, V_2]\}$, kde $V_1, V_2 \in V$ přiřazuje každé hraně E grafu G neprázdnou množinu dvojic uzlů z množiny V . (*lidsky: ρ označuje dvojici vrcholů, které jsou danou hranou spojeny.*)

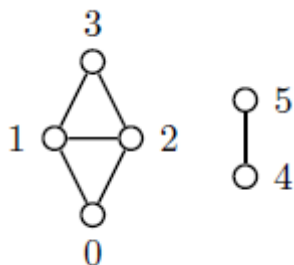
Maticí incidence grafu G nazýváme obdélníkovou matici $B = (b_{ij})$, ve které:

- prvek $(b_{ij}) = 1$ právě tehdy, když hrana e_k inciduje s uzlem v_i .

- prvek $(b_{ij}) = 0$ v opačném případě.

Řádky matice B tvoří vrcholy (uzly) grafu G .

Sloupce matice B tvoří incidence ρ grafu G .

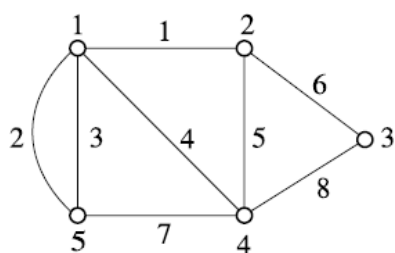


$$B = \begin{matrix} v \backslash e & 01 & 02 & 12 & 13 & 23 & 45 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{matrix}$$

OBRÁZEK 7-20: GRAF G SE ŠESTI VRCHOLY $0, 1, 2, \dots, 5$

OBRÁZEK 7-21: MATICE INCIDENCE GRAFU G

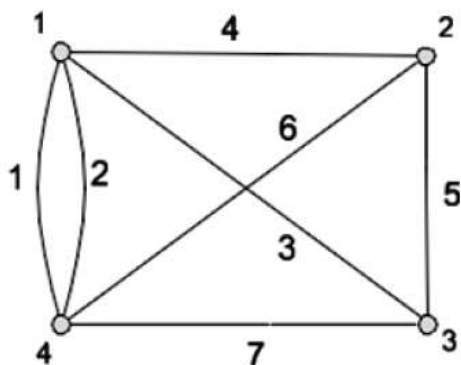
(Kovář, 2012)



$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{matrix}$$

$$V = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \end{pmatrix} \end{matrix}$$

OBRÁZEK 7-22: MATICE INCIDENCE (A) A MATICE SOUSEDNOSTI (V) NEORIENTOVANÉHO GRAFU (KOLÁŘ, 2000)



$$A = \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{pmatrix}$$

7.3 Prohledávání grafu (Kovář, 2012)

Ukážeme algoritmus, který umožní rychle ověřit, zda daný graf (například graf, který odpovídá reálné úloze) je souvislý nebo není. Algoritmus bude obecný, drobnou modifikací dostaneme variantu algoritmu, která bude hledat komponenty souvislosti grafu (resp. dané sítě). Jinou jednoduchou modifikací dostaneme variantu algoritmu pro „prohledávání do hloubky“ nebo pro

„prohledávání do šířky.“ Uvedený algoritmus půjde snadno modifikovat a použít pro jakékoliv zpracování struktury grafu, čímž rozumíme zpracování každého vrcholu grafu, případně každé hrany grafu. Nebudeme řešit implementaci algoritmu do konkrétního programovacího jazyka, ale zaměříme se na princip algoritmu.

```
// na vstupu je graf G
vstup < graf G;
stav(všechny vrcholy a hrany G) = iniciační;
uschozna U = {libovolný vrchol u grafu G};
stav(u) = nalezený;

// zpracování vybrané komponenty G
while (U je neprázdná) {
    vyber vrchol v a odeber jej z úschovny "U := U - {v}";
    ZPRACUJ(v);
    for (hrany e vycházející z v) { // pro všechny hrany
        if (stav(e) == iniciační)
            ZPRACUJ(e);
        w = druhý vrchol hrany e = vw; // známe sousedy?
        if (stav(w) == iniciační) {
            stav(w) = nalezený;
            přidej vrchol w do úschovny "U := U + {w}";
        }
        stav(e) = zpracovaná;
    }
    stav(v) = zpracovaný;

    // případný přechod na další komponentu G
    if (U je prázdná && G má další vrcholy)
        uschozna U = {vrchol v_1 z další komponenty G};
}
```

ALGORITMUS 10: PROCHÁZENÍ SOUVISLÝCH KOMPONENT GRAFU (KOVÁŘ, 2012)

Stručně popíšeme hlavní části algoritmu. Na začátku

- všem vrcholům i hranám přiřadíme iniciační stav,
- vybereme libovolný vrchol z úschovny.

V průběhu každého cyklu algoritmu zpracujeme nějaký jeden vrchol v . To znamená, že

- zpracovaný vrchol v z úschovny odstraníme,
- prozkoumáme (a případně zpracujeme) všechny dosud nezpracované hrany incidentní, s vrcholem v ,

Prozkoumáme (a zpracujeme) tak celou komponentu grafu, která obsahuje výchozí vrchol u .

Jestliže se v grafu nachází další nezpracované vrcholy, víme že graf nebude souvislý a bude mít více komponent. Algoritmus umí najít (a zpracovat) všechny komponenty, pro rozlišení komponent můžeme zpracované vrcholy zařazovat do různých struktur (polí, množin apod.).

Různým způsobem implementace úschovny dostaneme několik různých variant Algoritmus 10.

Procházení „do hloubky“ Jestliže úschovnu \mathcal{U} implementujeme jako zásobník, tak vrchol zpracovávaný v dalším cyklu bude poslední nalezený vrchol. Zpracováváme stále další a další, třeba i vzdálenější vrcholy, pokud existují.

Procházení „do šířky“ Jestliže úschovnu \mathcal{U} implementujeme jako frontu, tak nejprve zpracujeme všechny vrcholy sousední s vrcholem u . Označme si tyto vrcholy pro přehlednost v_1, v_2, \dots, v_d . Dále postupně zpracujeme všechny nezpracované vrcholy, které jsou sousední s vrcholy v_1, v_2, \dots, v_d . (jedná se o vrcholy ve vzdálenosti 2 od výchozího vrcholu u)

Dijkstrův algoritmus pro nejkratší cesty Z úschovny vybíráme vždy ten vrchol, který je nejbližší k výchozímu vrcholu u . Podrobně se tomuto algoritmu budeme věnovat později.

Poznámky o složitosti Algoritmus 10

Algoritmus 10 je nejen přehledný, ale současně rychlý. Počet kroků algoritmu je úměrný součtu počtu vrcholů a hran daného grafu. Obecně můžeme říci, že složitost algoritmu je $O(n+m)$, kde n udává počet vrcholů a m počet hran daného grafu. To znamená, že například pro graf se 100 vrcholy a 400 hranami bude Algoritmus 10

potřebovat řádově $c \cdot (100 + 400)$ kroků, kde parametr c závisí na konkrétní implementaci. Pokud bychom měli graf se 100 000 vrcholy a 400 000 hranami, tak počet kroků Algoritmus 10 bude řádově $c \cdot (100\,000 + 400\,000)$. Ve zmíněném algoritmu lze očekávat, že parametr c je v řádu desítek.

Cvičení

- 1) Jak pomocí Algoritmus 10 lze vypsat všechny hrany daného grafu?
Řešení: Stačí využít funkce `zpracuj(e)`. Jestliže e je zpracovávaná hrana, tak ji v těle funkce `zpracuj(e)` vypíšeme.
- 2) Jak pomocí Algoritmus 10 zjistíme, zda je graf G souvislý?
Řešení: Stačí upravit poslední řádek algoritmu. Jestliže úschovna U prázdná a v grafu G jsou další vrcholy, tak graf G není souvislý. Protože jsme zpracovali všechny vrcholy a hrany komponenty, která obsahuje vrchol u , tak úschovna U je prázdná. Ale graf obsahuje další vrcholy, které nejsou dosažitelné z vrcholu u a není proto souvislý.
Naopak, jestliže je úschovna U prázdná a žádné další vrcholy v grafu G nejsou, jsou všechny vrcholy dosažitelné z vrcholu u a graf G je souvislý.
- 3) Jak pomocí Algoritmus 10 najít a označit všechny komponenty grafu G ?
Řešení: Zavedeme pomocnou proměnnou označující číslo komponenty, například k . Na začátku algoritmu položíme $k = 1$ a při zpracování každého vrcholu mu přiřadíme číslo komponenty k .
Dále upravíme poslední řádek algoritmu. Jestliže je úschovna U prázdná a současně jsou v grafu G další vrcholy, tak graf G není souvislý. Do úschovny uložíme libovolný zbývajících vrchol `v` a zvýšíme hodnotu proměnné k . Při zpracování vrcholů pak každému vrcholu přiřazujeme již nové číslo komponenty. Podobně postupujeme i pro zbývajících komponenty.

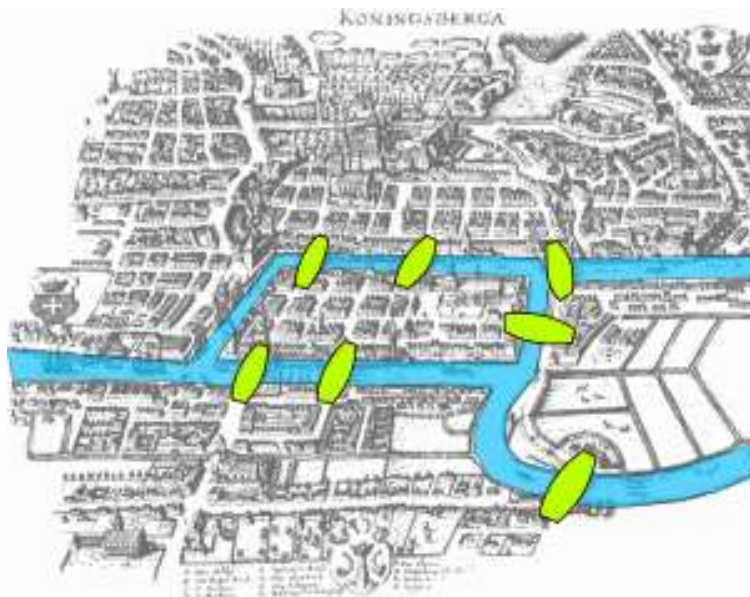
7.4 Eulerovské grafy

Historicky první problém vyřešený pomocí teorie grafů v roce 1736 byl tak zvaný Problém sedmi mostů města Královce.

V 18. století byl Královec (dnes Kaliningrad /Ruská federace/) bohatým městem. Na tehdejší dobu vyspělá ekonomika umožnila městu vystavět celkem sedm mostů přes řeku Pregolu

Podle tradice trávili měšťané nedělní odpoledne procházkami po městě. Vznikla tak

otázka, zda je možno projít všech sedm královeckých mostů, každý z nich právě jednou. Žádnému z měšťanů se to nedařilo, avšak nikdo z nich ani neuměl zdůvodnit, proč by to nebylo možné.



OBRÁZEK 7-23: SEDM MOSTŮ MĚSTA KRÁLOVCE (WIKIPEDIE, 2018)

Oslovili Leonharda Eulera, který v té době žil v Petrohradu, zda by problém uměl vyřešit. Euler (čti „ojler“) úlohu snadno vyřešil. Uvědomil si, že při jejím řešení nevyužil ani geometrii, ani algebru ani známe početní metody, ale metodu, kterou ve své korespondenci s německým matematikem Leibnizem nazvali „geometrie pozic.“ Euler při řešení problému sedmi mostů města Královce tuto metodu formálně zavedl. Dnes bychom řekli, že použil teorii grafů při hledání tahu, který obsahuje každou hranu daného grafu právě jedenkrát.

Pošťák má za úkol roznést poštu do každé ulice ve svém okrsku. Je přirozené, že si naplánuje trasu tak, aby každou ulicí procházel, pokud možno jen jednou – nachodí se tak co nejméně, a navíc poštu doručí dříve. Představme si graf, který modeluje ulice okrsku tak, že hrany odpovídají ulicím a křižovatky vrcholům grafu. Pošťáková úloha tak přesně odpovídá hledání tahu, který obsahuje každou hranu právě jednou. Podobně můžeme plánovat trasu sněžné frézy, která odklízí chodníky. Možná budeme chtít některé ulice projít dvakrát – jednou po každé straně kde se nachází chodník. Grafový model snadno upravíme tak, aby chodníky po obou stranách ulice odpovídaly násobným hranám nebo interně disjunktním cestám v grafu.

Podobně mohou trasy svých vozů plánovat popeláři, kropící vozy atd. (Kovář, 2012)

7.4.1 Kreslení jedním tahem

Sled:

Sled v grafu je posloupnost vrcholů taková, že mezi každými dvěma po sobě jdoucími vrcholy je hrana.

Souvislý graf:

Neorientovaný graf, v němž platí, že pro každé dva vrcholy $V_n, V_m, \forall n, m \in \mathbb{N}$ existuje sled z V_n do V_m nazveme Souvislým grafem.



A **souvislý graf**



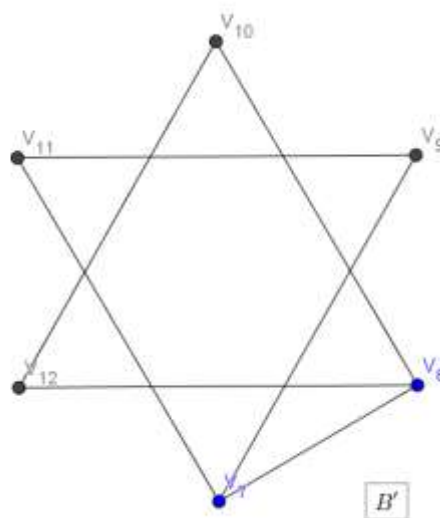
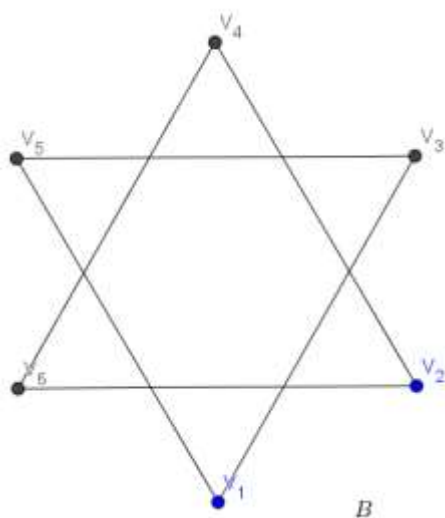
B **nesouvislý graf**

OBRÁZEK 7-24: SOUVISLÝ A NESOUVISLÝ GRAF (JIROVSKÝ, 2008)

Souvislý graf poznám tak, že pokud si vyberu dva libovolné vrcholy, tak vždy existuje různě dlouhá cesta z jednoho vrcholu do druhého vrcholu.

∴ Cvičení:

- 1) Prokaž je Graf B je nesouvislý.
- 2) Jednoduše uprav Graf B na Obrázek 7-24 tak, aby byl souvislý.



Uzavřený graf:

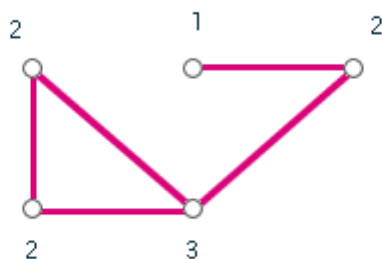
Tah T , který začíná a končí ve stejném vrcholu daného grafu G se nazývá *uzavřený tah*.

Uzavřený eulerovský tah:

Uzavřený tah v souvislém grafu G , který navíc obsahuje všechny hrany grafu G , se nazývá *uzavřený eulerovský tah*.

Otevřený eulerovský tah

Tah v souvislém grafu G , který obsahuje všechny hrany grafu G a výchozí vrchol se liší od koncového vrcholu se nazývá *otevřený eulerovský tah*.



OBRÁZEK 7-25: OTEVŘENÝ EULEROVSKÝ TAH (Jirovský, 2008)

Eulerovský graf:

Řekneme, že graf je *eulerovský*, právě tehdy, je-li v něm uzavřený eulerovský prvek.

Eulerovský graf (též E-graf) je takový souvislý neorientovaný graf, který má všechny uzly sudého stupně, tj. existuje uzavřený tah obsahující všechny jeho hrany.

Orientovaný graf je Eulerovský, existuje-li uzavřený tah obsahující všechny jeho hrany.



OBRÁZEK 7-26: KRESLENÍ JEDNÍM TAHEM I

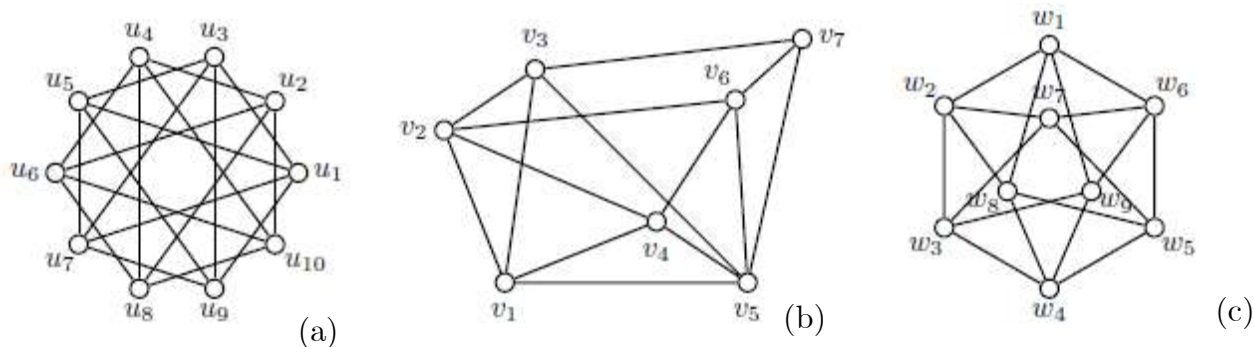
• Příklad: Který z grafů na Obrázek 7-26: Kreslení jedním tahem lze nakreslit jedním tahem? (Řešení viz (Kovář, 2012, s. 51))

Eulerova věta: Graf G lze nakreslit *jedním uzavřeným tahem* právě tehdy, když je graf G souvislý a všechny jeho vrcholy jsou sudého stupně.

Důkaz: viz (Kovář, 2012, s. 51)

Věta: Graf G lze nakreslit *jedním otevřeným tahem* právě tehdy, když je graf G souvislý, a právě dva jeho vrcholy jsou lichého stupně

Důkaz: viz (Kovář, 2012, s. 52)



OBRÁZEK 7-27: KRESLENÍ JEDNÍM TAHEM II

• Příklad: Který z grafů na Obrázek 7-27: Kreslení jedním tahem lze nakreslit jedním tahem?

Řešení:

Graf (a) má všechny vrcholy sudého stupně, nicméně graf (a) není souvislý. Proto v něm neexistuje uzavřený ani otevřený eulerovský tah

Graf (b) je souvislý a má právě dva vrcholy (v_5, v_7) lichého stupně. V grafu existuje otevřený eulerovský tah, např.:

$$v_7, v_6, v_4, v_1, v_2, v_3, v_5, v_1, v_3, v_7, v_5, v_6, v_2, v_4, v_5,$$

a tedy jej lze nakreslit jedním otevřeným tahem.

Graf (c) je souvislý a má všechny vrcholy sudého stupně, proto v něm existuje uzavřený eulerovský tah, např.:

$$w_1, w_2, w_7, w_5, w_8, w_2, w_3, w_4, w_8, w_1, w_9, w_4, w_5, w_6, w_7, w_3, w_9, w_6, w_1$$

a tedy jej lze nakreslit jedním uzavřeným tahem.

7.4.2 Algoritmus & eulerovské grafy (Kovář, 2012)

Pokud bychom chtěli sestavit algoritmus, který bude v daném grafu hledat uzavřený eulerovský tah, tak nejprve můžeme snadno ověřit, zda takový tah vůbec existuje.

Souvislost grafu ověříme užitím postupu popsaného ve *cvičení (3) na str. 144 (kapitola 7.3, Prohledávání grafu)* a ověřit, zda je každý vrchol sudého stupně je snadné.

Nyní najdeme libovolný uzavřený tah T postupem, který je popsán v (Kovář, 2012), str. 51 (indukční krok důkazu Věty 3.4). Dále najdeme komponenty souvislosti grafu $G - T$ (podle *cvičení (3) na str. 144 (kapitola 7.3, Prohledávání grafu)*) a pro každou netriviální komponentu najdeme rekurzivně uzavřený eulerovský tah (eulerovský vzhledem ke komponentě, nikoliv vzhledem k původnímu grafu G). Tyto uzavřené tahy vhodně vložíme do uzavřeného tahu T a dostaneme uzavřený eulerovský tah původního grafu G .

Vzhledem k praktickým motivacím, které jsme zmiňovali v úvodu kapitoly se kreslení více tahy může hodit při plánování tras pro více poštáků nebo pro více popelářských vozů.

7.5 Kostra grafu

Kostra souvislého grafu G je takový podgraf $G' \subset G$ na množině všech jeho vrcholů, který je stromem.

Příklady:

Mějme graf K_n (kompletní graf na n vrcholech), viz kap. 7.1.6.1 na str. 137. Graf K_n má právě n^{n-2} různých koster. (*Cayleyho vzorec*)

Mějme graf C_n (cyklus na n vrcholech), viz kap. 7.1.6.2 na str. 138. Graf C_n má n různých koster.

Libovolný strom má jedinou kostru – sám sebe.

7.5.1 Algoritmy pro hledání kostry grafu

Následující algoritmu nalezneme blíže neurčenou kostru grafu $G = (V, E)$:

- 1) Nechť $G = (V, E)$ je graf s n vrcholy a m hranami ($|V| = n$, $|E| = m$).
Seřaďme hrany grafu G libovolně do posloupnosti (e_1, e_2, \dots, e_m)
Položme $E_0 = \emptyset$.
- 2) Byla-li již nalezena množina E_{i-1} , spočítáme množinu E_i takto:
 - $E_i = E_{i-1} \cup \{e_i\}$, pokud neobsahuje graf $G'' = (V, E_{i-1} \cup \{e_i\})$ cyklus
 - $E_i = E_{i-1}$ ve všech ostatních případech.
- 3) Algoritmus se zastaví, pokud
 - a) buď E_i již obsahuje $n - 1$ hran,
 - b) nebo $i = m$, tedy se probraly všechny hrany z grafu G .

Graf $G' = (V, E_i)$ pak představuje kostru grafu G .

Je-li navíc definována funkce $w: E \rightarrow \mathbb{R}$ (tzv. ohodnocení hran), má smysl hledat minimální kostru grafu. Tedy takovou kostru $G' = (V, E')$, pro kterou platí:

$$w(E') = \min \left(\sum_{e \in E'} w(e) \right)$$

Tuto úlohu řeší mnoho různých algoritmů, které se řadí mezi tzv. hladové algoritmy (greedy search), neboť jednou provedená rozhodnutí už nikdy nemění čili *hladově* postupují přímo k řešení. Příkladem hladových algoritmů, které nalézají minimální kostru grafu je Borůvkův algoritmus nebo Jarník-Primův algoritmus.

7.5.2 Kruskalův algoritmus (x)

7.5.3 Borůvkův algoritmus (x)

7.5.4 Jarníkův-Primův algoritmus

Algoritmus je v zahraničí známý téměř výlučně pod označením Primův algoritmus, vzácně pak Jarníkův algoritmus nebo DJP algoritmus. Poprvé algoritmus popsal český matematik Vojtěch Jarník roku 1930. Později byl znovuobjeven roku 1957 Robertem Primem a poté ještě jednou roku 1959 Edsgerem Dijkstrou.

Algoritmus začíná s jedním vrcholem a postupně přidává další a tím zvětšuje velikost stromu do té doby, než obsahuje všechny vrcholy.

1. Vstup: souvislý ohodnocený graf $G(V, E)$.
2. Inicializace: $V' = \{v_0\}$, kde v_0 je libovolný vrchol z V ,
 $E' = \emptyset$ (prázdná množina).
3. Opakuj, dokud není $V' = V$:
 - a. Vyber hranu (v_a, v_b) z E s minimálním ohodnocením tak, že $v_a \in V' \wedge v_b \notin V'$.
 - b. Přidej v_b do V' ; přidej (v_a, v_b) do E' .
4. Výstup: $G'(V', E')$ je minimální kostra grafu G .

7.6 Hamiltonovský graf

Hamiltonovský graf je graf, který lze projít takovou cestou, že každý jeho uzel je navštíven právě jednou s výjimkou uzlu výchozího, který je zároveň uzlem cílovým.

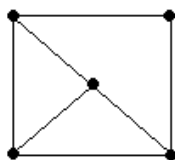
Graf je hamiltonovský, právě když obsahuje hamiltonovskou kružnici, tedy kružnici, která prochází všemi jeho uzly.

7.6.1 Podmínky postačující hamiltonovského grafu

Ačkoliv se hamiltonovské grafy zdají být obdobou eulerovských grafů, tak rozhodnout, zda je graf *hamiltonovský*, není vždy snadné. Dosud totiž není známa žádná primitivní nutná a postačující podmínka k tomu, aby graf byl hamiltonovský. Je však známo několik postačujících podmínek k *hamiltonovskosti grafu*. K tomu, aby byl graf *hamiltonovský* tedy stačí splnění některé z následujících podmínek:

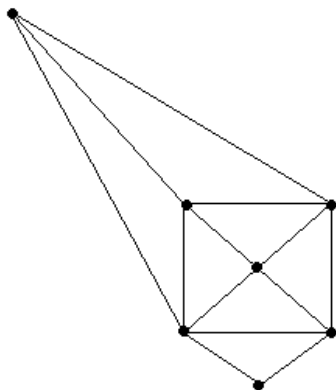
1. Označme $|V|$ počet uzlů grafu a předpokládejme, že $|V| \geq 3$. Má-li každý uzel stupeň alespoň $\frac{1}{2}|V|$, je graf *hamiltonovský*. (tzv. *Diracova podmínka*)
2. Označme $|V|$ počet uzlů grafu a předpokládejme, že $|V| \geq 3$. Je-li pro každou dvojici uzlů, které nejsou spojeny hranou, součet jejich stupňů alespoň $|V|$, pak je graf *hamiltonovský*. (tzv. *Oreho podmínka*)
3. Označme $|V|$ počet uzlů grafu a předpokládejme, že $|V| \geq 3$. Jestliže pro $\forall k \in \mathbb{N}: k < \frac{1}{2}|V|$ je počet uzlů, jejichž stupeň nepřevyšuje k , menší než k , pak je graf *hamiltonovský*. (tzv. *Pósova podmínka*)

7.6.2 Příklady



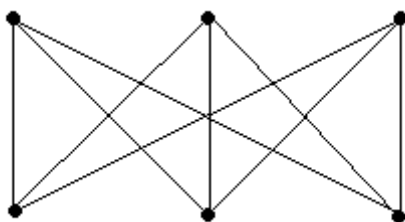
OBRAZEK 28: HAMILTONOVSKÝ GRAF SPLŇUJÍCÍ OREHO A PÓSOVU PODMÍNKU

Graf na obrázku ?? splňuje *Oreho podmínku*, tedy součet stupňů libovolných dvou nespojených uzlů je aspoň 5. Daný graf splňuje i *Pósovu podmínku*, tedy počet uzlů stupně 1 je menší než 1, počet uzlů stupně 1 a 2 je menší než 2.



OBRÁZEK 29: HAMILTONOVSKÝ GRAF SPLŇUJÍCÍ PÓSOVU PODMÍNKU.

Graf na obrázku ?? splňuje *Pósovu podmínku*, tedy počet uzlů 1. stupně je menší než 1, počet uzlů 1. a 2. stupně je menší než 3 a tedy je hamiltonovský.



OBRÁZEK 30: HAMILTONOVSKÝ GRAF SPLŇUJÍCÍ DIRACOVU, OREHO I PÓSOVU PODMÍNKU

Graf na obrázku splňuje Diracovu podmínku, neboť každý jeho uzel má stupeň alespoň 3 a splňuje i podmínku Oreho, neboť každé dva uzly nespojené hranou mají součet stupňů alespoň 6. A konečně splňuje i Pósovu podmínku, neboť v něm nejsou žádné uzly stupně menšího než 3.

7.6.3 Hamiltonovská kružnice (hamiltonovský cyklus), Hamiltonovská cesta

V teorii grafů je hamiltonovská kružnice (také hamiltonovský cyklus) grafu taková kružnice v tomto grafu, která projde právě jednou všemi jeho vrcholy.

Hamiltonovská cesta je taková cesta v daném grafu, která prochází každým jeho vrcholem právě jednou.

Každý graf nemusí mít nutně hamiltonovskou kružnici. Nutnými (avšak nikoli postačujícími) podmínkami je, že graf musí být souvislý a každému vrcholu musí vést alespoň 2 hrany.

7.7 Známé úlohy z teorie grafů

7.7.1 Problém obchodního cestujícího (x)

= Travelling Salesman Problem (TSP)

Jedná se o diskrétní netriviální matematický problém, vyjadřující a zobecňující úlohu nalezení nejkratší možné cesty procházející všemi zadanými body na mapě.

Zadání: V dané zemi existuje mnoho měst, mezi všemi městy je postavené silnice. Cílem obchodního cestujícího je objet všechna města takovým způsobem, aby cena za jízdenky byla minimální (odpovídá vzdálenosti měst) a vrátit se do výchozího města.

Matematická formulace téhož zadání: V daném ohodnoceném kompletním grafu najděte nejkratší hamiltonovskou kružnici.

Optimalizační verze problému obchodního cestujícího patří mezi tzv. NP-těžké úlohy, tzn. v obecném případě není známo, ani jak pro každý vstup nalézt přesné řešení v rozumném čase, a dokonce ani zda vůbec může existovat algoritmus, který takové řešení najde v čase úměrném nějaké mocnině počtu uzlů.

Že jde o nedeterministicky polynomiální problém, je patrné z toho, že nedeterministický počítač, umožňující v každém kroku rozvětvit výpočet na libovolný počet větví, by mohl začít v některém „městě“, rozdělit propočet délky trasy na tolik větví, kolik z

města vede silnic, a v každém z cílových měst postupovat stejně – s výjimkou tras vedoucích do již navštívených měst. Tak by prohledal všechny možné trasy v n výpočetních krocích, pokud počet měst činí n , a rozvětvil by se nejvýše do $(n-1)!$ větví. *Zjednodušeně řečeno, pro skoro všechna n není jisté, zda je problém řešitelný v použitelném čase.*

Řekneme, že vlastnost přirozených čísel V je splněna pro skoro všechna n , pokud

$$\exists n_0 \in \mathbb{N}: \forall n \geq n_0 : V(n).$$

V praxi se podobná úloha obvykle řeší pouze aproximačně (heuristickými algoritmy, např. genetickými algoritmy, tabu prohledáváním atd.). Tím se – za cenu vzdání se nároku na nalezení optimálního řešení – dosahuje prakticky použitelných časů.

Heuristické algoritmy obecně nijak negarantují, jak moc se získaný výsledek liší od optimální cesty, pro metrický problém obchodního cestujícího však existují prakticky použitelné (polynomiální) algoritmy, které toto dovedou (např. Christofidův algoritmus najde cestu, která je nejhůře o polovinu delší).

Pomocí celočíselného lineárního programování lze problém asymetrického obchodního cestujícího (tzn. hrany $A \rightarrow B$ a $B \rightarrow A$ nemusejí mít stejnou váhu) formulovat též jako:

$$\min \sum_{i=1}^n \sum_{j=1}^n x_{ij} \cdot c_{ij}$$

Jsou-li splněny podmínky:

$$\sum_{i=1}^n x_{ij} = 1; \quad j \in \{1, \dots, n\} \quad \text{a} \quad \sum_{j=1}^n x_{ij} = 1; \quad i \in \{1, \dots, n\}$$

$$s_i + c_{ij} - (1 - x_{i,j}) \cdot M \leq s_j; \quad i \in \{1, \dots, n\}, j \in \{1, \dots, n\}$$

$$x_{ij} \in \{0,1\}$$

7.7.1.1 Metrický problém obchodního cestujícího

Variantou tohoto problému je problém metrického obchodního cestujícího, ve kterém vzdálenosti na grafu splňují trojúhelníkovou nerovnost. Toto zjednodušení odpovídá velkému množství reálných problémů (např. hledání na mapě), a zároveň umožňuje konstrukci aproximačních algoritmů.

*Příklady algoritmů, které problém obchodního cestujícího řeší či aproximují:

- 2-aproximační algoritmus
- Christofidesův algoritmus
- Algoritmus Chiméra
- Fragmentační algoritmus
- Algoritmus postupných permutací
- Algoritmus hledání na trojúhelníkové síti
- Algoritmus polární trasy

7.7.2 Problém čínského listonoše

= Guan's route problem / Chinese postman problem / Postman tour / Route inspection problem

Jedná se o optimalizační úlohu, která spadá do problematiky nalezení cest v grafu.

Úloha pochází z roku 1960 a formuloval ji čínský matematik Mei-Ko Kwan.

Zadání: Listonoš musí zajít na poštu, vzít dopisy a obejít s nimi všechny ulice města a nakonec se vrátit do výchozího bodu – zpět na poštu. Musí přitom urazit minimální vzdálenost.

V grafu, který reprezentuje město, představují hrany grafu ulice a uzly odpovídají křižovatkám. Hrany jsou ohodnoceny kladnými čísly, které odpovídají délce ulic.

Je-li možné v grafu provést eulerovský tah, řešení je triviální. Listonoš projde všemi ulicemi (=hranami) právě jednou. Součet ohodnocení hran udává délku cesty.

V opačném případě (v případě, že neexistuje eulerovský tah) je nutné do grafu přidat hrany (resp. je zdvojit), tak aby bylo možné v novém grafu Eulerův cyklus nalézt. Vybíráme hrany s nejnižším ohodnocením (kvůli optimalizaci).

Aplikace tohoto problému jsou v reálném světě zcela zřejmé: úklid, čištění, sypání silnic, kontroly dopravního značení, ...

7.7.3 Problém čtyř barev (Barvení map) (x)

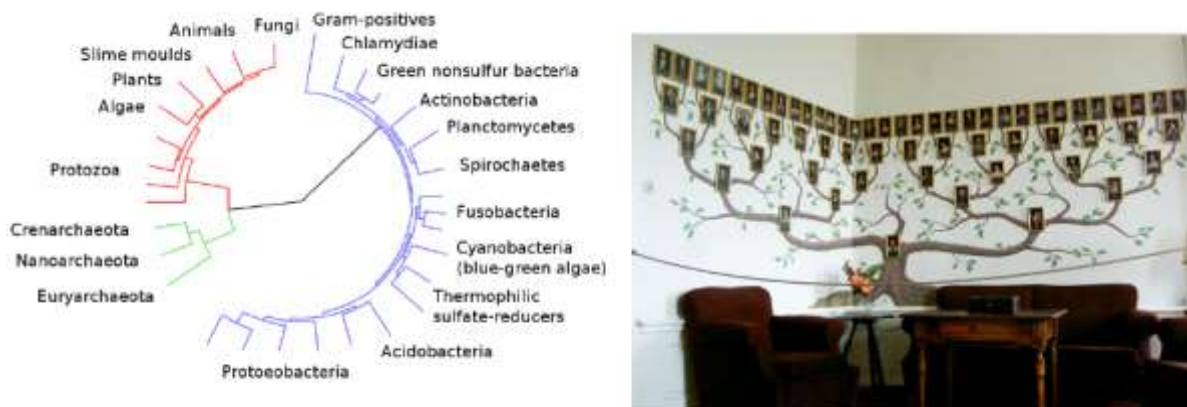
7.7.4 Další

Mezi další poměrně známé problémy z oblasti teorie grafů patří např. *hledání nejkratší hamiltonovské kružnice*, *problém kliky*, *hledání isomorfního podgrafu*, *zavazadlový problém* či *problém dvou loupežníků*.

Je nutno doplnit informaci, že se jedná většinou o tzv. NP-úplné problémy, tedy takové *nedeterministicky polynomiální problémy*, na které jsou polynomiálně redukovatelné všechny ostatní problémy z NP. To znamená, že třídu NP-úplných úloh tvoří v jistém smyslu ty nejtěžší úlohy z NP. Pokud by byl nalezen polynomiální deterministický algoritmus pro nějakou NP-úplnou úlohu, znamenalo by to, že všechny nedeterministicky polynomiální problémy jsou řešitelné v polynomiálním čase, tedy že třída NP se „zhroutí“ do třídy P ($NP = P$). Otázka, zda nějaký takový algoritmus existuje, zatím nebyla rozhodnuta, předpokládá se však, že $NP \neq P$ (je však zřejmé, že $P \subseteq NP$). Vztah mezi P a NP je jedním ze sedmi problémů milénia, které vypsala *Clayův matematický ústav asi v polovině roku 2000*. Za rozhodnutí vztahu nabízí 1 000 000 dolarů.

7.8 Stromy a lesy

Mezi nejběžnější útvary v přírodě i v matematice patří „stromy“, tj. objekty se „stromovou strukturou“. Existuje celá řada situací, které můžeme popsat „stromem“: rodokmeny, evoluční strom, elektrické rozvodné sítě, hierarchické struktury se vztahem nadřazený a podřízený, popis větvení při vyhledávání apod. Všechny tyto struktury mají jednu věc společnou – neobsahují „cykly“. (Kovář, 2012)

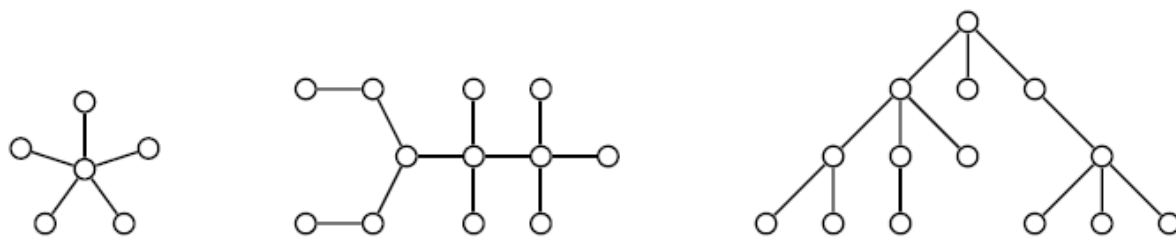


OBRÁZEK 7-31: ZJEDNODUŠENÝ EVOLUČNÍ STROMY A RODOKMEN (KOVÁŘ, 2012)

Z hlediska teorie grafů je stromem každý souvislý graf bez kružnic o $|V| - 1$. Jedná se o hierarchickou strukturu, kde každý má otec O až n synů a každý syn právě jednoho otce takovým způsobem, že v této struktuře nejsou cykly.

Strom je souvislý graf, který neobsahuje cyklus.
Les je graf, jehož komponenty jsou stromy.

Lesem též můžeme rozumět libovolný graf G , který neobsahuje cykly.



OBRÁZEK 7-32: PŘÍKLADY STROMŮ (KOVÁŘ, 2012)

Na *Obrázek 7-32: Příklady stromů* jsou tři příklady grafů, které nazývám stromy.

Tyto grafy tvoří dohromady jediný graf, kterému říkáme les.

V: Každý strom s více než jedním vrcholem má alespoň jeden list.

Důkaz: viz (Kovář, 2012, s. 83)

V: Strom s n vrcholy má právě $n - 1$ hran.

Důkaz: viz (Kovář, 2012, s. 84)

Cvičení:

- ~ A) V databázi je 12 záznamů (objektů) a 34 vazeb mezi nimi. Chtěli bychom strukturu databáze přehledně znázornit grafem, ve kterém jsou objekty znázorněny jako vrcholy a vazby mezi nimi jsou zakresleny jako hrany.
- ~ ~ ~
- ~ B) Bude takový graf stromem?
- ~
- ~ C) Můžeme tvrdit, že takový výsledný graf bude souvislý?

Řešení:

- a. Strom s 12 vrcholy má právě $12 - 1 = 11$ hran. V databázi je však 12 objektů (vrcholů) s 34 vazbami (hranami). Zřejmě, tedy graf bude obsahovat cykly, tedy nejedná se o strom.
- b. Výsledný graf může, ale nemusí být souvislý. Výsledek je silně závislý na struktuře dat. Výsledný graf by mohl například obsahovat jednu komponentu s 9 vrcholy a 3 izolované vrcholy, neboť kompletní graf K_9 obsahuje $\binom{9}{2} = 36$ hran a my z něj můžeme vynechat 2 libovolné hrany.

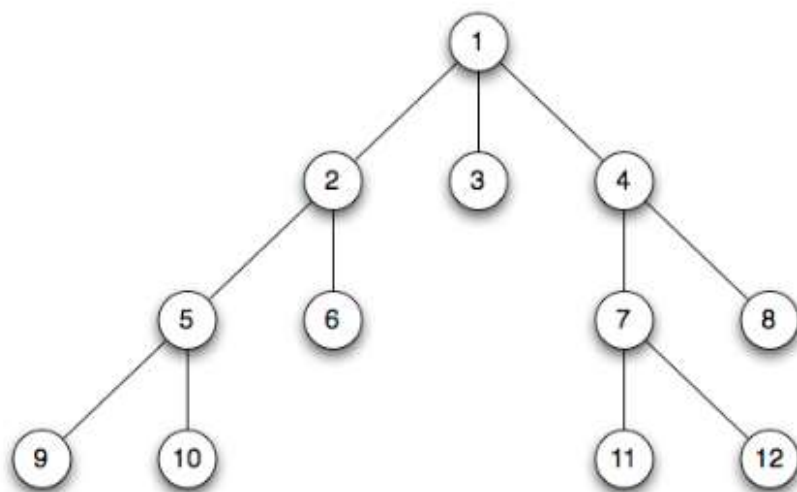
poznámka: $\binom{n}{k} = \frac{n!}{k!(n-k)!}, \forall n \in \mathbb{N}, \forall k \in \{\mathbb{N} \cap (0, k)\}$

Jestliže existuje uzel, který je základem orientace stromu, potom se jedná o tzv. *kořen stromu*.

Počet vrcholů daného stromu můžeme vyjádřit jako $|V| = |E| + 1$.

Počet hran daného stromu můžeme vyjádřit jako $|E| = |V| - 1$.

Strom je tedy minimální souvislý graf bez kružnic. Odebráním libovolné hrany se stane nesouvislým a přidáním libovolné hrany vznikne kružnice!



OBRÁZEK 7-33: PŘÍKLAD STROMU

Každý souvislý graf **G** má faktor⁸, který je stromem.

⁸ Necht' $G = (V_1, E_1)$ je graf. Graf $G' = (V_2, E_2)$ nazveme faktor grafu G , pokud $V_1 = V_2$.

Faktor grafu G má shodnou množinu vrcholů a vznikne pouze vynecháním některých hran! Mohou pak vznikat izolované uzly, které vynechané hrany tvořily.

7.8.1 Využití

Stromy mají široké uplatnění jako datové struktury pro různé algoritmy. Jsou to matematické abstrakce organizace dynamických množin, kterou v běžném životě používáme velice často.

Příkladem může být:

- rodokmen,
- soutěže založené na vylučovacím principu (dál postupuje jeden ze soupeřů),
- botanický klíč
- systém souborů uložených v adresářích, (definovaném rekurzivně) jako množinu souborů a adresářů.

7.8.2 Rekurzivní definice

Prvky ve stromu nazýváme vrcholy. Některé vrcholy jsou spojeny a toto spojení nazýváme hrany.

Strom potom můžeme definovat rekurzivně:

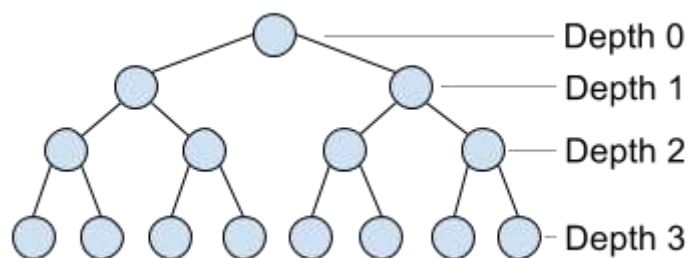
1. Jeden vrchol je strom. Tento vrchol se nazývá kořen stromu.
2. Nechť x je vrchol stromu a T_1, T_2, \dots, T_n jsou stromy. Strom je vrchol x spojený s kořeny stromů T_1, T_2, \dots, T_n .

7.8.3 Pojmy

Cesta je posloupnost po sobě jdoucích vrcholů, které jsou spojeny jednou, nebo více hranami.

Délka cesty je počet hran cesty. Délku cesty z vrcholu k sobě samému potom můžeme definovat jako nulovou. Ke každému vrcholu je z kořene právě jedna cesta.

Hloubka vrcholu ve stromě (depth) (úroveň, na které se vrchol nachází) je definována jako délka této cesty. Úroveň (hloubka) kořene stromu je tedy nulová.



OBRÁZEK 7-34: HLOUBKA VRCHOLU VE STROMĚ (STEYN, 2019)

Výška stromu je maximální hloubka vrcholu stromu.

Šířkou stromu rozumíme počet uzlů na stejné úrovni.

7.8.4 Základní prvky stromu – shrnutí

Kořen stromu (root)

- Nejvyšší vrchol (uzel) stromu
- Kořen je jediným vrcholem bez otce
- V každém stromu se nachází nejvýše jeden kořen

Vnitřní uzly (internal node)

- Uzel/vrchol, který není kořen a zároveň není koncovým uzlem.

Koncové uzly (listy) (leaf node)

- Takový vrchol stromu, který nemá žádného syna.
- Má-li strom pouze jeden vrchol, pak je tento vrchol kořenem a listem zároveň

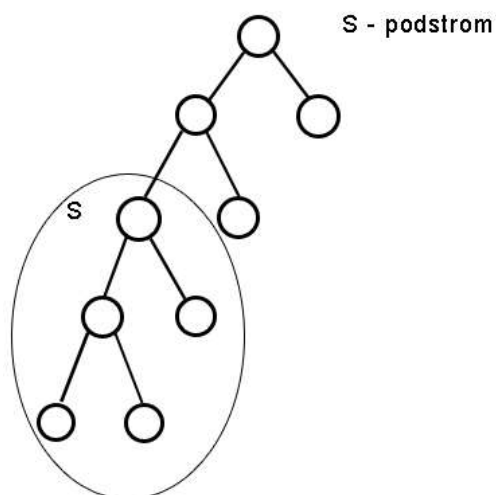
Maximální počet potomků

Konkrétní typy stromů mají většinou jednoznačně definován maximální počet potomků:

typ stromu	maximální počet potomků
binární strom	2
2-3 strom	3
B-strom	$n, \forall n \in \mathbb{N}$
speciální typ extrémně nevyrovnaného stromu (silně podobný lineárnímu seznamu)	1

Podstrom (subtree)

Část stromové struktury, tvořené jedním uzlem (kořenem podstromu) a všemi jeho potomky



OBRÁZEK 7-35: PODSTROM

```

/**
 * N-arní strom
 * @author Pavel Míčka
 */
public class Tree {
    /**
     * Kolekce potomků
     */
    private ArrayList<Tree> children;
    /**
     * Hodnota uchovávaná v uzlu
     */
    private Object value;
    /**
     * Pridá podstrom jako potomka tohoto kořene
     * @param t strom k přidání
     * @param index pořadí potomka

```

```

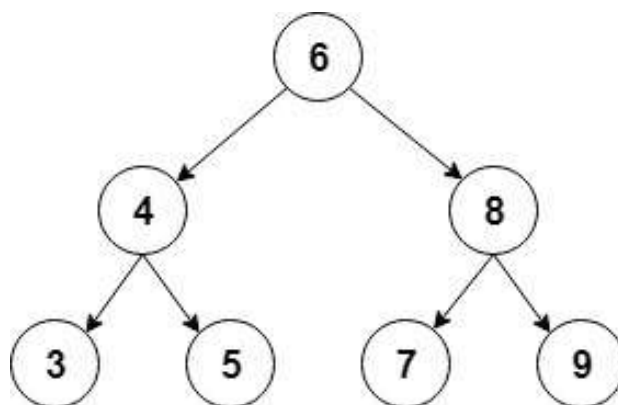
    */
    public void addChild(Tree t, int index){
        children.add(index, t);
    }
    /**
     * Odstrani potomka na danem indexu
     * @param index index potomka k odstraneni
     */
    public void removeChild(int index){
        children.remove(index);
    }
    /**
     * Vraci potomka na danem indexu
     * @param index poradi potomka
     * @return potomek na zadanem indexu
     */
    public Tree get(int index){
        return children.get(index);
    }
    /**
     * @return the value
     */
    public Object getValue() {
        return value;
    }
    /**
     * @param value the value to set
     */
    public void setValue(Object value) {
        this.value = value;
    }
}

```

ALGORITMUS 11: STROM (ALGORTIMY.NET, 2019)

7.8.5 Binární strom

Binární strom je prázdný strom nebo vrchol, který má právě dva syny, resp. levý a pravý podstrom. Tyto podstromy jsou binární stromy. Každý uzel ve stromu může tvořit kořen nového podstromu.



OBRÁZEK 7-36: BINÁRNÍ STROM

Binární stromy známe velmi důvěrně z kapitoly 2.2.4 Heapsort (řazení haldou), na str. 43.

7.8.5.1 Implementace binárního stromu

Pole

Binární strom můžeme implementovat pomocí pole, jehož prvky mají typ klíče. Pozice vrcholů binárního stromu lze očíslovat následujícím způsobem:

- Kořenu přiřadíme 1 (obecně i).
- Levému následníku 2 (obecně $2i$).
- Pravému následníkovi .. 3 (obecně index $2i + 1$).
- Předchůdce (pokud existuje) má index $\frac{i}{2}$ (celočíslný).

Obecně není tato implementace efektivní:

- musíme vytvořit pole s dostatečnou velikostí \Leftrightarrow předpokládána maximální velikost stromu)
- musí obsahovat i prvky pro neobsazené pozice

Spojový seznam

Strom lze implementovat pomocí *spojového seznamu*. Vrchol binárního stromu pak vytvoříme jako prvek spojového seznamu, akorát místo položky `další` (implementace polem), budou v implementaci položky `levý` a `pravý`.

```
class DVrchol {  
    int klic;  
    DVrchol levý;  
    DVrchol pravý;  
  
    DVrchol (int klic) {  
        this.klic = klic;  
    }  
  
    void tiskVrcholu() {  
        System.out.print(klic+" ");  
    }  
}
```

ALGORITMUS 12: BINÁRNÍ STROM V JAVĚ (IMPLEMENTACE POMOCÍ SPOJOVÉHO SEZNAMU)

7.8.6 Procházení stromem do šířky (BFS)

= *breadth-first search*

- Jeden ze základních grafových algoritmů (slouží k průchodu daného grafu)
- Principy BFS jsou základem pro další algoritmy (Jarník-Primův algoritmus, Dijkstrův algoritmus)

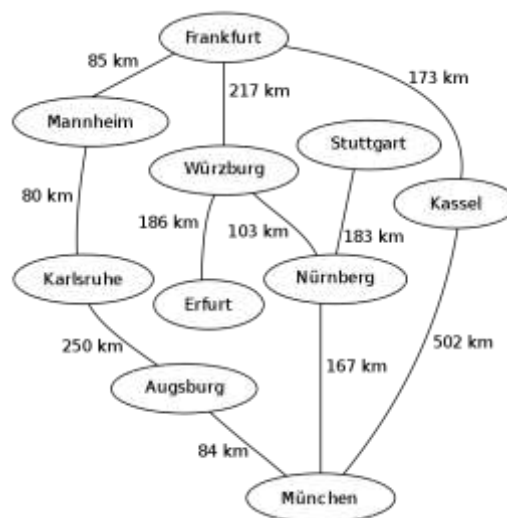
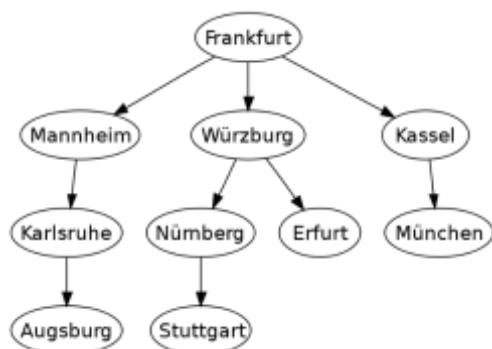
Princip

Algoritmus postupně prochází všechny sousedy startovního vrcholu (libovolný vrchol může být startovním vrcholem BFS), následně prochází sousedy sousedů atd.

Jedná se o metodu, která postupuje *systematickým prohledáváním grafu přes všechny uzly*. Prochází všechny uzly a pro každý projde všechny jejich syny. Přitom si poznamenává předchůdce jednotlivých uzlů a tím je poté vytvořen strom nejkratších cest k jednotlivým uzlům z vrcholu v kterém jsme začínali. Veškerí následovníci daného uzlu získaní expandujícím uzlem jsou vkládáni do FIFO fronty. FIFO fronta znamená, že první uzel, který do fronty vstoupil jí také první opustí.

OBRÁZEK 7-37 (NAHOŘE): MAPA NĚMECKA S HRANAMI MEZI MĚSTY (WIKIPEDIE, 2018)

OBRÁZEK 7-38 (DOLE): STROM VZNIKLÝ ALG BFS Z OBRÁZEK 7-37 (NAHOŘE): MAPA NĚMECKA S HRANAMI MEZI MĚSTY , KTERÝ STARTOVAL VE FRANKFURTU (WIKIPEDIE, 2018)



Pozn.: příklad zdrojového kódu v Javě není v

učebnici pro jeho rozsáhlost uveden. Lze jej najít např. na webu www.algoritmy.net

7.8.7 Procházení stromem do hloubky (DFS)

= *depth-first search*

- Jeden ze základních grafových algoritmů (prochází graf metodou backtracingu⁹)
- Široké uplatnění – jeho principů využívá zjišťování počtu komponent, topologické uspořádání, detekce cyklů daného grafu

Procházení začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větvi (tj. v žádném podstromu) již není následník.

Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:

- Přímý průchod (preorder) - navštívíme vrchol a potom levý a pravý podstrom.
- Vnitřní průchod (inorder) - navštívíme levý podstrom, vrchol a pravý podstrom.
- Zpětný průchod (postorder) - navštívíme levý a pravý podstrom a potom vrchol.

Binární strom je definován rekurzivně, jednotlivé průchody můžeme též definovat rekurzivně.

Preorder

```
// Preorder
void preorder(DVrchol v) {
    if (v == null) {
        return;
    }
}
```

⁹ zpětné vyhledávání, metoda pokusů a oprav, metoda zpětného sledování, metoda prohledávání do hloubky (řešení problému osmi dam, viz str. 46)

```

    }
    else {
        v.tiskVrcholu();
        preorder(v.levy);
        preorder(v.pravy);
    }
}

void pruchodPreorder() {
    preorder(koren);
}

```

ALGORITMUS 13: PREORDER

Inorder

```

// Inorder
void inorder(DVrchol v) {
    if (v == null) {
        return;
    }
    else {
        inorder(v.levy);
        v.tiskVrcholu();
        inorder(v.pravy);
    }
}

void pruchodInorder() {
    inorder(koren);
}

```

ALGORITMUS 14: INORDER

Postorder

```

// Postorder
void postorder(DVrchol v) {
    if (v == null) {
        return;
    }
    else {
        postorder(v.levy);
        postorder(v.pravy);
        v.tiskVrcholu();
    }
}

void pruchodPostorder() {
    postorder(koren);
}

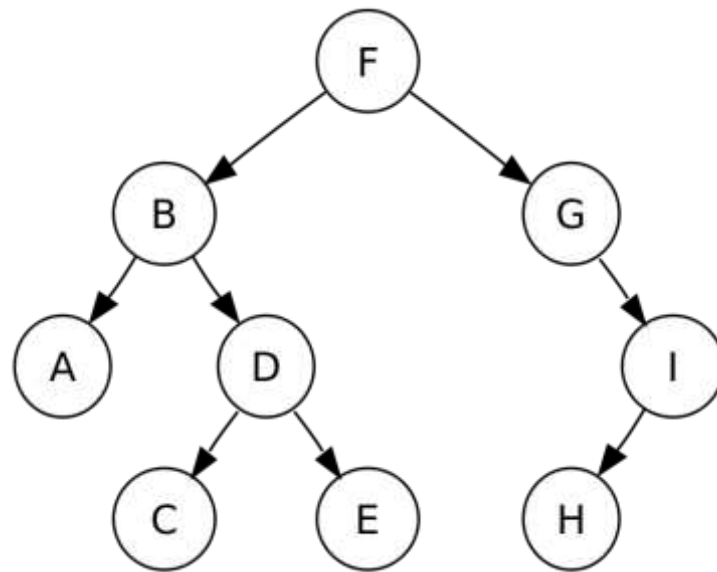
```

ALGORITMUS 15: POSTORDER

Při použití metody Inorder se prochází uzly v uspořádaném stromě podle jejich přirozeného pořadí.

Procházení stromem do hloubky lze řešit pomocí:

- Rekurze – funkce volá sama sebe
- Iterace – provádění stejné operace znovu a znovu



OBRÁZEK 7-39: BINÁRNÍ STROM

Výsledky způsobu procházení ve stromu znázorněném na *Obrázek 7-39: Binární strom*:

Terminologická poznámka:

- N navštívený uzel
- L levý uzel (left)
- R pravý uzel (right)

Odlišnosti v procházení do hloubky/do šířky:

- DFS – Preorder (N→L→R): F, B, A, D, C, E, G, I, H
- DFS – Inorder (L→N→R): A, B, C, D, E, F, G, H, I
- DFS – Postorder (L→R→N): A, C, E, D, B, H, I, G, F
- BFS – level-order: F, B, G, A, D, I, C, E, H

7.9 Speciální typy stromů

7.9.1 Binární halda (*binary heap*)

Připomeňte si! Kapitola 2.2.4 Heapsort (řazení haldou), str. 43 - 47.

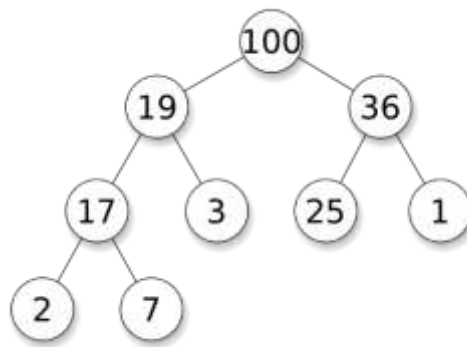
- Třídící algoritmy
- Vlastnosti heapsort

Je dána datová sada: | 4 | 2 | 5 | 3 | 1 | 9 |.

Datovou sadu seřaď vzestupně (tj. od nejnižší hodnoty po nejvyšší) algoritmem heapsort.

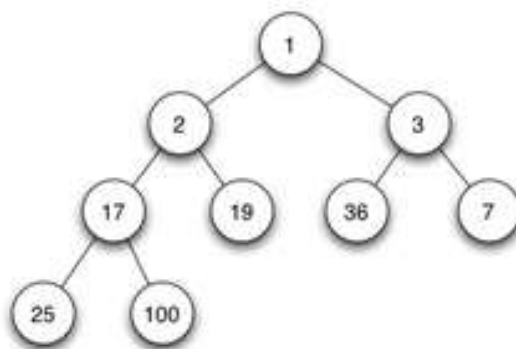
Načrtni jednotlivé kroky. Zaměř se na průběh algoritmu.

- Halda = stromová datová struktura, které má následující vlastnosti (3):
 - o (1) Všechna „*patra*“ haldy až na poslední jsou plně obsazeny prvky (tedy každý vnitřní vrchol má právě 2 syny \rightarrow (strom je velmi vyvážený)
 - o (2) Poslední patro haldy je zaplněno zleva (může být i zaplněno celé)
 - o (3a) Pro prvky v tzv. max haldě (*max-heap*) platí tzv. speciální vlastnost haldy: oba synové jsou vždy menší nebo rovny otci. nechť S je potomek O , pak $x(S) \geq x(O) \Rightarrow$ v kořenu stromu je vždy prvek s nejvyšším klíčem (nejvyšší hodnotou)



OBRÁZEK 7-40: MAX HALDA - UKÁZKA

- o (3b) Pro prvky v tzv. min haldě (*min-heap*) platí tzv. speciální vlastnost haldy: oba synové jsou vždy větší nebo rovny otci. nechť S je potomek O , pak $x(S) \leq x(O) \Rightarrow$ v kořenu stromu je vždy prvek s nejvyšším klíčem (nejvyšší hodnotou)



OBRÁZEK 7-41: MIN HALDA - UKÁZKA

- Úplný binární strom
- Vlastnost *být haldou* je rekurzivní – tzn. všechny podstromy haldy jsou také haldy
 - Tato vlastnost nám říká, že se halda chová analogicky k prioritní frontě (viz kap. 6.2 Fronta (*Queue*), str. 119). Na vrcholek hlady musí vždy vstoupit prvek s nejvyšší prioritou).
 - Tuto vlastnost využívá heapsort (řazení haldou). Více viz kap. 2.2.4 nahoře na stranì 43.

Další specifické typy stromů již velmi dobře známe ze str. 138 (cyklické a acyklické grafy)

7.10 Hledání nejkratší cesty – Dijkstrův algoritmus

*Dr. Edsger Wybe Dijkstra, IPA [čti: 'ɛt, sɛr 'dɛɪk, stra] (*1930 †2002) byl nizozemský informatik. V roce 1972 obdržel Turingovu cenu za své příspěvky rozvoji programovacích jazyků. Dijkstra se narodil v Rotterdamu ve vzdělané rodině. Otec byl chemikem a matka matematickou. Byl abiturientem střední školy pro neobyčejně nadané studenty. Ovládal mnoho jazyků, např. latinu, řečtinu, francouzštinu, němčinu, angličtinu. Vynikal v přírodních vědách – biologii, matematice a chemii.*

Po středoškolských studiích začal studovat obecnou fyziku na Leidenské univerzitě.

nejstarší univerzita v Nizozemsku (založena 1575, Vilémem I. Oranžským)

Významní absolventi: Adams (6. prezident USA); Albert Einstein; nizozemská královská rodina, ...

V létě 1951 docházel do letní školy na Cambridge, kde se účastnil předmětu programování.

V roce 1956 získává první doktorský titul z Leidenské university a druhý doktorský titul získává o 3 roku později na Universitě v Amsterdamu, v témže roce představuje algoritmus pro nalezení nejkratší cesty v grafu (později známý pod pojmem Dijkstrův algoritmus). Pozn.: E.

W. Dijkstra navrhl tento algoritmus ve svých 20 letech po nakupování se snoubenkou během 20 minut bez tužky a papíru.

V 80. letech 20. století byl jmenován předsedou oboru informatiky na Texaské univerzitě v Austinu, kde zůstává až do r. 2000. Umírá v 72 letech v rodném Nizozemí na rakovinu.

Dijkstrův algoritmus je algoritmus sloužící k nalezení nejkratší cesty v grafu.

- konečný (pro jakýkoliv konečný vstup algoritmus skončí) \Rightarrow v každém průchodu cyklu se do množiny navštívených uzlů přidá právě jeden uzel,
- průchodů cyklem je tedy nejvýše tolik, kolik má graf vrcholů. Funguje nad hranově kladně ohodnoceným grafem (neohodnocený graf lze však na ohodnocený snadno převést) \Rightarrow nejrychlejší známý algoritmus.

Řeší nejkratší cestu z vrcholu s (startovního vrcholu) do ostatních vrcholů grafu.

Označme si *výchozí* uzel, ze kterého budeme plánovat cestu. Dijkstraův algoritmus je založen na tom, že na začátku přiřadíme uzlům určité hodnoty a postupně je budeme vylepšovat.

One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention

– E. W. Dijkstra.

7.10.1 Použití

- všechny kladně ohodnocené grafové struktury algoritmů
- routovací protokoly (algoritmus OSPF¹⁰)
- Algoritmus A*

¹⁰ Open Shortest Path First (OSPF) je hierarchický interní směrovací protokol, fungující na bázi link-state, tzn. každý směrovač zná strukturu celé sítě (v případě OSPF přesněji celé oblasti). OSPF je nejpoužívanějším směrovacím protokolem pro směrování uvnitř autonomních systémů. Část sítě, v níž působí OSPF, se nazývá OSPF doména. (Wikipedia, 2019)

7.10.2 Postup a kroky I

První možnost představní Dijkstrova algoritmu

Algoritmus lze popsat slovně (slovním pseudokódem):

1. Ke každému uzlu přiřaď aktuální vzdálenost od počátečního uzlu. V případě samotného výchozího uzlu je vzdálenost 0, v případě ostatních uzlů je to nyní nekonečno.
2. Označ všechny uzly jako nenavštívené. Označ výchozí uzel jako aktuální uzel. Vytvoř množinu nenavštívených uzlů, která bude obsahovat všechny uzly kromě výchozího. Nyní se dostáváme k třetímu bodu, který se bude opakovat ve smyčce.
3. Podívej se na všechny nenavštívené sousedy aktuálního uzlu a vypočítej jejich vzdálenost od počátečního uzlu.
 - a. Pokud má *např.* aktuální uzel vzdálenost 3 a délka hrany mezi tímto uzlem a jeho sousedem je 2, potom vzdálenost k tomuto sousedovi bude $3 + 2 = 5$.
 - b. Pokud je aktuální vzdálenost menší než dříve zaznamenaná vzdálenost tohoto souseda, pak ji přepiš lepší, kratší hodnotou. Poznamenejme, že i když jsme se na sousední uzly aktuálního uzlu podívali, stále zůstávají v množině nenavštívených.
4. Jakmile jsme se podívali na sousedy aktuálního uzlu, budeme považovat tento aktuální uzel za navštívený a vyjmemme ho z množiny nenavštívených uzlů. K tomuto uzlu už se nikdy nevrátíme.
5. Pokud byla cílová destinace označena jako navštívená, nebo nejmenší aktuální vzdálenost mezi uzly v množině nenavštívených uzlů je nekonečno, ukonči algoritmus.
6. Vyber uzel z množiny nenavštívených uzlů, který má nejmenší aktuální vzdálenost od výchozího uzlu, nastav jej jako „aktuální“ uzel a vrať se k bodu 3.

7.10.3 Postup a kroky II

Druhá možnost představní Dijkstrova algoritmu

Vymežeme si vstup (input) a výstup (output):

Vstup

- Končený ohodnocený graf $G = (V, E)$,
kde V je neprázdná množina uzlů a E je neprázdná množina hran
- počáteční (inicializační) uzel $s \in V$

Výstup

- asociativní pole $D[u]$ – udává nejkratší vzdálenost mezi uzlem s a uzlem u

Inicializace

- vytvoř množinu uzlů $X \subseteq V$
- do množiny X vlož počáteční uzel s
- Vytvoř asociativní pole čísel pro každý uzel $D[u]$
- Inicializuj hodnoty pole D takto:
 - Pro počáteční uzel $s = 0$
 - Pro každý uzel u sousedící s počátečním uzlem $s = |s, u|$.
 - Pro ostatní uzly ohodnocení $e = \infty$, kde $e \in V \setminus X$

Výpočet

- Dokud nejsou v množině X všechny uzly grafu G , opakuj:
 - Najdi uzel $w \in V \setminus X$ s minimální hodnotou $D(w)$
 - Přidej uzel w do množiny X
 - Pro každý uzel u , sousedící s uzlem w
(za nutného předpokladu, že $u \notin X$) proveď:
 - Hodnota $D(u)$ je minimum ze stávající hodnoty a $D(w) + \textit{ohodnocení hrany } (w, u)$, tedy:

$$D(u) = \min (D(w) + |(w, u)|$$

Pokud Vám výše popsaný postup je povědomý, není tomu tak náhodou. Dijkstrův algoritmus je totiž rozšířením procházení grafu do šířky (BFS), přičemž místo datové struktury *fronta* zde využíváme prioritní fronty. Prioritní fronta funguje jako fronta, ve které mohou předbíhat uzly s nejkratší aktuální vzdáleností od zdroje.

Algoritmus tedy systematicky postupuje od výchozího uzlu k cílové destinaci a aktualizuje vzdálenosti. Výstupem algoritmu je pak délka nejkratší cesty z výchozího uzlu do všech ostatních uzlů.

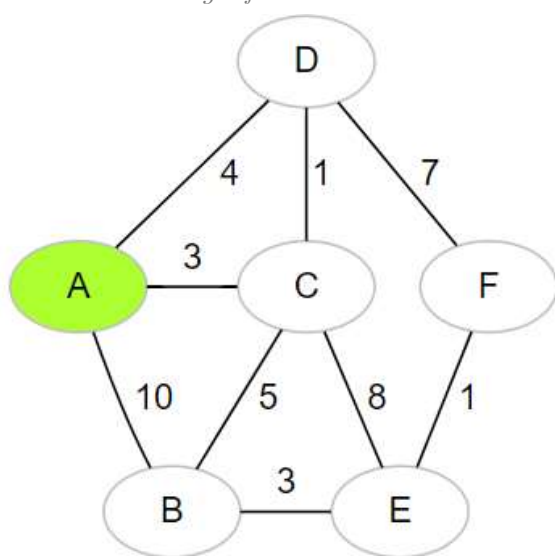
Pro všechny potomky Dijkstrův alg ověřuje:

$$l_z + h < l_p,$$

kde:

- l_z vzdálenost zpracovávaného
- h délka hrany zpracovávaného a potomka
- l_p vzdálenost potomka

~ Příklad 1: Vyhledejte nejkratší cestu z počátečního uzlu A do všech ostatních uzlů
~ ohodnoceného grafu G.



Úmluva: Jako množinu X označme množinu nenavštívených vrcholů.

Řešení: Vstupem algoritmu je graf G . Počáteční uzel $s = A$

Kroky zaznamenáme do tabulky:

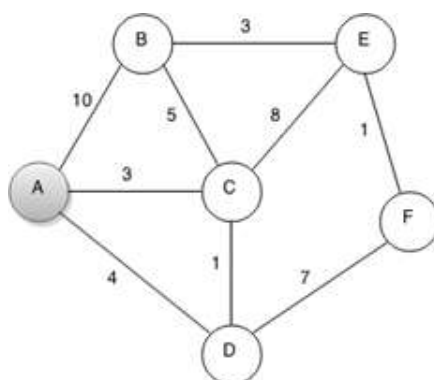
Každý řádek tabulky představuje jeden krok algoritmu. Sloupce představují uzly grafu G a množina X . Jednotlivé buňky pak obsahují hodnotu $D(u)$ odpovídajícího uzlu u ve sloupci.

A	B	C	D	E	F	množina X
0	10, A	3, A	4, A	∞	∞	A
-	8, C	-	4, A	11, C	∞	A, C
-	8, C	-	-	11, C	11, D	A, C, D
-	-	-	-	11, C	11, D	A, C, D, B
-	-	-	-	-	11, D	A, C, D, B, E
-	-	-	-	-	-	A, C, D, B, E, F

Výsledek

A	B	C	D	E	F
0	8, C	3, A	4, A	11, C	11, D

Příklad 2: Vyhledejte nejkratší cestu z počátečního uzlu A do všech ostatních uzlů ohodnoceného grafu G .



Řešení: Použijme Dijkstrův algoritmus:

A	B	C	D	E	F	množ. X
0	10, A	3, A	4, A	∞	∞	A
-	8, C	-	4, A	11, C	∞	A,C
-	8, C	-	-	11, C	11, D	A,C,D
-	-	-	-	11, C	11, D	A,C,D,B
-	-	-	-	-	11, D	A,C,D,B,E
-	-	-	-	-	-	A,C,D,B,E

Výsledek tedy bude:

A	B	C	D	E	F
0	8, C	3, A	4, A	11, C	11, D

7.10.4 Ukázka Dijkstrova algoritmu v JAVA kódu

```
/**
 * Dijkstruv algoritmus
 * @param d matice deleku, pozice [0 1] = 2 znamena, ze z uzlu 0 vede do uzlu 1
 * hrana delky 2
 * @param from uzle ze ktereho se hledaji nejkratsi cesty
 * @return strom predchudcu (z ciloveho uzlu znaci cestu do uzlu from)
 */
procedure int[] doDijkstra(d, from) {
    // vlozi vsechny prvky do prioritni fronty (radi vzestupne dle vzdale-
    // nosti), uzle from ma vzdalenost 0, vsechny ostatni nekonecno
    Q = InsertAllNodesToTheQueue(d, from)
    CLOSED = {} //uzevrene uzly - prazdna mnozina
    predecessors = new array[d.nodeCount] //pole predchudcu

    while !Q.isEmpty() do
        //vrat uzle v nejnijsi vzdalenosti a odstan jej z fronty
        node = Q.extractMin()
        CLOSED.add(node) //uzavri uzle

        //zkrat vzdalenosti
        for a in Adj(node) do //pro vsechny potomky uzlu
            if !CLOSED.contains(a) //pokud jiz nebyl potomek uzavren
                //pokud se pridanim uzlu vzdalenost potomeka snizila
                if Q[node].distance + d[node][a] < Q[a].distance
                    //zmen prioritu (vzdalenost) uzlu
                    Q[a].distance = Q[node].distance + d[node][a]
                    //zmen predka uzlu a na node
                    predecessors[a] = node

    return predecessors
}
```

```

/**
 * Dijkstruv algoritmus
 * @param d matice delek (Integer.MAX_VALUE pokud hrana mezi uzly neexistuje)
 * @param from uzel ze ktereho se hledaji nejkratsi cesty
 * @return strom predchudcu (z ciloveho uzlu znaci cestu do uzlu from)
 */
public static int[] doDijkstra(int[][] d, int from) {
    Set<Integer> set = new HashSet<Integer>();
    set.add(from);

    boolean[] closed = new boolean[d.length];
    int[] distances = new int[d.length];
    for (int i = 0; i < d.length; i++) {
        if (i != from) {
            distances[i] = Integer.MAX_VALUE;
        } else {
            distances[i] = 0;
        }
    }

    int[] predecessors = new int[d.length];
    predecessors[from] = -1;

    while (!set.isEmpty()) {
        //najdi nejblizsi dosazitelny uzel
        int minDistance = Integer.MAX_VALUE;
        int node = -1;
        for(Integer i : set){
            if(distances[i] < minDistance){
                minDistance = distances[i];
                node = i;
            }
        }

        set.remove(node);
        closed[node] = true;

        //zkrat vzdalenosti
        for (int i = 0; i < d.length; i++) {
            //existuje tam hrana
            if (d[node][i] != Integer.MAX_VALUE) {
                if (!closed[i]) {
                    //cesta se zkrati
                    if (distances[node] + d[node][i] < distances[i]) {
                        distances[i] = distances[node] + d[node][i];
                        predecessors[i] = node;
                        set.add(i); // prida uzel mezi kandidaty, pokud je jiz
obsazen, nic se nestane
                    }
                }
            }
        }
    }
    return predecessors;
}

```

7.11 Cvičení

1. Nakreslete graf železniční sítě, popř. její části v okolí svého bydliště
2. Nakreslete graf sítě PID (metro, tramvaj, bus), popř. její části v okolí školy.
3. Reprezentujte graf u úloh 1) a 2) libovolnou maticí.
4. Nakreslete pravidelný graf stupně 4, který není úplným grafem o 5 vrcholech
5. Nakreslete pravidelný graf stupně 5, který není úplným grafem o 6 vrcholech
6. Který kompletní graf je současně cyklem?
7. Nechť G je pravidelný graf stupně k o n vrcholech. Určete jeho počet hran.
8. Může být kružnice vlastních podgrafem jiné kružnice?
9. Určete počet hrad kompletního grafu o n vrcholech.
10. Pro libovolnou šachovou figuru kromě pěšce nakreslete graf, jehož vrcholy jsou pole šachovnice a v němž jsou dva vrcholy spojeny hranou právě tehdy, je-li možné přejít z jednoho z nich do druhého jedním tahem šachové figury.

8 Stavový diagram, stavový prostor a jeho prohledávání

8.1 Stavový diagram (UML state machine)

= též diagram stavů

UML state machine zobrazuje životní cyklus dané entity. Jedná se o UML diagram, který znázorňuje chování IS, resp. stavy, ve kterých se daná entita IS může nacházet a přechody mezi stavy

- Způsob standardizovaného grafického zápisu vývoje IS s konečným počtem stavů (např. *konečný automat*)
- Popisuje chování systémů, které vyjadřují stavy určitého objektu a přechody mezi nimi (tzv. přechodová funkce)
- Modeluje dynamické chování reaktivního objektu¹¹
- *UML state machine* poskytuje sadu elementů pro popis konkrétního systému (vyjádřen průchodem stavy, řízen vnějším vstupem)
- Umožňuje přehledný náhled na entity (jakého stavu kdy jaká entita nebude)
- Lze vidět funkčnost IS v reálném čase (změnu stavu iniciuje událost /uživatel, čas/)

Podstatou je, že entita se vždy nachází v nějakém stavu a tohoto stavu může dosáhnout za určitých okolností (např. *dokument je vytisknutý, pokud tisk proběhl v pořádku*).

8.1.1 Základní prvky stavového diagramu:

- Stavy
 - Přechody
-

¹¹ Reaktivní objekt je objekt (v širším slova smyslu), který reaguje na vnější události a má zajímavé chování (třída, případ užití, podsystém).

- Události

8.1.1.1 Stavy

- Určen hodnotami atributů, relacemi s dalšími objekty a aktuální aktivitou
- Mohou být složené (obsahovat vnořené stavové automaty se sekvenčním i paralelním během, modelování synchronizace, komunikace, historie)

Stav může obsahovat:

- Vstupní akce (provádí se automaticky při vstupu do stavu)
- Výstupní akce (provádí se automaticky při opuštění stavu)
- Interní přechody
- Pozdržené události
- Interní aktivity

8.1.1.2 Přechody

- viz 8.1.4.4 Transition (přechod), str. 185
- Syntaxe: `Trigger[Guard]/Effect`
- Sémantika: Při výskytu události, je-li podmínka splněna, vykonaj akci a přejdi do následného stavu.

8.1.1.3 Události

- Rozlišujeme události volání, signální události, události změny a časové události

8.1.2 Modelování stavového diagramu

- Objekty mění stav jako důsledek přechozí události
- UML state machine d. rozlišuje pouze stavy, jejichž diferenciaci má pro modelování chování objektu smysl (je mezi nimi sémanticky významný rozdíl)
- UML state machine d. modeluje chování systému

8.1.3 Souvislost stavového diagramu a diagramu aktivit

Společné rysy

- Podobná grafická notace (syntaxe)

Odlišnosti

- Diagramy aktivit se užívají pro modelování obchodních procesů (účast několika objektů), šipky propojují jednotlivé aktivity
- Stavové diagramy se užívají spíše k modelování životního cyklu jednoho reaktivního objektu, šipky propojují jednotlivé statické stavy (Zimmerová, 2008)

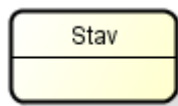
8.1.4 Grafická notace

UML state machine diagram se skládá z několika prvků:

1. State (stav)
2. Initial (počáteční bod)
3. Final (konečný bod)
4. Transition (přechod)
5. Submachine state
6. Choice (volba)
7. Junction (uzel)

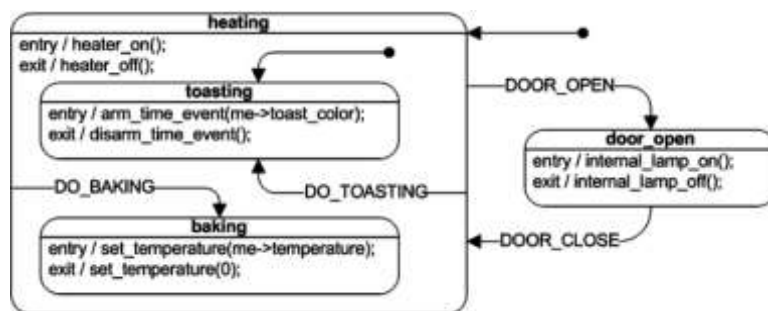
8.1.4.1 State (stav)

- Základní prvek Stavového diagramu
- Reprezentuje jeden daný stav (ve kterém se daná entita může nacházet)



OBRÁZEK 8-1: STATE (ČÁPKA, 2019)

- Stav může v sobě obsahovat další stavy (→ složený stav)



OBRÁZEK 8-2: SLOŽENÝ STAV

- U stavu lze definovat akce, např.:
 - `entry` akce, která nastane při vstupu do tohoto stavu
 - `do` akce, které nastanou v průběhu stavu
 - `exit` akce, které nastanou při opuštění stavu
- tyto akce se dle notace zapisují do dolní části stavu, a to jednoduše slovně (např. `entry/ načíst nastavení`)

8.1.4.2 Initial (počáteční bod)

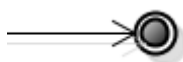
- Počáteční bod (začíná zde přechod do stavu). Odsud se obvykle přesouvá na první počáteční stav



OBRÁZEK 8-3: INITIAL (ČÁPKA, 2019)

8.1.4.3 Final (konečný bod)

- Změny stavů mohou (ale nemusí) někdy skončit.
- Jakmile stav přejde do bodu final, již nenastávají žádné další jeho změny.
- Bodů final může být v diagramu libovolný počet.



OBRÁZEK 8-4: FINAL (ČÁPKA, 2019)

8.1.4.4 Transition (přechod)

- Vazba mezi dvěma stavy



OBRÁZEK 8-5: TRANSITION (ČÁPKA, 2019)

- Šipka směřuje k novému stavu
- Stav může obsahovat i přechod sám na sebe

Přechod (transition) je relace ve stavovém stroji mezi dvěma stavy, kde objekt v prvním stavu přejde do druhého stavu tehdy, když nastane specifikovaná událost (event) a jsou splněny specifikované podmínky (guards), přičemž se provede specifikovaný efekt (effect). Přechod je nepřerušitelný. Říká se, že přechod je odpálen (transition is fire). Přechod může mít jeden nebo více zdrojových stavů a jeden nebo více cílových stavů.

Vazba může mít následující vlastnosti (zapisují se nad nic v daném formátu)

`Trigger[Guard]/Effect`

- Trigger (příčina) – proč přechod nastal, např. `registrace byla potvrzena` – může vyvolat u entity `User` před ze stavu `Čeká na ověření` na stav `Ověřen`
- Guard (podmínka) – podmínka pro přechod na daný stav. Zapisuje se do hranatých závorek, tedy např. `[odkaz s id 256654221 byl otevřen]`. Tedy uživatel otevřel odkaz s jedinečným id, který mu byl zaslán ověřovacím e-mailem
- Effect – akce, která nastane s přechodem na daný stav. Tedy např. `Prava = 2`. Tedy povýšení práv, které pro která je ověřený e-mail premisou.

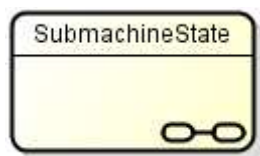
`Ověřen[odkaz s id 256654221]/Prava=2` finální podoba vlastnosti přechodu.

Příklad: startování při automobilových soutěžích

`změnaSemaforu(barva)[barva="zelená", závodník je na řadě]/rozjede` → pokud semafor změní barvu na zelenou a závodník je na řadě, tak se rozjede.

8.1.4.5 Submachine State

- Stav, který se uvnitř skládá ze složité struktury dalších podstavů, ale navenek se tváří jako stav jeden.
- Stav vložené do Submachine State nejsou viditelné a diagram je od nich odstíněn.
- Grafická notace vypadá stejně jako stav, jen má v pravém dolním rohu ikonu brýlí.

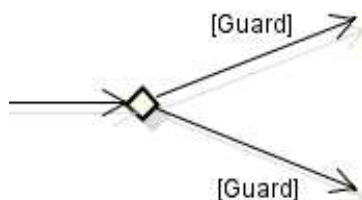


OBRÁZEK 8-6: SUBMACHINE STATE (ČÁPKA, 2019)

Trik spočívá v tom, že když v UML CADu klikneme na ikonu brýlí, otevře se nám další stavový diagram, který reprezentuje *Submachine State*. Tento prvek je tedy pro složité diagramy a nesetkáme se s ním často.

8.1.4.6 Choise (volba)

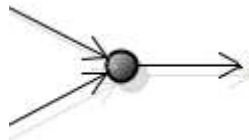
- Pomocí Choice můžeme určit, jakého stavu entita dále nabude.
- Princip je jednoduchý, pouze uvedeme guardy (podmínky) pro jednotlivé větve
- Grafická notace vazby je kosočtverec, do kterého přichází přechod a další z něj vychází.



OBRÁZEK 8-7: CHOISE (ČÁPKA, 2019)

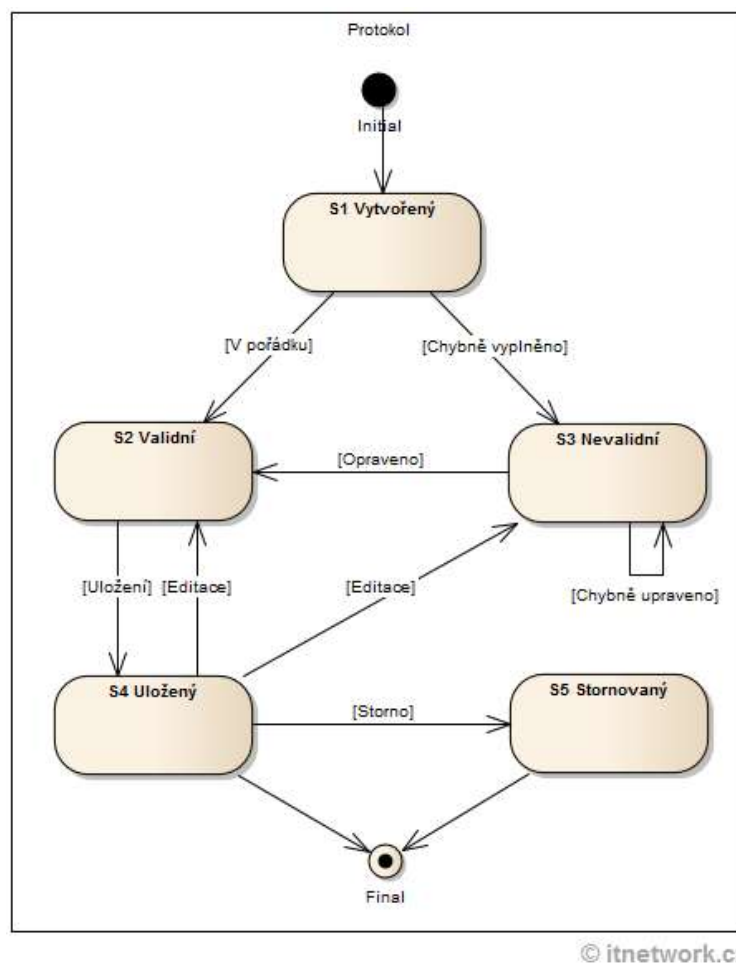
8.1.4.7 Junction (uzel)

- Analogicky jako *choice* umožňoval větvit jednotlivé přechody, *junction* je umožňuje opět sjednotit do jednoho.
- Grafická notace = vyplněný kruh.



OBRÁZEK 8-8: JUNCTION (ČÁPKA, 2019)

- Z uzlu může opět vycházet několik přechodů s *guardem* a může tedy opět plnit funkci větvení.
- Podstatou je, že do něj může narozdíl od *Choice* vcházet více přechodů.



OBRÁZEK 8-9: REKLAMAČNÍ PROTOKOL V UML STATE MACHINE DIAGRAM

Cvičení: Vytvoř stavový diagram pro bezpečnostní dveře (viz obrázek níže). (Dveře se odemknou pouze tehdy, je-li přiložen čip se správnými právy.)



8.2 Stavový diagram – matice přechodu

Každý model se vždy nachází v určitém stavu (stavy se v průběhu času mění). Přechody mezi stavy se mohou zaznamenat do matice – viz kap. 7.2.1 Matice sousednosti, str. 139

8.3 Stavový prostor

Stavovým prostorem rozumíme konfiguraci *diskrétních stavů* sloužících jako výpočetní model. Formálně může být úloha ve stavovém prostoru definována jako čtveřice $[N, A, S, G]$, kde:

- N je množina stavů
- A je množina přechodů mezi stavy
- S je neprázdная podmnožina N obsahující počáteční stavy
- G je neprázdная podmnožina N obsahující cílové stavy

Na procházení stavového prostoru je založena metoda řešení úloh zvaná *Prohledávání stavového prostoru*. S analýzou stavového prostoru souvisí také Teorie grafů.

Počáteční období výzkumu v oblasti umělé inteligence (50. a 60. léta) bylo charakterizováno představou, že několik jednoduchých ale mocných technik umožní vytvořit inteligentní „všeřešící“ programy. Používané techniky byly založeny na vnitřním (strojovém) modelu světa a na schopnosti vytvářet v tomto modelu plán pro řešení dané úlohy. Model světa byl obvykle založen na stavovém prostoru.

Stavový prostor $\mathcal{S}_p = (\mathcal{S}, \Phi)$ je dvojice tvořená:

- Množinou stavů: $\mathcal{S} = \{s\}$
- Množinou operátorů (přechodů mezi stavy): $\Phi = \{\phi\}, s_k = \phi_{ik}(s_i), \forall i, k \in \mathbb{Z}$

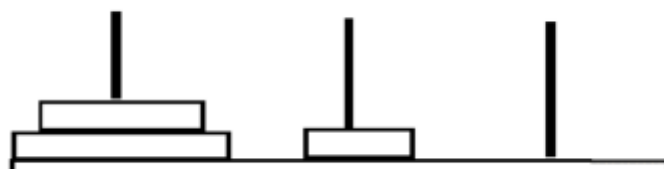
Hanojské věže:

Viz kap. 2.3.3.5 Hanojské věže, str. 58 v kap. Rekurse (Algoritmy).

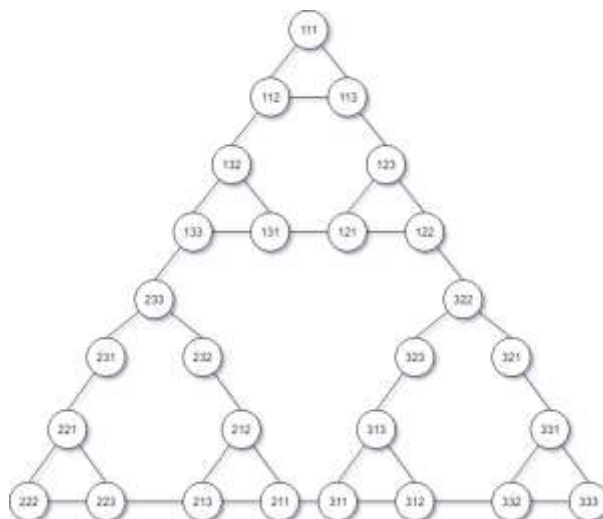
Příkladem stavového prostoru jsou možná rozložení disků u hlavolamu „Hanojské věže“.

Tento hlavolam je tvořen třemi tyčkami, na kterých je navlečeno několik různě velikých disků. Jeden stav tohoto prostoru viz Obrázek 8-10: Hanojské věže.

Velikost stavového prostoru tohoto hlavolamu závisí na počtu disků, pro $n \in \mathbb{N}$ disků je to 3^n (každý z n disků může být na jedné ze tří tyček, *učivo matematiky, téma: kombinatorika, 4. ročník*), pro tři disky je stavový prostor znázorněn na Obrázek 8-11: Stavový prostor Hanojských věží. Použitý formalismus kóduje pozici velikost disku (první zleva je největší disk) a číslem tyčky (1 je levá krajní tyčka).



OBRÁZEK 8-10: HANOJSKÉ VĚŽE (STAV 112)



OBRÁZEK 8-11: STAVOVÝ PROSTOR HANOJSKÝCH VĚŽÍ

V tomto případě máme 3 disky, tedy $n = 3 \Rightarrow 3^3 = 27$

Cvičení:

~ Jaký bude velikost stavového prostoru, máme-li na věžích: (a) 3 disky, (b) 5 disků,
 ~ (c) 15 disků, (d) 30 disků?

Operátory se odvodí na základě pravidla, že se postupně přesouvají disky mezi tyčkami, přičemž:

- může se přesouvat vždy jeden disk,
- větší disk se nelze položit na menší.

Celkem je k dispozici 6 operátorů: $\phi_{12}, \phi_{13}, \phi_{21}, \phi_{23}, \phi_{31}, \phi_{32}$, kde ϕ_{ki} znamená přesun volných disk z k na i .

Hlavalam lišák (též Loydova patnáctka)

Hlavalam „lišák“ je jednodušším ekvivalentem Loydovy patnáctky. Loydova patnáctka je sestavena na poli 4x4, Lišák je omezen do pole 3x3.



OBRÁZEK 8-12: LOYDOVA PATNÁCTKA

Stavový prostor obsahuje $9! = 362880$ stavů.

2	8	3
1	6	4
7	ℓ	5

V tomto stavovém prostoru jsou definovány 4 operátory:

- φ_1 : právě tehdy když ℓ není v horním řádku; přesun ℓ nahoru
- φ_2 : právě tehdy když ℓ není v dolním řádku; přesun ℓ dolů
- φ_3 : právě tehdy když ℓ není v levém sloupci; přesun ℓ vlevo
- φ_4 : právě tehdy když ℓ není v pravém sloupci; přesun ℓ vpravo

Úloha ve stavovém prostoru $(\mathcal{S}, \Phi, s_0, G)$ je definována jako stavový prostor, tj.

- množina stavů $\mathcal{S} = \{s\}$
- množina přechodů mezi stavy $\Phi = \{\phi\}$

úloha ve stavovém prostoru pak dále musí obsahovat:

- počáteční stav s_0
- množina koncových stavů $G = \{g\}$.

V případě Hanojské věže je počáteční stav znázorněn vlevo a její (jediný) koncový stav na obrázku vpravo.

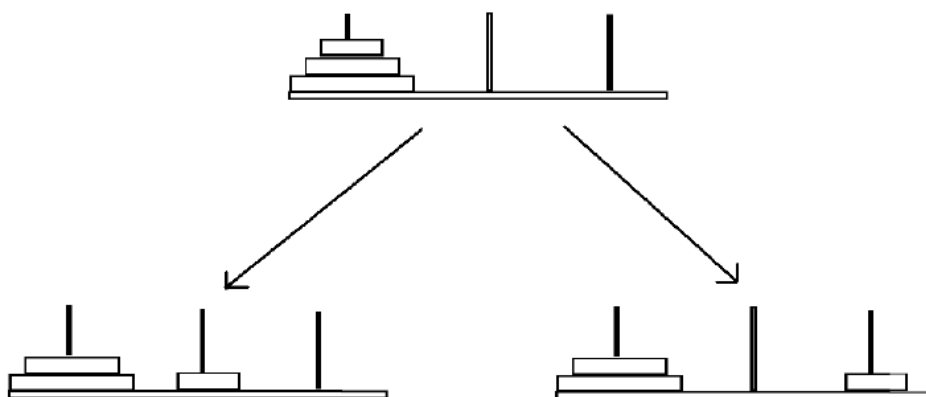


OBRÁZEK 8-13: ÚLOHA VE SP - HANOJSKÉ VĚŽE (VLEVO POČÁTEČNÍ STAV, VPRAVO KONCOVÝ STAV)

Strom řešení úlohy je grafová reprezentace procesu hledání řešení kde uzly odpovídají stavům a hrany odpovídají přechodům mezi stavy daných aplikací operátorů.

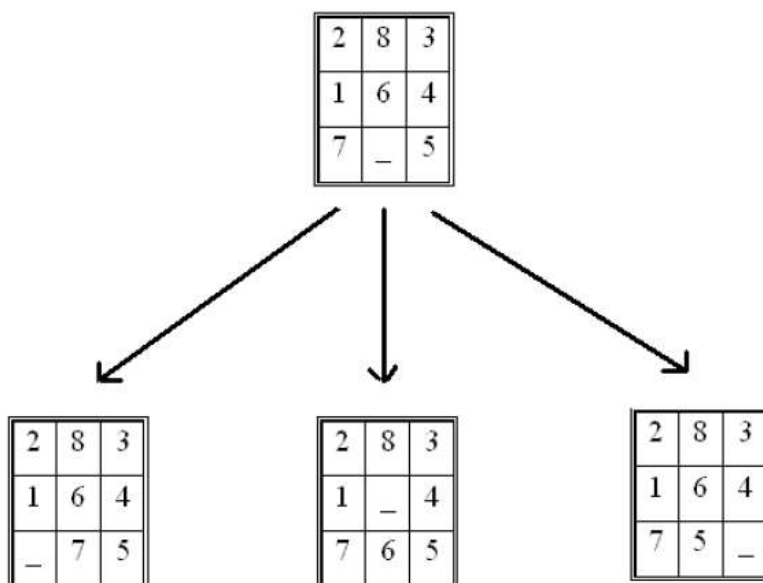
Ve chvíli, kdy pro nějaký stav je použitelných více operátorů se stává klíčovou otázka, který operátor zvolit.

Pro volbu prvního operátoru v případě hanojských věží ukazuje příslušný podstrom viz Obrázek 8-14: Strom řešení úlohy (Hanojské věže).



OBRÁZEK 8-14: STROM ŘEŠENÍ ÚLOHY (HANOJSKÉ VĚŽE)

Pro volbu prvního operátoru v případě hlavolamu lišák ukazuje příslušný podstrom viz Obrázek 8-15: Strom řešení úlohy (Lišák).



OBRÁZEK 8-15: STROM ŘEŠENÍ ÚLOHY (LIŠÁK)

Volba operátoru pro daný stav může být založena na myšlence postupně zkusit aplikovat všechny použitelné operátory, může být založena na nějakém kritériu, které hodnotí vhodnost jednotlivých operátorů, nebo může být volba operátoru náhodná.

8.4 Prohledávání stavového prostoru

MATURITNÍ TÉMA (programové vybavení)

Stavový prostor lze reprezentovat orientovaným grafem $G = (V, E)$. Graf G je stromem.

- Uzly V reprezentují stav
- Hrany E reprezentují přechody mezi stavy.

Jedná se o skupinu metod řešení úloh, které spadají do oblasti AI¹². Princip je založen na vhodném procházení stavů určité *domény* (entity) s účelem nalezení požadovaného

¹² Artificial intelligence (umělá inteligence)

stavu, tedy nalezení přijatelné cesty v orientovaném grafu z počátečního uzlu (kořene stromu) k některému listu.

Příklad

Jsou dány dvě nádoby, větší A o obsahu a litru, a menší B obsahu b litrů:

- na začátku jsou obě prázdné (= počáteční stav)*
- cílem je stav, kdy nádoba A je prázdná a v B je přesně $2 \cdot (a - b)$ litrů vody*

K dispozici je neomezený zdroj vody a nádoby nemají označené míry.

Naplňte nádobu B (= najděte posloupnost akcí takovou, že bude splněn daný cíl).

Efektivní řešení problému \Leftrightarrow zvolení vhodné reprezentace

Stavy jsou v drtivé většině případů generovány v průběhu algoritmu (nejsou předpočítány). Nechť tedy máme úlohu ve stavovém prostoru (S, Φ, s_0, G) , kde S je množina stavů, Φ je množina operátorů, s_0 je počáteční stav a G je množina koncových stavů.

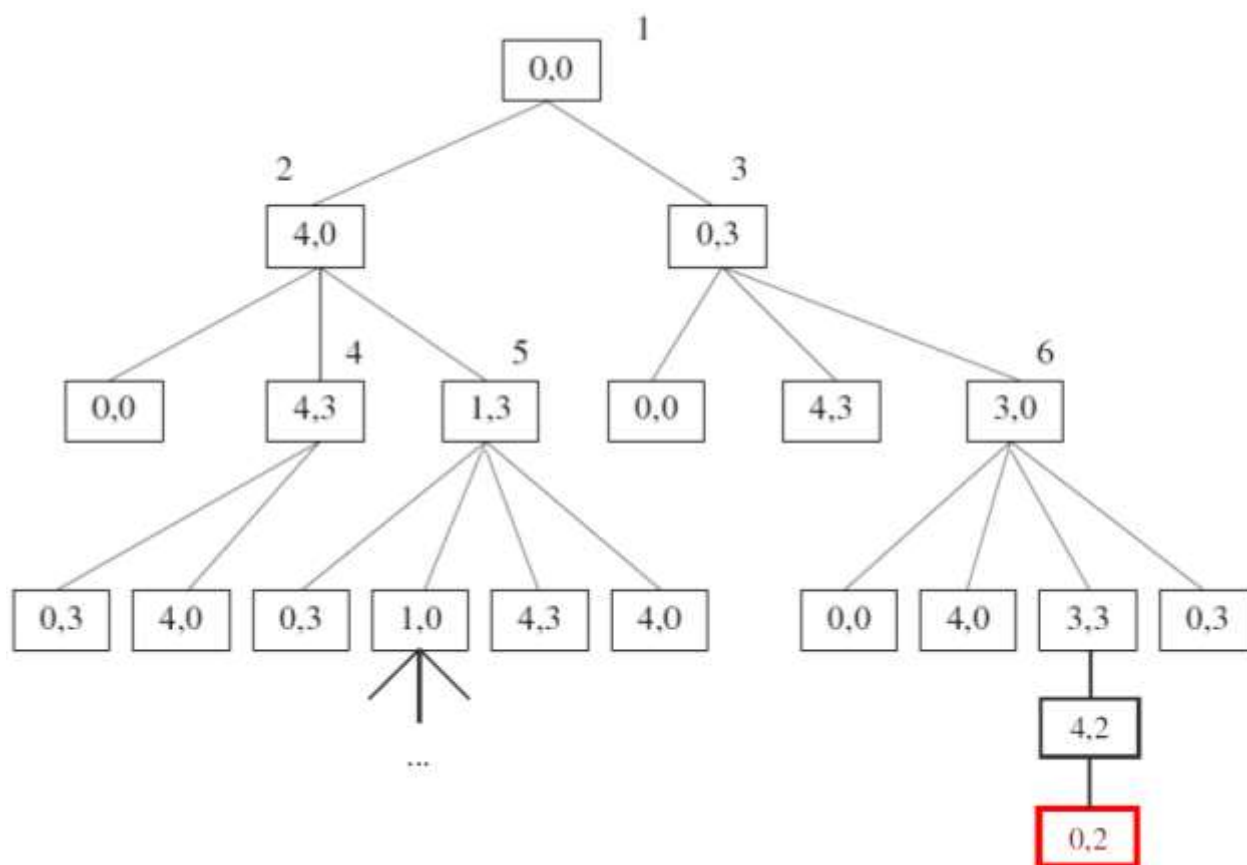
Stav = dvojice (c_A, c_B) , množství vody v nádobách A, B

- Počáteční stav: $(0, 0)$
- Množina koncových stavů: $\{(0, 2a - b)\}$.

Pozn.: Množina koncových stavů obsahuje jeden prvek.

Operátory

- Vylití nádoby A : $(c_A, c_B) \rightarrow (0, c_B)$
- Vylití nádoby B : $(c_A, c_B) \rightarrow (c_A, 0)$
- Naplnění nádoby A, B :
- Přelití A do B : $(c_A, c_B) \rightarrow (\max(c_A - (b - c_B), 0); \min(c_A + c_B, b))$
- Příp. další operátory



OBRÁZEK 8-16: GRAFOVÁ REPREZENTACE PLNĚNÍ NÁDOB

Metody prohledávání lze rozdělit do 3 základních skupin:

1. Slepé

- úplné prohledávání nevyužívající žádné dodatečné informace
- zde postupně aplikujeme všechny použitelné operátory
- *příklad:* BFS, DFS, uniform-cost, iterativní prohlubování, ...

2. Heuristické

- úplné nebo částečné prohledávání využívající hodnocení zvolené cesty
- Zde operátory vybíráme na základě daného kritéria
- *příklad:* beam-search, hill-climbing, best-first search, algoritmus A*,

3. Náhodné

- volíme operátory náhodně

Způsob hodnocení metod

Různé metody jsou pro rozdílná procházení jinak efektivní. Pro správnou volbu požadované metody je nezbytné umět zhodnotit její *účinnost*. Existují 3 základní vlastnosti, podle kterých lze metody hodnotit:

1. Časová složitost

- Minimální/maximální/průměrný čas potřebný k vyřešení úlohy danou metodou
- Čas je přímo závislý na výpočetním výkonu
- V praxi se často reprezentuje např. jako počet prozkoumaných stavů v daném čase (objektivnější hodnota)

2. Prostorová složitost

- Minimální/maximální/průměrné množství operační paměti, potřebné k vyřešení úlohy danou metodou
- Nezávislost na platformě \Rightarrow náhrada údaje o počtu MB \rightarrow počet stavů současně uchovaných v paměti

3. Kvalita získaných výsledků

- Hodnota, která udává výpověď o tom, zda je daná metoda:
 - i. *úplná* (tj. vždy, když existuje řešení, tak jej nalezne),
 - ii. *optimální* (tj. nalezené řešení je nejlepší ze všech),apod.

8.4.1 Slepé metody prohledávání stavového prostoru

= Neinformované metody

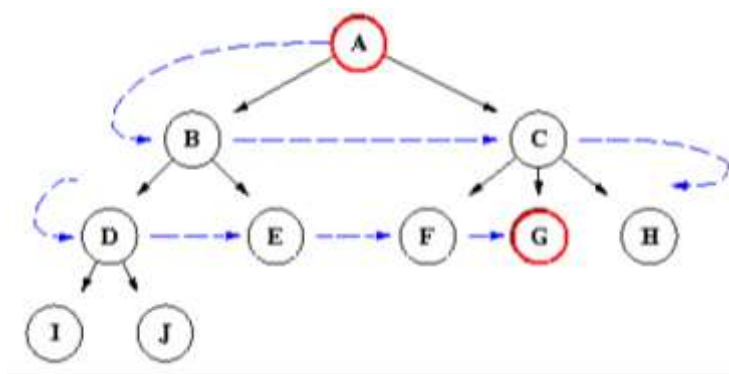
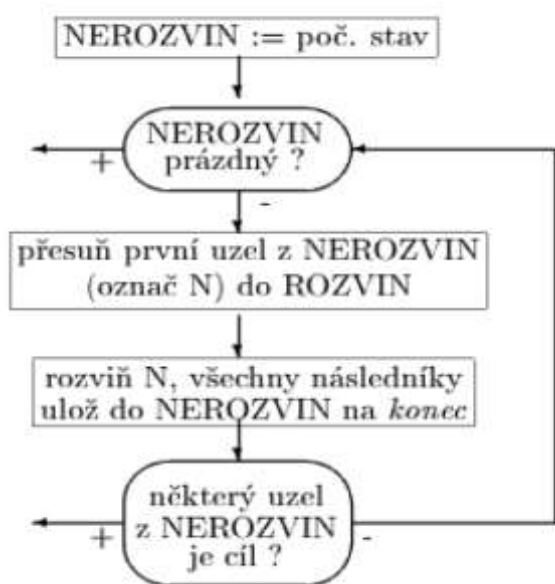
- Nemají k dispozici žádné vhodné znalosti o stavovém prostoru, které by mohly urychlit cestu k cíli (nejsou informované 😊)
- Systematické procházení všech uzlů

- Jednotlivé algoritmy se od sebe liší *jen* způsobem, jakým systematické procházení provádějí

8.4.1.1 Prohledávání do šířky (BFS)

= breadth-first search

- Následníci vybírání pro expanzi ze seznamu typu *fronta*, viz kap. 6.2 Fronta (*Queue*), str. 119
- Postupně se prochází strom řešení po vrstvách (prohledáme veškeré uzly, které mají nižší hloubku, než je hloubka *koncového stavu*)
- Každý uzel je navštíven nejvýše *jednou*.
- Vždy je nalezeno *optimální* řešení (koncový stav s nemenší hloubkou)

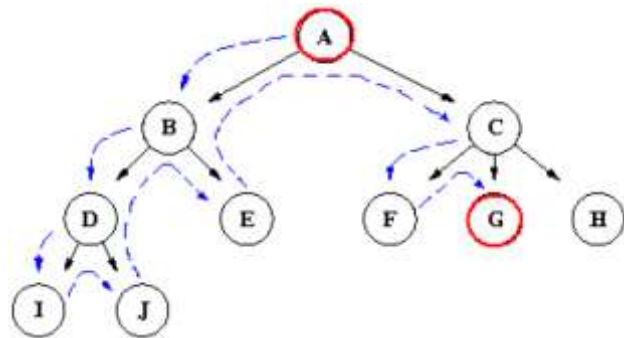


OBRÁZEK 8-17: SCHÉMA ALGORITMU BFS

<<< OBRÁZEK 8-18: ALGORITMUS BFS

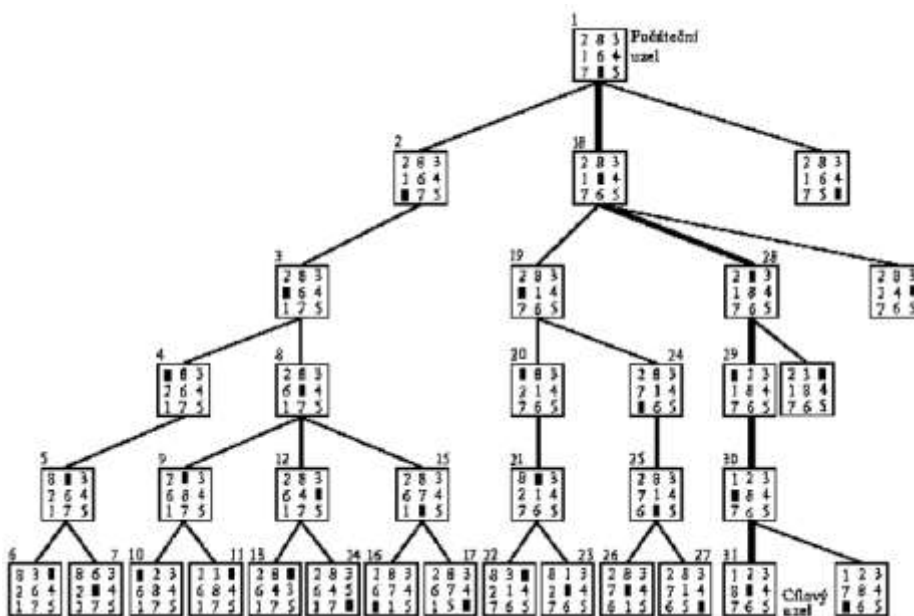
krok	NEROZVIN	ROZVIN
1	A	
2	B, C	A
3	C, D, E	A, B
4	D, E, F, G, H	A, B, C

- V úlohách s konečným stavovým prostorem a s jedním koncovým stavem nalezneme stejné řešení jako u BFS



OBRÁZEK 8-21: SCHÉMA ALGORIMTU DFS

<<< OBRÁZEK 8-20: ALGORITMUS DFS



OBRÁZEK 8-22: LIŠÁK (DFS)

8.4.1.3 Prohledávání se stejnou cenou (uniform cost)

- V každém kroku expanduje uzel, která má nejnižší cenu cesty od počátku
- Jde o zobecnění slepého BFS
- BFS je de facto prohledávání s *jednotnou* cenou (cena = hloubka uzlu)

8.4.1.4 Prohledávání do hloubky s omezením (depth-limited search)

- Omezené DFS se od klasického liší v tom, že je dána maximální hloubka, do které se prohledává. Uzly této hloubky se již dále nerozvíjejí.
- *Užití:* Eliminace problematiky DFS prohledávání u stromů s nekonečnou větví

8.4.1.5 Iterativní prohlubování (iterative deepening search)

- De facto depth-limited search s rozdílem, že maximální hloubka není pevně stanovená (\Rightarrow postupně se zvětšuje maximální hloubka)
- Nejprve se prohledává (do hloubky) stavový prostor hloubky 1, pak do hloubky 2, pak do hloubky 3, atd.
- Iterative deepening search dokáže odstranit jednak problém nekonečných větví, jednak též problém klasického DFS (nalezne optimální řešení, pokud existuje)

8.4.2 Heuristické metody prohledávání stavového prostoru

= Informované metody

- Mají znalosti o stavovém prostoru (umožňují jim odhadnout délku cesty koncového stavu od *aktuálníhoho*)
- Odhad reprezentuje tzv. *heuristická funkce* $h(n)$: Čím nižší hodnoty $h(n)$ nabývá, tím je vyšší pravděpodobnost, že cesta k řešení povede skrze stav n
- Heuristickou funkci dodává člověk (na základě znalostí), informované metody jsou na heuristické funkci kriticky závislé
- Čím je heuristika (kritérium) lepší, tím rychleji a s menším zatížením paměti dojde k nalezení řešení.

Heuristická funkce je funkce, která každému stavu s přiřadí číslo, které vyjadřuje jeho kvalitu z hlediska řešení úlohy.

Vhodnost operátorů (tedy kvalita následníka) může být v dané úloze definována různě. Různé podoby heuristiky mají vliv na volbu operátorů, a tedy také na celkové hodnocení metod (časová složitost, prostorová složitost, kvalita získaných výsledků).

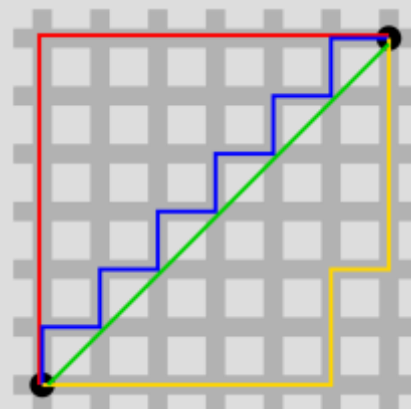
Poznámka (Manhattanská metrika):

těž newyorská metrika (obojí dle pravoúhlého systému ulic na Manhattanu v New Yorku) je metrika na množině \mathbb{R}^n definovaná vztahem:

$$\rho(x, y) = |x_1 - y_1| + |x_2 - y_2| + \dots + |x_n - y_n|$$

Tato metrika odpovídá představě nejkratší vzdálenosti, kterou musí urazit automobil (v populární verzi *vůz newyorské taxislужby*) při cestě z jedné křižovatky na jinou – předpokládáme-li, že systém ulic je pravoúhlý.

Vzdálenost mezi křižovatkami je stejná, ať zvolíme červenou, modrou nebo žlutou trasu. Zelená čára naznačuje způsob, jakým měří vzdálenosti Euklidovská metrika



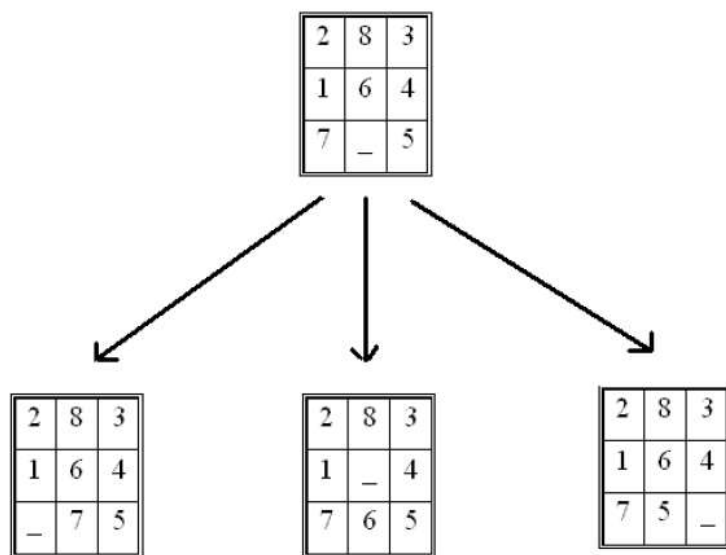
OBRÁZEK 8-23: MANHATTANSKÁ METRIKA

Vraťme se k hlavolamu lišák, kde lze jako možné heuristiky uvažovat:

- H1: počet chybně umístěných číslic
- H2: Manhattan vzdálenost od koncového stavu

Význam heuristiky *H1* – zřejmý.

Heuristika *H2* vyjadřuje kolik tahů ($\leftrightarrow - \updownarrow$) je každá chybně umístěná číslice vzdálena od své správné pozice.



OBRÁZEK 8-24: HLAVOLAM LIŠÁK

Podíváme-li se na volbu operátoru pro první tah dle obrázku Obrázek 8-24, pak kvalitu jednotlivých stavů (1–4) můžeme znázornit do tabulky:

	stav 1	stav 2	stav 3	stav 4
H1	4	5	3	5
H2	5	5	4	7

Obě heuristiky $H1, H2$ pak vyhodnotí jako nejlepší tah $1 \rightarrow 3$.

Pro další tah se budou heuristiky lišit.

	stav 3	stav 6	stav 7	stav 8
H1	3	3	3	4
H2	4	5	3	5

Zatímco heuristika $H1$ vyhodnotí tahy $3 \rightarrow 6$ a $3 \rightarrow 7$ jako stejně kvalitní, heuristika $H2$ vyhodnotí jako nejlepší tah pouze $3 \rightarrow 7$, což skutečně vede k nejrychlejšímu řešení úlohy.

Def.: Algoritmus heuristického prohledávání se nazývá přípustný, jestliže pro libovolně ohodnocený graf ukončí svoji činnost nalezením optimální cesty k cíli.

V: Jestliže heuristická funkce každému uzlu přiřadí hodnotu, která je nezáporným dolním odhadem skutečné ceny, pak algoritmus je přípustný (a příslušná funkce se nazývá přípustná).

Def.: Algoritmus využívající přípustnou heuristickou funkci H_1 je informovanější než algoritmus využívající přípustnou heuristickou funkci H_2 , právě když:

$$\forall s \in S: H_1(s) \geq H_2(s) \wedge \exists s_i \in S: H_1(s_i) > H_2(s_i)$$

O heuristice H_1 pak řekneme že je dominuje heuristice H_2 .

8.4.2.1 Paprskové prohledávání (beam search)

Heuristické prohledávání do šířku (heuristika = odhad vzdálenosti ke koncovému stavu)

OBRÁZEK 8-25: ALGORITMUS BEAM SEARCH

>>>

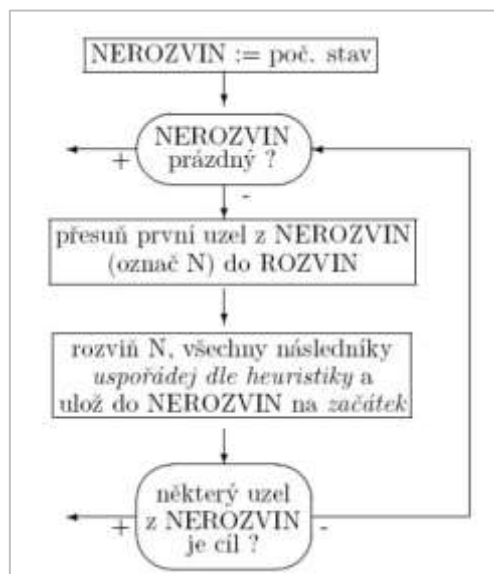
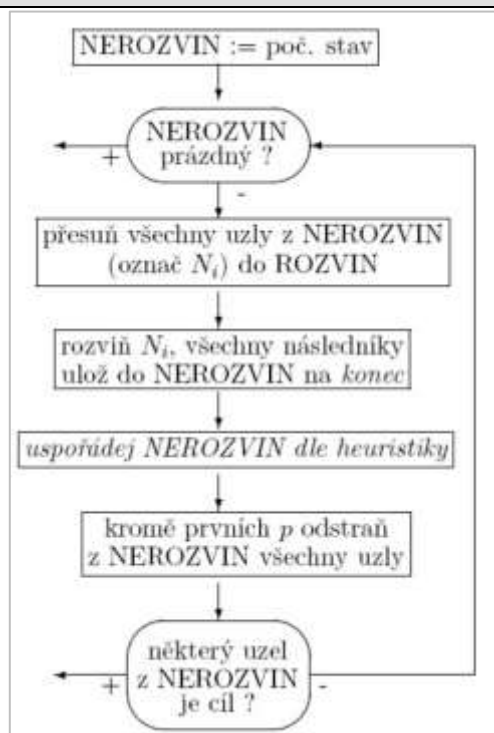
8.4.2.2 Gradientní prohledávání globální (hill climbing)

= též horolezecký algoritmus

Heuristické prohledávání do hloubky (heuristika = odhad vzdálenosti ke koncovému stavu)

Následníky rozvinutého uzlu nejprve uspořádáme dle heuristiky a pak zařadíme do zásobníku.

OBRÁZEK 8-26: GRADIENTNÍ PROHLEDÁVÁNÍ GLOBÁLNÍ



8.4.2.3 Gradientní prohledávání lokální

Vychází z globálního gradientního prohledávání (= heuristické prohl. do hloubky)

Založena na faktu, že pro daný stav volíme dle heuristiky nejlepšího následníka a pracujeme pouze s ním (lokálně). Seznam `NEROZVIN` je jednoprvkový. Lokální gradientní prohledávání hledá *pouze lepší* uzel, než je ten současný, což může vyvolat řadu dilemat:

- *Uváznutí v lokálním extrému* (na listu) – žádný následník (syn) není lepší než původní uzel (nevede k řešení problému), nebo následník vůbec neexistuje
- *Tzv. plošinové dilema* – rozvíjený uzel i jeho následníci jsou stejně kvalitní (není zřejmé kterým směrem pokračovat)

Dilema uváznutí v lokálním extrému ilustruje hlavolam lišák. Ve smyslu heuristiky *H1* (viz výše) není lepší žádný následný stav (viz Obrázek 8-27: Lokální extrém hlavolamu lišák). Prohledávání by tedy umřelo 😞.



a	2	b
8		4
c	6	d

OBRÁZEK 8-27: LOKÁLNÍ
EXTRÉM HLAVOLAMU LIŠÁK

OBRÁZEK 8-28: ALGORITMUS LOKÁLNÍHO GRADIENTNÍHO PROHLEDÁVÁNÍ

8.4.2.4 Uspořádané prohledávání (best-first search)

Úplné heuristické prohledávání do hloubky (doplnění gradientního prohledávání o paměť).

- Prohledávání do hloubky (DFS) = vždy pokračuje v hledání v nejvzdálenějším dostupném stavu
- Prohledávání do šířky (BFS) = vždy pokračuje naopak z jednoho k počátku nejbližších stavů

Uspořádané prohledávání funguje tak, že si z vrcholů k prohledávání vybere ten z hlediska hledaného stavu *nejslibnější*.

Přirozená implementace best-first search aplikuje k ukládání prioritní frontu

- BFS používá obyčejnou frontu
- DFS využívá zásobník (obvykle se jedná v rámci rekurze o zásobník volání)

V porovnání algoritmy DFS a BFS je best-first search algoritmem *informovaným*, neboť se při uspořádávání vrcholů opírá vstupní data.

Vstupní data mohou obsahovat:

- známou cenu vrcholů nebo hran
- heuristiku, kterou se odhaduje jejich slibnost.

Obdobnou myšlenku má nebo rozvíjí řada specializovanějších vyhledávacích algoritmů, například Dijkstrův algoritmus, A* nebo paprskové prohledávání. Naopak na samotný algoritmus uspořádaného prohledávání se lze dívat jako na rozšíření gradientního prohledávání o paměť.

8.4.2.5 Algoritmus A*

= a-star

- vyhledávání optimálních cest v kladně ohodnocených grafech.
- Vytvořen l.p. 1968 Peterem Hartem, Nilsem Nilssonem a Bertramem Raphaellem.

- Používá stejné principy jako *Dijkstrův algoritmus*, ale přidává navíc heuristický prvek.

Hlubší podrobnosti o A^ algoritmu a jeho činnosti viz elektronická podpora v Moodle nebo v příslušné literatuře.*

9 Kódování, úvod do kryptografie a hashovací funkce

Jaký je rozdíl mezi kódováním a šifrováním?

Kódování = metoda, jak upravit přenášená data pro dané přenosové médium, tak, abychom jej využili co nejefektivněji.

Šifrování = metoda, jak přenášená data upravit (zašifrovat) tak, aby je třetí osoba nemohla rozluštit ani pokud je schopná data v pořádku přijmout (zná použité kódování)

Znaková sada je posloupnost znaků.

Kódování znaků (též znakový kód) je reprezentace znakové sady nějakým (číselným) kódem.

Kód je množina symbolů, kterými vyjadřujeme jednotlivé stavy systému.

Zatímco znaková sada je jednoduchý *souhrn (uspořádání) znaků*, kódování znaků definuje navíc i přiřazení kódů jednotlivým znakům.

Příkladem budiž např. Morseova abeceda:

Morseova abeceda kóduje grafémy latinky, číslice a další symboly / znaky pomocí sérií dlouhých a krátkých stisků telegrafního klíče, nebo kombinace dírek na děrném štítku nebo děrné pásce

A	..-.	akát	M	--	mává
B	-...-	blýskavice	N	-. -	nástup
C	-.-.-	cilovnici	O	---	ó náš pán
D	-.-.	dálava	P	..-.	papírnici
E	...	erb	Q	--..	kvůli orkán
F	..-.-	filipiny	R	..-.	rarášek
G	...-	Grónská zem	S	...	sekera
H	Hrachovina	T	-	trám
CH	----	chléb nám dává	U	..-	učený
I	..	ibis	V	...-	vyučeny
J	..-.-	jasmín bílý	W	..-.	vagon klád
K	-. -.	krákorá	X	..-.-	xénokrátés
L	..-.	lupíneček	Y	..-.-	ý se ztrácí
			Z	--..	známá žena
1-		6	..-.-	
2-		7	..-.-	
3-		8-	
4-		9-	
5-		0-	

OBRÁZEK 9-1: TABULKA MORSEOVY ABECEDY

Příklad 1: Kóduj

- ⌘ (a) Tallinn je hlavním městem Estonska
- ⌘ (b) Země vycházejícího slunce. Tak nazýváme Japonsko.

Příklad 2: Dekóduj

- ⌘ (a)
- ⌘ (b)

Příklad 3: Mějme kódovací tabulku:

Znak	kód	Znak	kód
A	B	N	U
B	H	O	V
C	Z	P	A
D	*	Q	Y
E	M _x	R	Q
F	L	S	O
G	S	T	&
H	#	U	N
CH	J	V	F
I	W _t	W	D
J	C	X	R
K	1	Y	I
L	@	Z	G
M	X		

· Zapiš jednoduché slovo a zakóduj jej, dle tabulky. Diakritiku zanedbáme. Nechť
 · soused v lavici tvé slovo dekóduje.

Například: Windows Vista = DWtU*VDO FWtO&B

Kódování ve výpočetní technice pak rozumíme reprezentaci znaků nejčastěji celými čísly (např. Unicode).

Množina symbolů může být vyjádřena, kromě tabulky, též dohodnutým systémem pravidel (algoritmem).

Mějme např. systém, který je určen polohou hrací kostky (krychle):

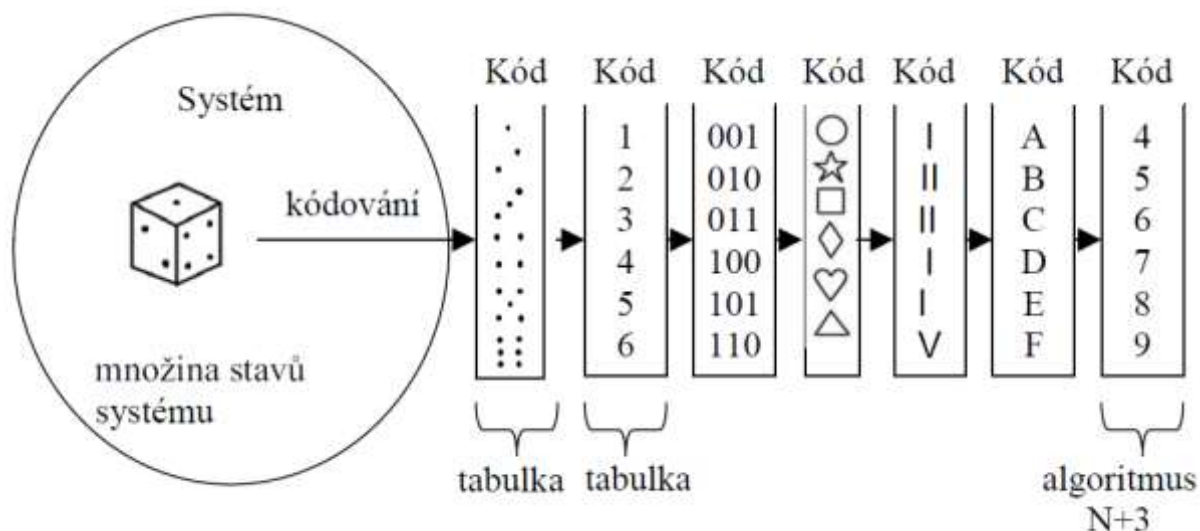
Po hodu se může kostka (systém) nacházet v 6 různých stavech. (poloh kostky)

Poloha kostky je nejčastěji vyjadřována pomocí množiny tečkových symbolů. Padne-li určitý počet teček, pak si jistě tento stav nepoznačíme v tečkovém kódu, ale pomocí arabských číslic 1 až 6, které tvoří jinou množinu symbolů pro vyjádření stavu systému (kostky).

Jiným příkladem množiny symbolů jsou např. různé jednoduché obrázky (○☆□◇♥△), které umožní hrát dětem předškolního věku „Člověče, nezlob se“, aniž by uměli počítat (mohou posouvat své figurky na odpovídající obrázky hrací cesty).

Kódování je činnost, během které převádíme jeden kód na druhý a zpět, a to buď pomocí tabulky nebo vhodným algoritmem.

Vztah mezi systémem, kódem a kódováním na předchozím příkladu ilustruje .



OBRÁZEK 9-2: VZTAH MEZI KÓDOVÁNÍM, KÓDEM A SYSTÉMEM (MATOUŠEK, 2006)

9.1 Binární kódování

Binární kód je možnost zápisu informace vymezený jako počet bitů, z nichž každý může nabývat právě jednu ze dvou hodnot, stanovených jako 0 a 1.

Pro snadnější zápis čísel se dnes převážně používá byte s délkou slova 8 bitů. Pro výpočet hodnoty binárního zápisu se používá dvojková soustava.

9.1.1 Význam binárního kódování

Binární kód = označení, které je užito tehdy, neví-li člověk, jaká informace je v zápisu hodnot použita (text, obrázek, strojové učení, ...).

Aby mohla být informace uložena a později opět obnovena, používá se při převodech do binárního kódu vždy nějaké kódování.

- Určuje, jakým způsobem je informace převedena do číselného zápisu a zpět
- Liší se v závislosti na konkrétních datech:
např.: text používá znakovou sadu, kdy každému symbolu odpovídá nějaké číslo.

ASCII Code Chart																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

OBRÁZEK 9-3: ASCII KÓDOVACÍ TABULKA

V kódování ASCII je pro znak písmene A určen kód 41 (v hexadecimální soustavě), a tedy 1000001 v binární soustavě. Analogicky se zpracovávají i další data.

Pro zjednodušení práce s počítačem se používají různé počítačové programy, které dokážou uloženou informaci interpretovat, tj. zprostředkovat člověku. Obrázky se zobrazují prostřednictvím programu označovaného jako prohlížeč obrázků (strojový kód dokáže zobrazit disassembler¹³), text je zobrazován pomocí textového editoru atd.

	0	1
0	NUL	SOH
1	DLE	DC1
2		
3	0	1
4	@	A

¹³ Disassembler je program převádějící strojový kód do symbolického zápisu v assembleru (překladač z nízkoúrovňového jazyka blízkého jazyku procesoru). Je používán k analýze přeložených programů, u nichž není k dispozici jejich zápis v žádném vyšším programovacím jazyce.

Cvičení A: Zapište své křestní jméno bez diakritiky v hexadecimální podobě podle ASCII kódování

Cvičení B: Zapište své křestní jméno bez diakritiky v binární podobě podle ASCII kódování

Cvičení C: Je dán ASCII kód: 074 111 115 101 102 032 090 105 108 118 097 114 v dekadické podobě. Dekódujte

9.1.1.1 *Okno do historie

Binární kód poprvé předvedl v 17. století německý polyhistor, matematik a filosof G. W. Leibniz (integrální počet, moderní logika, analytická filosofie, scholastika). Leibniz se snažil najít systém, který by umožňoval převody slovních spojení na matematické formule. Jeho nápady nebyly brány vážně a nikdo jiný o tuto problematiku nejevil zájem. Jeho teorii (zjednodušení, zredukování života na řadu elementárních problémů) potvrzoval tradiční čínský text *I-ťing* (Čínská kniha proměn, 2. tis. př. Kr.), který je psán v určitém typu binárního kódu (zjednodušeně formulováno). Leibniz vytvořil systém, skládající se z nul a jedniček, nicméně v jeho době tato práce nenašla žádné využití.

V r. 184 představil George Boole ve své publikaci *Matematická analýza pro logiky* algebraický systém logiky, který je dnes znám jako booleova algebra. Systém je založen na binárním přístupu a používá tři operace: AND, OR, NOT. Tento systém však též nenašel praktické využití, dokud si Claude Shannon (tehdy postgraduální student MIT) nevšiml, že booleova algebra, je podobná elektrickému obvodu. V roce 1937 Shannon představil svoji práci, která následně stala základem pro využití binárního kódu v praktických aplikacích jako jsou počítače, elektrické obvody a další.

1875: Émile Baudot: přidal binární řetězec do svého šifrovacího systému, což nakonec vedlo k dnešnímu kódování ASCII

Počátky využití binárního kódování

- 1932: C. E. Wynn-Williams: "Scale of Two" počítač
- 1936: Konrád Zuse: počítač Z1
- 1937: Alan Turing: elektricko-mechanická binární násobička
- 1938: Atanasoff-Berry Computer
- 1939: George Stibitz: realizace Booleovy algebry logickým obvodem za využití elektromechanického relé jako přepínacího prvku

9.1.1.2 *Další formy binárního kódu

Řetězec bitů není jediným typem binárního kódu. Binární systém je obecná soustava, která obsahuje všechny struktury, jejichž výsledek má pouze dvě možnosti: (I/O, pravda/nepravda, ano/ne, ...)

Pa Kua

Pa Kua je sbírka osmi diagramů které v taoistické kultuře zastupují základní principy reality (ohně, voda, země atd.). Lze jej nalézt v mnoha oblastech čínské filosofie, v Tchaj-ti, Feng-šuej, atd.

- *Pa* v názvu znamená 8,
- *Kua* označuje věštebný obrazec.



OBRÁZEK 9-4: SYMBOL JIN A JANG
S OSMI HEXAGRAMY Z I-ŤINGU

Každý z osmi diagramů, zvaných trigramy, se skládá ze tří

čar, z nichž každá je buď přerušená nebo souvislá a reprezentuje jin nebo jang (viz Obrázek 9-4). Dvojice trigramů tvoří 64 hexagramů, které jsou základem Knihy proměn (I-ťing).

Braillovo písmo

Braillovo písmo je druh binárního kódu, který je využíván slepci pro čtení a psaní. Tento systém se skládá z 6 pozic teček, tři v každém sloupci.

Každý bod má dva stavy, vystouplý nebo nevystouplý.

§ Cvičení: S pomocí chytrých zařízení s připojením k internetu запиšte
v Braillově písmu své křestní jméno bez diakritiky. Nepoužívejte
převodníky.



OBRÁZEK 9-5:
SYMBOL PRO A
NEBO I V
BRILLOVĚ PÍSMU

IFÁ Oracle

IFÁ Oracle se používá mezi *Jorubským* obyvatelstvem v Nigérii od roku 7000 př. Kr. Jedná se o 16 hlavních tzv. zpívaných (Odu) a jejich spojením je celkem 256 (Odus) stejně jako v binárním kódu. Kód se zapisuje pomocí linek, dvou linek nebo nul. Oracle používá Babalawos (vysoký kněz) pro konzultace s IFA (zdroj univerzální moudrosti) nebo Orunmila (Orisa, Boží moudrosti), předpovídání budoucnosti a doporučení řešení denních lidských otázek o životě. Systém věštění Ifa byl přidán v roce 2005 do seznamu UNESCO „*Mistrovská díla ústního a nehmotného dědictví lidstva*“.

9.1.2 Binární čísla

Dvojková (binární) soustava je číselná soustava, používající dvě číslice 0 a 1.

Binární číslo je zastupováno posloupností tzv. binárních hodnot, zapisovány jako 0 nebo 1. Taková čísla se používají v digitální technice.

Nejpoužívanější mocniny tvaru n^2

mocnina	hodnota	poznámka
2^2	4	
2^3	8	byte
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^{10}	1 024	1 KiB
2^{20}	1 048 576	1 MiB
2^{30}	1 073 741 824	1 GiB
2^{30}	4 294 967 296	1 GiB
2^{40}	1 099 511 627 776	1 TiB

KiB = Kibibajt; MiB = Mebibajt;

GiB = Gibibajt; TiB = Tebibajt

Konvenční poznámka: Ačkoliv se korektně zkratka předpony kilo zapisuje obecně s malým k, tedy k2; u binární předpony kibi se dle normy ISO/IEC 80000 zapisuje Ki2, s velkým Ki, kde 2 je jednotka veličiny.

Jednoduché převody:

$$n \text{ KiB} = \frac{1000}{1024} n \text{ kB}$$

$$n \text{ kB} = \frac{1024}{1000} n \text{ KiB}$$

$$n \text{ MiB} = \left(\frac{1000}{1024}\right)^2 n \text{ MB}$$

$$n \text{ MB} = \left(\frac{1024}{1000}\right)^2 n \text{ MiB}$$

$$n \text{ GiB} = \left(\frac{1000}{1024}\right)^3 n \text{ GB}$$

$$n \text{ GB} = \left(\frac{1024}{1000}\right)^3 n \text{ GiB}$$

Hexadecimální čísla

HEX	BIN	DEC
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

HEX	BIN	DEC
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

9.1.3 Nejvýznamnější bit (MSB), nejméně významný bit (LSB)

Termínem nejvýznamnější bit (MSB z angl. *Most Significant Bit*) rozumíme bit s nejvyšší hodnotou v binárním vyjádření čísla; v obvyklém dvojkovém zápisu jde o bit nejvíce vlevo.

1	0	0	1	0	1	0	1
---	---	---	---	---	---	---	---

OBRÁZEK 9-6: ZÁPIS ČÍSLA 149 VE DVOJKOVÉ SOUSTAVĚ SE ZVÝRAZNĚNÝM NEJVÝZNAMNĚJŠÍM BITEM

Termínem nejméně významný bit (LSB z angl. *Least Significant Bit*) rozumíme bit, jehož pozice udává hodnotu, která určuje paritu čísla (sudost / lichost).

Těž lze LSB označit jako bit, který je nejvíce vpravo.

1	0	0	1	0	1	0	1
---	---	---	---	---	---	---	---

OBRÁZEK 9-7: ZÁPIS ČÍSLA 149 VE DVOJKOVÉ SOUSTAVĚ SE ZVÝRAZNĚNÝM NEJMÉNĚ VÝZNAMNÝM BITEM

9.1.4 Bitové a booleovské operátory

Rozlišujeme následující bitové operace:

- Logický součin (AND) booleovský operátor bitový operátor
- Logický součet (OR) booleovský operátor bitový operátor
- Exkluzivní součet (XOR) --- bitový operátor
- Bitová negace (NOT) booleovský operátor bitový operátor

Booleovské operátory (AND - &&, OR - ||, NOT - !):

- Oba operandy musí být stejného typu a výsledek je typu `unsigned 1`.

Příklad:

```
■ a = 0xF; // výsledek 1
```

Bitové operátory (XOR - ^, OR - |, AND - &, NOT - ~):

- Oba operandy musí být stejného typu a výsledek je stejného typu jako operandy

Příklad:

```
■ a = 0xFF & 0xF0; // výsledek je 0xF0  
b = 0xFF | 0xF0; // výsledek je 0xFF  
c = 0xFF ^ 0xF0; // výsledek je 0x0F  
d = 0xFF ~ 0xF0; // výsledek je 0x00
```

Logický součin

Logický součin vrací 1 právě tehdy, když jsou oba činitelé jsou rovny 1, jinak 0.

Značí se též jako `AND`, `&&`, `&`

a	b	a AND b
0	1	0
0	0	0
1	1	1
1	0	0

Příklad v programovacím jazyce Java:

```
int a = 0xC1500137;  
int b = 0x84101157;  
int r = a & b;
```

Logický součet

Logický součet vrací 1 právě tehdy, aspoň jeden činitel je roven 1, jinak 0. Značí se též jako `OR`, `||`, `|`

a	b	a AND b
0	1	1
0	0	0
1	1	1
1	0	1

Příklad v programovacím jazyce Java:

```
int a = 0xC1500137;  
int b = 0x84101157;  
int r = a | b;
```

Exkluzivní součet

Exkluzivní součet (někdy *též* Bitová nonekvivalence) vrací 1 právě tehdy, pokud se hodnoty liší, jinak 0. Značí se též jako `XOR`.

a	b	a XOR b
0	1	1
0	0	0
1	1	0
1	0	1

Příklad v programovacím jazyce Java:

```
int a = 0xC1500137;  
int b = 0x84101157;  
int r = a ^ b;
```

Bitová negace

Bitová negace (někdy též doplněk) vrací 1 právě tehdy, pokud původní hodnota byla rovna 0, jinak 1. Značí se též jako `XOR`, `!`.

a	NOT a
0	1
1	0

Příklad v programovacím jazyce Java: `triv`.

Cvičení: `~77 == 178` Dokaž.

Řešení:

$$(77)_{10} = 01001101$$

$$\text{NOT}(01001101) = 10110010$$

$$(10110010)_2 = \mathbf{178} \blacksquare$$

Bitové operace by Vám měli nápadně připomínat výrokovou logiku (Matematika, 1.ročník SŠ)

9.1.5 Bitový posun

Rozlišujeme dvě různé posuvné operátory:

Bitový posun vlevo `<<`

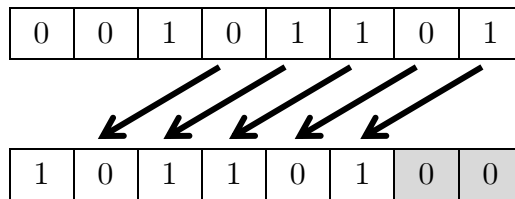
Tímto operátorem jednoduše provedeme posun všech bitů levého operandu o určitý počet míst, který udává hodnota pravého operandu. Při posunu se bity nejvíce nalevo ztrácejí a zleva jsou uvolněná místa doplněna nulami.

Následující příkaz vrátí o dvě místa bitově posunutou hodnotu čísla 45.

```
45 << 2;
```

Postup:

$(45)_{10} = 00101101$



Na místo LSB doplňujeme nulu.

$(10110100)_2 = (180)_{10}$

Bitový posun doleva se často používá k provádění násobení mocninami dvou.

Je totiž znatelně rychlejší než normální násobení. Platí, že $x \ll n == x * 2^n$. Schopnější překladače dnes již samy nahrazují (je-li to možné) obyčejné násobení bitovým posunem.

Bitový posun vpravo >>

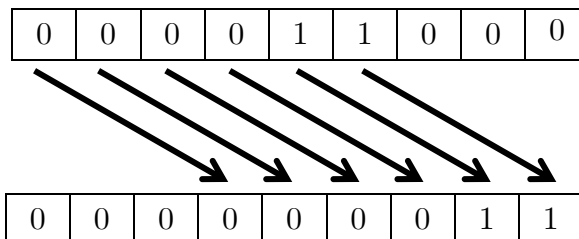
Jedná se o komplementární operaci k bitovému posunu vpravo. V tomto případě se bity zprava ztrácí a zleva jsou doplňovány nulou u neznaménkových typů, nebo znaménkovým bitem u typů znaménkových, což je ale implementačně závislé.

Stejně jako se dal bitový posun doleva použít k násobení, lze zase bitový posun doprava použít k celočíselnému dělení mocninami dvou. $x \gg n == x / 2^n$.

```
48 >> 4;
```

Postup:

$(48)_{10} = 000011000$



Na místo LSB doplňujeme nulu.

$(11)_2 = (3)_{10}$

Bitový posun o konstantu je v hardwaru realizován přepojováním propojovacích vodičů. Bitový posun o hodnoty proměnné zajišťují multiplexory.

9.1.6 Maskování bitů

Místo bitových manipulací je možné použít i maskování bitů známých z jazyka C.

- Efektivní hardwarová realizace
- Malá přehlednost a hodně kódu

Maska definuje, které bity se zachovají, a které bity se odstraní. Jde o úlohy, kdy se aplikuje určitá maska na hodnotu. Toho lze dosáhnout:

- Bitový součin (AND) - pro extrahování podmnožiny bitů v hodnotě

	1	1	1	0	1	1	0	1	[input]
(&)	0	0	1	1	1	1	0	0	[mask]

	0	0	1	0	1	1	0	0	[output]

- Bitový součet (OR) - pro nastavení podmnožiny bitů v hodnotě

	1	1	1	0	1	1	0	1	[input]
(^)	0	0	1	1	1	1	0	0	[mask]

	1	1	0	1	0	0	0	1	[output]

- Bitový exkluzivní součet (XOR) - pro přepnutí podmnožiny bitů do hodnoty
(příklad ukázky je vhodný jako domácí cvičení)



OBRÁZEK 9-8: POUŽITÍ MASKY NA OBRÁZEK

Cvičení: Zapiš podle ASCII tabulky jméno „Petr Bajza“ binárním kódem. Na tento binární kód aplikuj masku, která je tvořena posuvem vpravo o 3 původního binárního kódu a proveď logický součet s maskou. Zapiš výsledný binární kód v dekadické soustavě.

9.2 Bitové pole

9.3 Systémy kódování

9.3.1 Pomocí kódovacích tabulek

9.3.1.1 ASCII

Standard ASCII (American Standard Code for Information Interchange) předepisuje kódovací tabulku, která vznikla v 60. letech 20. stol. a definuje číselnou reprezentaci nejčastěji používaných symbolů.

Tabulka obsahuje celkem 128 znaků (písmena, číslice, závorky, matematické znaky, interpunkční znaménka, speciální znaky, řídicí znaky, ...)

- 95 znaků viditelných (tzv. *printable*)
- 33 znaků neviditelných (tzv. *non-printable*)

ASCII Code Chart																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

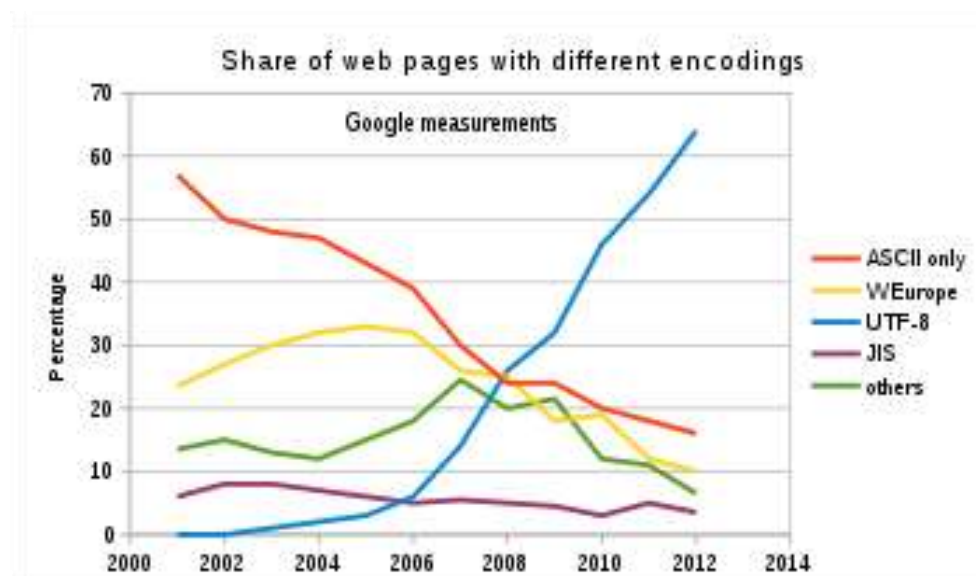
Na webu <http://www.ascii-table.com/> je k dispozici online ASCII tabulka včetně převodů z hexadecimálního zápisu do dekadické, binární i oktalové¹⁴.

Tato tabulka tvoří základ pro většinu moderních znakových sad.

¹⁴ *Zajímavost:* Osmičková (oktalová) soustava se používá např. při nastavování přístupových práv v OS unixového typu.

9.3.1.2 UTF-8

UTF-8 (angl. *Unicode Transformation Format*) je způsob kódování řetězců znaků (např. textu) do sekvencí bajtů. UTF-8 používá proměnnou délku znaku, od 1–4 (6) bajtů (8 – 16 (64) bitů), zatímco varianty UTF-16 a UTF-32 používá právě 16bitová (2 bajty), resp. 32bitová slova.



OBRÁZEK 9-9: WEBOVÉ STRÁNKY S ROZDÍLNÝM KÓDOVÁNÍM

Byl navržen pro zpětnou kompatibilitu s ASCII, se kterým tak má totožný způsob kódování 1bajtových znaků. UTF-8 je definováno v ISO 10646-1:2000 Annex D, v RFC 3629 a v Unicode 4.0.

<https://cs.wikipedia.org/wiki/UTF-8>

<https://www.dasm.cz/clanek/prevod-ceskeho-textu-z-utf-8-do-ascii>

9.3.2 Pěstování stromů (Huffmanovo kódování)

Zopakuj si: graf, hrana, vrchol, strom, kořen, binární strom, podstrom, souvislý graf, váha

Otázka efektivity kódování je úzce spjata s úlohou komprese a často se s ní překrývá. Úkolem komprese je zpracování dat s cílem zmenšit jejich objem při současném zachování informací v datech obsažených, snížit datový tok při jejich přenosu. Opakem komprese je dekomprese, která data zrekonstruuje zpět do původní podoby.

Dalším důležitým prvkem kódování může být odolnost vůči chybám při přenosu. Tzv. samoopravné kódy jsou navrženy tak, že pokud je počet chyb při přenosu menší než daná *mez*, tak jsme stále schopni ze zakódovaného textu i s chybami způsobenými přenosem zrekonstruovat původní text.

Huffmanovo kódování

= kód, který převádí jednotlivé symboly původního textu a dělá to co nejúspornějším způsobem – výsledná délka zakódovaného textu je v nějakém smyslu nejmenší možná.

Huffmanův kód získáme pomocí Huffmanova algoritmu, který David Huffman navrhl v 50. letech 20. století na univerzitě MIT.

Huffmanův algoritmus je jednoduchý grafový algoritmus, který se využívá zejména pro komprimaci dat.

- Vstup znaků = množina znaků spolu + četnost jejich výskytu ve zprávě
- Výstup znaků = Huffmanův strom, pomocí kterého lze jednotlivé znaky zakódovat do binárního kódu i dekodovat zpět.

Během přenosu zprávy se však musí kromě zakódovaného binárního řetězce přenést i použitý Huffmanův strom (za *předpokladu* není-li předem dohodnutý).

Princip komprese = častěji používané znaky zakódují menším počtem bitů než znaky méně časté.

Výsledný kód se nazývá prefixový, protože žádné kódové slovo není prefixem nějakého jiného. To zaručuje jednoznačnost kódování i dekódování.

Jinými slovy:

Princip metody spočívá ve vytvoření binárního stromu, jehož

- koncové uzly odpovídají symbolům původní abecedy,
- hrany jsou ohodnoceny symboly 0 a 1 a
- uzly jsou ohodnoceny pravděpodobností výskytu.

Pravděpodobnost vnitřního uzlu je přitom rovna součtu pravděpodobností jeho následníků. Uzly řadíme do posloupnosti podle rostoucí pravděpodobnosti, v každém kroku z ní odstraníme dva uzly s nejnižší prioritou, vytvoříme z nich následníky nového uzlu a ten opět zařadíme do seznamu.

Huffmanův kód má dvě důležité vlastnosti:

- Jedná se o kód s minimální délkou,
- Jedná se o tzv. prefixový kód a je tedy jednoznačně dekódovatelný.

Jeho problémem je to, že musíme znát rozdělení pravděpodobnosti výskytu jednotlivých symbolů. To lze nahradit odhadem, případně je možné tento odhad v průběhu komprese upřesňovat.

Použití Huffmanova kódu je časté v kombinaci s jinými kompresními algoritmy, například při kompresi obrazu a videa ve standardech JPEG a MPEG.

Průběh algoritmu

1. Vytvoř graf, který se skládá z n kořenových stromů. Každý podstrom T_i se skládá z jednoho kořenového uzlu reprezentujícího jeden symbol S_i . Váha stromu $w(T_i)$ je rovna četnosti použití daného symbolu S_i v dané zprávě
2. Dokud není graf souvislý, opakuj:
3. Najdi dva různé stromy T_1 a T_2 s minimální váhou.

4. Vytvoř nový kořenový strom T_3 s novým kořenem, jehož levý potomek je T_1 a pravý potomek T_2 . Váha tohoto nového stromu je rovna součtu vah stromů T_1 a T_2 , tedy $w(T_3) = w(T_1) + w(T_2)$.
5. Odeber oba dva stromy T_1, T_2 z grafu.
6. Pokud graf není souvislý, vrať se k bodu (2).

Výsledný binární kořenový strom se nazývá Huffmanův strom, kde každá

- levá hrana je ohodnocena 0 a
- pravá hrana je ohodnocena 1.

Kód pro každý znak je dán cestou z kořene do odpovídajícího listu tak, že se postupně zapisují binární čísla u navštívených hran.

Stejně znaky se stejnými četnostmi mohou díky rozdílné implementaci vytvořit i několik rozdílných Huffmanových stromů - délka kódových slov pro stejné symboly je však shodná.

• *Příklad (níže):*

Chceme zakódovat a zkomprimovat zprávu `AHOJ, JAK SE MAS, KAMARADE?`. Tato zpráva je dlouhá 27 znaků a obsahuje 13 různých symbolů.

Triviální kód

Následující tabulka udává jednotlivé znaky, jejich četnost a triviální kódování:

Znak	Četnost	Triviální kód
mezera	4	0000
,	2	0001
?	1	0010
A	6	0011
D	1	0100
E	2	0101
H	1	0110

J	2	0111
K	2	1000
M	2	1001
O	1	1010
R	1	1011
S	2	1100

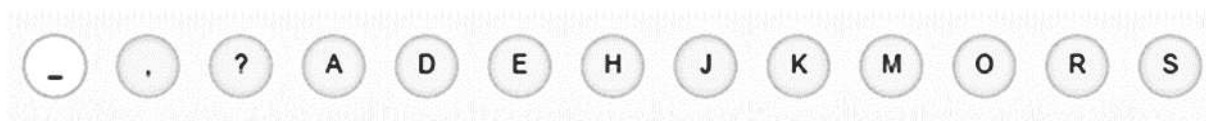
Zpráva by se pomocí triviálního kódování zakódovala do následujícího řetězce:

```
0011, 0110, 1010, 0111, 0001, 0000, 0111, 0011, 1000, 0000, 1100, 0101, 0000,
1001, 0011, 1100, 0001, 0000, 1000, 0011, 1001, 0011, 1011, 0011, 0100, 0101,
0010
```

Výpočtem zjistíme, že výsledná zpráva má velikost 108 bitů.

Huffmannův kód:

Prvním krokem kódování je sestavení Huffmanova stromu. V prvním části máme stromy představující symboly a váhy jsou rovny jejich četnosti.



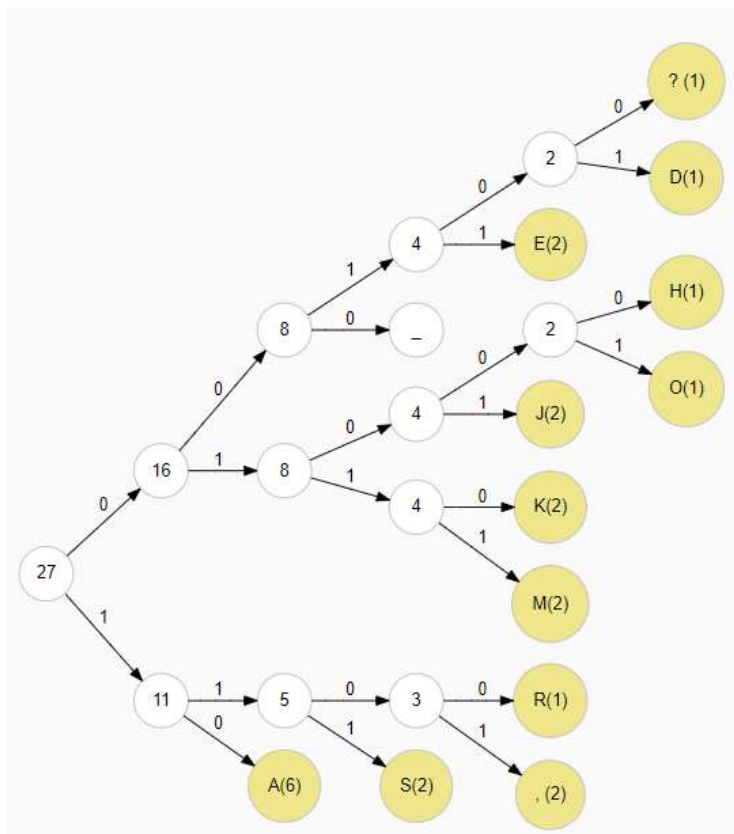
OBRÁZEK 9-10: LISTY HUFFMANOVA STROMU (HORDÉJČUK, 2019)

Následující kroky mohou vypadat například takto:

- $?(1) + D(1)$ s výslednou vahou 2
- $H(1) + O(1)$ s výslednou vahou 2
- $R(1) + , (2)$ s výslednou vahou 3
- $?D(2) + E(2)$ s výslednou vahou 4
- $HO(2) + J(2)$ s výslednou vahou 4
- $K(2) + M(2)$ s výslednou vahou 4
- $R, (3) + S(2)$ s výslednou vahou 5
- $(4) + ?DE(4)$ s výslednou vahou 8
- $HOJ(4) + KM(4)$ s výslednou vahou 8
- $A(6) + R,S(5)$ s výslednou vahou 11
- $?DE(8) + HOJKM(8)$ s výslednou vahou 16

- `?DEHOJKM(16) + AR, S(11)`...s výslednou vahou 27
- strom je dokončen

Výsledný Huffmanův strom



OBRÁZEK 9-11: VÝSLEDNÝ HUFFMANŮV STROM (HORDĚJČUK, 2019)

Výsledná kódovací tabulka:

Znak	Četnost	Triviální kód (pro srovnání)	Huffmanův kód
mezera	4	0000	000
,	2	0001	1100
?	1	0010	00100
A	6	0011	10
D	1	0100	00101
E	2	0101	0011
H	1	0110	01010
J	2	0111	0100

K	2	1000	0110
M	2	1001	0111
O	1	1010	01011
R	1	1011	1101
S	2	1100	111

Pomocí Huffmanova kódu se zpráva zakóduje takto:

```
10, 01010, 01011, 0100, 000, 1100, 0100, 10, 0110, 1100, 111, 0011, 1100,
0111, 10, 111, 000, 1100, 0110, 10, 0111, 10, 1101, 10, 00101, 0011, 00100
```

Výpočtem zjistíme, že výsledná zpráva má velikost 96 bitů. Spolu se zprávou je však nutné přenést i odpovídající Huffmanův strom (není-li dohodnutý předem) a tak se výhoda Huffmanova kódování projeví až u zpráv určité délky.

9.4 Kontrolní součty přenosu kódovaných zpráv

Informace

- Zpráva, kterou jsme schopni vnímat, rozlišit a porozumět
- Význam, který přisuzujeme datům (vztah mezi symboly a okolním světem)
- Informaci tvoří kódovaná data (data lze přijímat a vysílat)
- Význam připraný datům, přená a včasná data

Data

- Reprezentace informace formálním způsobem, tak aby bylo možné je přenášet a zpracovávat stroji
- Dělení:
 - Textová
 - Binární

Zpráva

- Jednorázová informace, kterou člověk / skupina lidí předává jiným lidem
- Složena z abecedních a číselných znaků a dílčími prvky
- Nejmenší jednotka zprávy = znak

- Množina všech znaků = abeceda
- Zpráva má svoji abecedu a gramatiku

Přenos informace

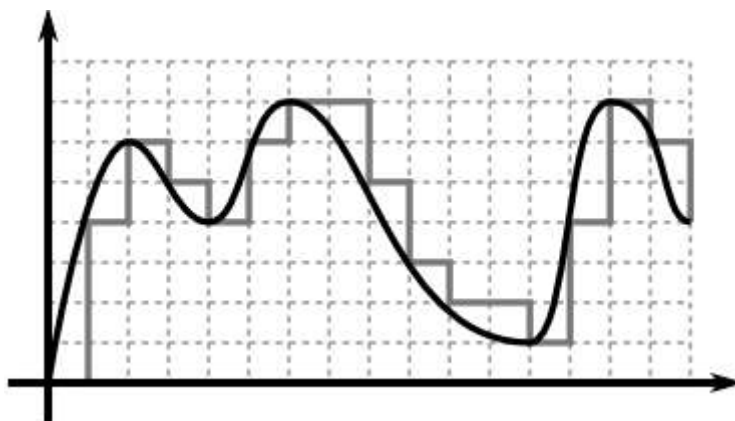
- Informace se přenáší pomocí nosičů (informace je v nosiči zakódována)
- Nosič obsahuje informace v podobě slov, písmen, obrázků (= data)

Šíření např.:

- mechanicky, vlnění vzduchu
 - zvuky / řeč
- opticky světelnými paprsky
- změnami elektrických veličin
 - telefon, počítač
- chemicky koncentrací určitých molekul
 - čich, chuť

Obecně se informace přenáší pomocí fyzikálních a chemických signálů, které můžeme rozdělit do dvou skupin:

- analogové signály = mění svou hodnotu spojitě s časem
- diskrétní signál = nespojitý, skokové změny hodnot



OBRÁZEK 9-12: ANALOGICKÝ A DISKRÉTNÍ SIGNÁL

Kontrolní součet

= informace, která se přenáší spolu s vlastní (předávanou) informací

- slouží k ověření, zda vlastní informace byla přenesena správně
 - resp. slouží k detekci chyb v přenosu informace
- Jedná se o výsledek předem určené operace, provedené s vlastní informací
- Příjemce má možnost si spočítat vlastní kontrolní součet – pokud je výsledek odlišný, znamená to, že přenos byl chybný.

9.4.1 Metody kontrolních součtů

Používají se různé metody s různou efektivitou pro různá použití:

9.4.1.1 Zaslání úplné kopie celé informace

- Nejtriviálnější metoda
- Porovnávají se samotné informace, pokud se liší – kopie se vzájemně liší
- Silně neefektivní – vysoká míra redundance¹⁵

9.4.1.2 Parita (paritní bit)

- Parita (mat.) = sudé / liché číslo
- Redundantní (nadbytečný) bit, přidaný k datovému slovu
- Obsahuje informaci o počtu jedničkových bitů ve *slově*
- Detekuje lichý počet chybných bitů
- Speciální případ 1bitového CRC (polynom $x + 1$)

Výpočet

Hodnotu paritního bitu lze jednoduše vypočítat jako XOR mezi všemi datovými bity slova.

¹⁵ Redundance = informační nebo funkční nadbytek, například větší množství informace, prvků nebo zařízení, než je nezbytné.)

7bitová data	1 byte s paritním bitem	
	sudá parita	lichá parita
0000000	00000000	10000000
1010001	11010001	01010001
1101001	01101001	11101001
1111111	11111111	01111111

9.4.1.3 Modulo

Operace modulo se hojně využívá v programování a návrhu algoritmů, např. při testu sudosti čísla nebo výpočtu dne v týdnu.

Často se též využívá při generování kontrolních součtů, které bývají součástí komunikačních protokolů.

$$(a \bmod m) = a - \left\lfloor \frac{a}{m} \right\rfloor m$$

kde $\left\lfloor \frac{a}{m} \right\rfloor$ je dolní celá část podílu $\frac{a}{m}$, tedy největší celé číslo, které je menší nebo rovno číslu $\frac{a}{m}$.

$$\frac{a}{m} \in \mathbb{R}, n \in \mathbb{N}: \left\lfloor \frac{a}{m} \right\rfloor = n \Leftrightarrow n \leq \frac{a}{m} < n + 1$$

Neboli:

$$(70 \bmod 8) = 6 \Leftrightarrow \frac{70}{8} = 8 + 6$$

$$70 - \left\lfloor \frac{70}{8} \right\rfloor \cdot 8 = 70 - 8 \cdot 8 = 70 - 64 = 6$$

9.4.1.4 Hammingův kód (x)

Richard W. Hamming (1915 – † 1998) matematik.*

Významný pro matematickou informatiku a telekomunikace.

Dílo: Hammingův kód, Hammingova vzdálenost, Hammingovo okno.

Lineární kód pro detekci až dvou chybných bitů nebo opravu jednoho chybného bitu.

Své využití našel v oblasti telekomunikací.

Základem je Hammingův kód (7, 4), ale lze jej zobecnit i na jiné počty datových a paritních bitů.

Binární kód je Hammingův, právě tehdy když má kontrolní matici, jejíž sloupce jsou všechna nenulová slova dané délky $n - k = r$ a žádné z nich se neopakuje

rozpracováno

9.4.1.5 Cyklický redundantní součet (CRC)

- Hashovací funkce (používaná pro detekci chyb během přenosu a ukládání dat)
- Jednoduchý způsob + dobré matematické vlastnosti = rozšířený a oblíbený způsob realizace kontrolního součtu
- CRC bývá odesílán společně s přenášenými informacemi; po přenosu, resp. při přijetí je CRC znovu nezávisle spočítán a porovnán s původním
 - Odlišný → při přenosu došlo k chybě
 - Shodný → přenos proběhl v pořádku
- Vhodný pro zjišťování chyby v důsledku selhání techniky
- Nevhodný pro odhalení záměrné změny dat – zde jsou doporučeny jiné speciální hashovací funkce
- V určitých případech je možné chybu pomocí CRC opravit

Princip fungování CRC

Vektor = posloupnost bitů (posloupnost jedniček a nul)

Je dán vektor `10100001011111110001`. Tento vektor budeme přenášet.



Příjemce *však* přijal tento vektor: `10100000010111110001`

Došlo k částečnému poškození dat. Valná většina bitů zůstala nepoškozena.

Abychom mohli říci, jaké bity se při přenosu změnili, zavedeme si pojem chybový vektor, ten bude mít hodnotu:

- 1 na místě kde došlo ke změně původní hodnoty a
- 0 tehdy, zůstal-li bit nezměněn.

V našem případě vypadá chybový vektor následovně: `00000001001000000000`

Charakter chybových vektorů je pro různá média různý. Budeme-li přenášet data pomocí diskety a dané médium (*disketa*) se poškrábe, je zřejmé, že takový škrábanec na disketě zničí data v určité souvislé oblasti v závislosti na tloušťce rýhu. Čili výsledný chybový vektor bude mít s největší pravděpodobností jedničky kumulované velmi blízko u sebe.

Při návrhu algoritmu CRC pro konkrétní médium si určitě prvně uděláte představu o tom, jak bude nejčastěji vypadat chybový vektor v případě chyby. CRC je kontrolní součet, který, jak za okamžik uvidíme, lze jednoduchou modifikací do značné míry přizpůsobit různým typům chybových vektorů.

Hlavní operací, kterou budete při počítání CRC potřebovat, je exkluzivní součet (bitová nonekvivalence) neboli XOR:

a	b	a XOR b
0	1	1
0	0	0
1	1	0
1	0	1

XOR vrací 1 právě tehdy, pokud se hodnoty liší, jinak 0.

```
c = a xor b
```

`a` lze chápat jako vektor před přenosem a `b` jako vektor po přenosu. `c` pak definuje chybový vektor.

Exkluzivní součet je komutativní a asociativní.

```
a xor b = b xor a
(a xor b) xor c = a xor (b xor c)
```

Další zajímavé vlastnosti:

```
a xor b xor b = a
a xor b xor a = b
a xor 0 = a
a xor a = 0
```

Příklad:

```
A =      10011101
B =      00100010
A xor B = -----
          10111111
```

Výstupem funkce, počínající CRC je číslo Toto číslo je obecně n -bitové, nejčastěji však pro $n = 2^m$, kde $m, n \in \mathbb{N}$, tedy CRC 16bit, CRC-32bit, příp. CRC-64bit nebo CRC-128bit. Čím je CRC „více-bitové“, tím je zřejmě nižší pravděpodobnost, že bude mít špatně přenesená informace stejné CRC jako zpráva původní, *tedy* že se chyba nepozná.

Ukázka výpočtu CRC-4bitového:

Jediný parametr, který můžeme ovlivňovat je tzv. generační polynom (GP). GP je 4bitové číslo (pozn.: pro CRC-32-bit by byl GP 32bitový).

```
■ GP = 1001
```

Data, pro která budeme CRC počítat:

```
■ 101101110
```

V první řadě k datům přidáme na konec tolik nul, kolik je délka GP, resp. kolika bitové CRC počítáme. V našem případě budeme přidávat 4 nuly:

```
■ 1011011100000
```

Dále vezmeme GP a různě jej rotujeme vlevo a posíláme operaci XOR do doby, než budou původní data vynulována:

```
■
  1011011100000  0010011100000  0000001100000
  1001           1001           1001
xor -----
  0010011100000  0000001100000  0000000101000

      0000000101000
          1001
xor -----
      0000000001100
          ^^^^
```

Po této operaci si všimněte, že přidané nuly se zaměnily na 1100, což je žádoucí výsledek, tedy CRC-4bit

Nová zpráva, kterou odesíláme bude vypadat následovně:

```
■ 1011011101100
```

Příjemce informace má pak dvě možnosti:

- První – zřejmá. Shodným způsobem vypočítají CRC a následně porovnají, zda se CRC shodují (funkce XOR)
- Druhý způsob – Výpočet CRC pro celou došlou posloupnost znovu – měla by vyjít 0 (*viz ukázka*):

```

1011011101100
1001
  1001
    1001
      1001
xor -----
0000000000000

```

Důkaz proč tomu tak je nechám na zvědavém čtenáři. Úvahově směřujte ke skládání XOR funkcí. Dále je dobré si uvědomit komutativnímu bitové nonekvivalence, tedy že nezáleží na pořadí, v jakém budeme GP xorovat.

Obecně nám ale nic nebrání to udělat třeba takto:

```

1011011101100
1001
  1001
    1001
      1001
xor -----
0000000000000

```

Efektivita (resp. šikovnost) je však diskutabilní.

Ukázka detekce chybného přenosu

Předpokládejme, že data přišla porušená s chybovým vektorem 0000000100000.

Přijatá zpráva tedy bude vypadat následovně:

```

1011011101100
0000000100000
xor -----
1011011001100

```

Pozn.: Příjemce samozřejmě neví, že je zpráva porušená, natož aby znal chybový vektor. Vektor zde uvádím z didakticko-edukativních účelů.

Při kontrolním xorování tedy nevyjdou samé nuly, ale chybový vektor

```

10110111011000000
1001
  1001
    1001
      1001
xor -----
00000001000000000

```

Pokud chybový vektor budeme xorovat s přijatou (leč chybně přenesenou) informací, můžeme ji do jisté míry opravit.

```
10110111011000000 //chybně přenesená informace
00000001000000000 //chybový vektor
xor -----
10110110011000000 //správná informace 😊
10110111011000000
xor -----
00000000000000000
```

Jak volit GP?

Volba GP není úplně triviální a v pozadí hraje vysokou roli jistá dávka matematiky. Obvykle si ale GP nevymýšlíme, ale používáme GP, které již někdo vymyslel za nás.

Jiná možnost je postup, pomocí přepisu vektoru na tvar polynomu

Interpretujme vektor 100101 jako polynom:

Mocnina polynomu (stupeň polynomu)	5	4	3	2	1	0
Odpovídající bit	1	0	0	1	0	1

Přepis do tvaru polynomu:

$$1x^5 + 0x^4 + 0x^3 + 1x^2 + 0x + 1 = x^5 + x^2 + 1$$

· Mějme posloupnost bitů 100101 a 110011. Převeď tyto posloupnosti na polynomičtý tvar. Převeďte nad bity těchto dvou posloupností operace XOR a AND a výsledky porovnejte

9.4.1.6 Error Checking and Correcting (ECC)

česky též *kontrola a oprava chyb*

- technologie umožňující zjistit a opravit chybu
- využití se u serverů, příp. dražších počítačů
- čím vyšší kapacita operační paměti, tím vyšší riziko takových chyb, kdy dojde náhodně ke změně bitu z 1 na 0 nebo naopak (může způsobit značné následky)
- ECC snižuje riziko těchto chyb na minimum → paměť typu ECC (dokáže chyby zjistit a opravit)
- možné díky využití několika bitů v každém slově navíc

9.4.1.7 Message-Digest algorithm (MD5)

- rodina hašovacích funkcí, která z libovolného vstupu dat vytváří výstup pevné délky (= hash)
- malá změna na vstupu vede k velké změně na výstupu, tj. k vytvoření diametrálně odlišného otisku

Podrobněji se tímto tématem zabývá kapitola xy.

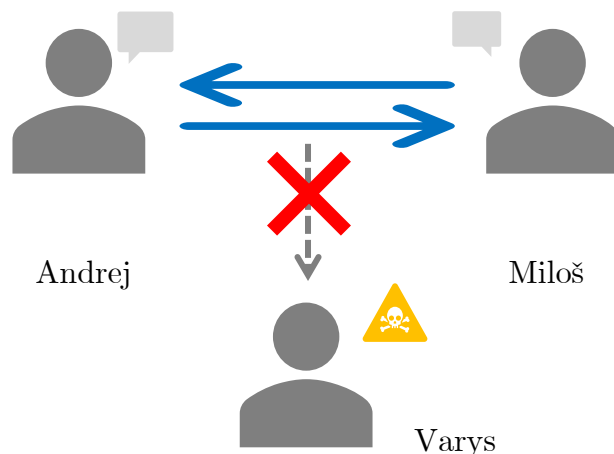
9.4.1.8 Secure Hash Algorithm (SHA)

- hashovací funkce (z libovolného vstupu vytváří hash)
- ze znalosti otisku je nemožné zrekonstruovat původní – vstupní – data
- malá změna na vstupu vede k velké změně na výstupu, tj. k vytvoření diametrálně odlišného otisku.

Podrobněji se tímto tématem zabývá kapitola xy.

9.5 Úvod do kryptografie

Kryptografie = Utajování smyslu zpráv převodem do podoby, která je čitelná jen se speciální znalostí



- Slovo kryptografie má původ v řečtině – *kryptós* = skrytý; *gráphein* = psát
 - dosl. skryté psaní, v důsledku - skryté předávání informací
- Pojem *kryptografie* se též obecně používá pro vědu spojenou se šiframi, alternativně k pojmu kryptologie
- Kryptologie, coby věda, zahrnuje
 - kryptografii (ve výše popsaném smyslu) a
 - kryptoanalýzu (= luštění zašifrovaných zpráv)

9.5.1 Historie kryptografie

Kryptografie se po staletí rozvíjela k větší složitosti zároveň s lidskou civilizací a mnohokrát ovlivnila běh dějin. Zejména utajení či vyzrazení strategických vojenských informací může mít zásadní vliv. Ale také prozrazení politických intrik, přípravy atentátů nebo i jen prozrazení milenců a podobně, to vše může úzce podléhat bezpečnému přenosu informací, a také na schopnostech šifru rozbít. První zmínka o zašifrování informace pochází z roku 480 př. Kr. z období Řecko-Perských válek v bitvě u Salamíny.

Do historie kryptografie se zapsal i významný římský vojevůdce a politik Julius Caesar, a to vynalezením šifry, která byla pojmenována jako Caesarova šifra. Ta byla ovšem dosti primitivní. Obor šifrování pak zaznamenal značný rozvoj a už na

přelomu středověku a novověku dosáhl pozoruhodných pokroků a šifry nečekaného stupně odolnosti. (např. expertům španělské tajné služby trvalo v roce 2017 i s veškerým moderním vybavením celého půl roku, než prolomili šifrování korespondence mezi španělským králem Ferdinandem Aragonským a jeho generálem Gonzalem Fernándeze de Córdoba, který vedl tažení španělských vojsk na Apeninském poloostrově na přelomu 15. a 16. století.)

Celé období kryptografie pak můžeme rozdělit do dvou částí:

- klasická kryptografie, která přibližně trvala do poloviny 20. stol.
- moderní kryptografie

Klasická kryptografie se vyznačovala tím, že k šifrování stačila pouze tužka a papír, případně jiné jednoduché pomůcky. Během 1. poloviny 20. stol. se však začaly formovat sofistikovanější přístupy k šifrování, a to pomocí moderní technologie. Tato technologie odstartovala novou éru v šifrování, kterou nazýváme moderní kryptografie. V dnešní době se k šifrování zpravidla nepoužívají žádné zvlášť vytvářené přístroje, ale klasické počítače.

9.5.2 Základní pojmy

- Šifrování = kryptografický algoritmus, který převádí klasický – čitelný text na jeho nečitelnou podobu – šifru, šifrový text
- Dešifrování = opačný postup šifrování
- Klíč = parametr šifrování, množina možných klíčů musí být tak velká, aby nebylo možné zprávu rozluštit postupným zkoušením všech klíčů
- Symetrická šifra = používá k šifrování a dešifrování stejný klíč
- Asymetrická šifra = používá k šifrování dvojici klíčů, z nichž jeden slouží pro šifrování a druhý pro dešifrování
 - Pokud z šifrovacího klíče nelze určit dešifrovací, pak může majitel dvojice klíčů svůj šifrovací klíč zveřejnit – tzv. *veřejný klíč*. Druhý klíč z dvojice sloužící pro dešifrování je *soukromý klíč*. Ten nikdy nezveřejňujeme!

- Vícenásobné šifrování = Danou šifru / skupinu šifer použijeme na zprávu několikrát po sobě
- Hashovací funkce = způsob, jak z libovolně dlouhého textu vytvořit jedinečný krátký řetězec, který s velmi velkou pravděpodobností text identifikuje;
 - pro kryptografii musí být hashovací funkce speciálně navržené, aby nebylo možné vytvořit jiný text, který bude mít stejnou hodnotu hashovací funkce
- Elektronický podpis = slouží pro ověření autenticity (podpisu) odesílatele, zpráva nebyla po podpisu změněna.
 - Při využití *asymetrické kryptografie* je pak zpráva i *nepopíratelná*. Autor nemůže popřít, že zprávu podepsal (jeho soukromý klíč nikdo jiný nezná a nelze tedy zprávu podvrhnout)
- Digitální certifikát = prostředek, kterým lze prokázat totožnost majitele díky tomu, že je certifikát elektronicky podepsán důvěryhodnou autoritou.

9.5.3 Metody klasické kryptografie

9.5.3.1 Steganografie

- „předchůdce“ kryptografie
- Ukryvání zprávy (neviditelné inkousty, vyrývání zprávy do dřeva, zalévání voskem, ...)
- V současnosti lze tajné texty ukrývat do souborů s hudbou či obrázky (soubor.txt pojmenuju jako soubor.jpg)
- Johannes Trithemius (* 1462 – † 1516)

9.5.3.2 Substituční šifry

- Substituce = záměna
- Nahrazení každého znaku zprávy jiným znakem podle daného pravidla

Caesarova šifra (Posun písmen)

- Symetrická šifra
- Pojmenována po Juliu Caesarovi (používal šifru jako první)
- Každé písmeno šifrované zprávy je posunuto v abecedě o pevný počet pozic
- V současnosti je silně triviální (existuje málo možných klíčů)
- Ve své době však svou funkci plnila

GAIUS JULIUS CAESAR = JDLXV MXOLXV FDHVDV

Tabulka záměn

- Založeno na záměně znaku za jiný bez jakékoliv vnitřní souvislosti či na základě znalosti hesla (klíče)

Příklad tabulky s heslem *Mrkev*

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
M	R	K	E	V	W	X	Y	Z	A	B	C	D	F	G	H	I	J	L	N	O	P	Q	S	T	U

BRAMBORY = RJMDRGJT

9.5.3.3 Aditivní šifry

Vigenèrova šifra

- Základem Vigenérovky šifry je heslo, jehož znaky určují posunutí otevřeného textu \Rightarrow otevřený text se rozdělí na bloky znaků dlouhé stejně jako heslo a každý znak se sečte s odpovídajícím znakem hesla.
- Caesarova šifra je tedy speciální případ Vigenèrovky šifry s heslem o délce jeden znak.
- Vigenèrova šifra způsobuje změny pravděpodobnosti rozložení znaků a tím výrazně znemožňuje kryptoanalýzu na základě analýzy četnosti znaků v textu.
- *Luštění založeno na vyhledávání vzdálenosti dvojic v zašifrovaném textu a určováním jejich společného dělitele vedoucího k zjištění délky hesla

Příklad s použitím slova HESLO jako klíče k zašifrování:

otevřený text	S	T	A	S	T	N	E	A	V	E	S	E	L	E
klíč	H	E	S	L	O	H	E	S	L	O	H	E	S	L
šifrový text	A	Y	T	E	I	V	J	T	H	T	A	J	E	Q

Vernamova šifra (one-time pad)

- Česky též jednorázová tabulková šifra
- Jediná známá šifra, u které byla exaktně dokázána nerozluštitelnost
- Absolutně bezpečná (délka klíče však běžné používání znemožňuje)
- Využití
 - spojení Moskvy s Washingtonem za dob studené války
 - dnes v kombinaci s kvantovou kryptografií

9.5.3.4 Transpoziční šifry

Transpozice / přesmyčka = změna pořadí znaků dle smluveného pravidla.

Skytalé*

- šifrování, které používali staří Řekové (Spartané) během válek
- První zmínka: řecký básník Archilochos z Paru (7. stol. př. Kr.) a Apollónios z Rhodu (3. stol. př. Kr.)
- Princip fungování odhalil Plútarchos (přelom 1. a 2. stol. po Kr.)

Princip: na dřevěný válec předem dohodnutého průměru se namotal pásek pergamenu a ve směru hlavní osy válce se na něj napsal text od jednoho konce k druhému. Po jeho odmotání na pásku zůstala písmena, která nedávala spolu smysl. Jediným způsobem, jak dešifrovat text, bylo namotání pásky na válec o stejné velikosti průměru.

9.5.4 Historie moderní kryptografie

Použití mechanických a elektronických strojů přineslo do šifrování nové možnosti (stroje je schopny velkého počtu operací a cyklů během krátkého času).

Enigma

- Šifrovací stroj (šifrování a dešifrování informací), 20. léta 20. století
- Civilní zprávy, armáda a vláda, např. Německo, II. světová válka
- Prolomen 1932 Marianem Rejewskim (polský kryptoanalytik) za pomoci informací z Francie, která je získala od německého špiona Hanse Schmidta.
- Po operaci Fall Weiß (Invaze do Polska, IX/X 1939) navázali na jejich práci britští kryptoanalytici pod vedením Alana Turinga^{16 17}
- Němci používali tento stroj ještě v 50. letech minulého století (domnívali se, že je stroj nerozluštěný)

¹⁶ prof. Alan Turing (* 1912 – † 1954), britský matematik, logik, kryptoanalytik a zakladatel moderní informatiky

¹⁷ Film Kód Enigmy pojednává o Alanu Turingovi, a jeho práci s dešifrováním Enigmy

Thriller, Drama, Životopisný; USA, 2014

Hrají: B. Cumberbatch, K. Knightley, M. Strong

Režie: M. Tyldum

9.5.5 Moderní šifry

Síla moderního šifrování je přímo závislá na generátoru pseudonáhodných čísel.

9.5.5.1 Symetrické šifry

- používá k šifrování a dešifrování stejný klíč
- První standardizovaná šifra DES, v současnosti AES a Rijndael

9.5.5.2 Asymetrické

- používá k šifrování dvojici klíčů, z nichž jeden slouží pro šifrování a druhý pro dešifrování (soukromý a veřejný klíč)
- RSA – nejpoužívanější asymetrická šifra (využití např.: digitální podpis)
- Platí, že informace, která je zašifrována veřejným klíčem, lze dešifrovat pouze odpovídajícím klíčem soukromým. Z opačné strany lze zprávy zašifrované soukromým klíčem rozšifrovat pouze příslušným veřejným klíčem.
- *Touto problematikou se blíže zabývám v kapitole 9.6 RSA, str. 245*

9.5.5.3 Hashovací funkce

- Jednosměrná matematická funkce pro převod vstupních dat do malého čísla (hashe)
- *Touto problematikou se blíže zabývám v kapitole 9.7 Hashovací funkce, str. 249*

9.6 RSA

- RSA = iniciály autorů Rivest, Shamir, Adleman
- šifra s veřejným klíčem
- první algoritmus – vhodný jak pro podepisování, tak pro šifrování
- dostatečné délce klíče je považován za bezpečný – používá se i dnes

9.6.1 Princip a popis algoritmu

Mějme dva uživatele A a B . Naznačme, jakým způsobem musí postupovat, chtějí-li spolu komunikovat pomocí RSA.

Eulerova věta: $a^{\varphi(m)} = 1$ v \mathbb{Z}_m , kde $\varphi(m)$ je Eulerova funkce a $\text{NSD}(a, m) = 1$.

Eulerova funkce

Eulerova funkce $\varphi(n)$, kde $n \in \mathbb{N}_2$ ($\mathbb{N}_2 = \mathbb{N} \setminus \{1\}$) je definovaná jako počet kladných celých čísel, která jsou nižší než n a jsou s n nesoudělná, resp.

$$\text{NSD}(a_1, a_2, a_3 \dots) = 1.$$

Platí, že $\varphi(1) = 1$.

Multiplikativní inverze

Multiplikativní inverzí čísla $x \in \mathbb{Z}_p$, kde p je prvočíslo rozumíme takové číslo x^{-1} , pro které platí: $x \cdot x^{-1} \equiv 1$. Existují dva různé postupy pro zjištění hodnoty:

- (a) Hádání, resp. zkoušení ostatních čísel (neefektivní, silně obtížné pro velká p)
- (b) Rozšířený Euklidův algoritmus

Výpočet klíčů

Uživatel A si zvolí dvě náhodná různá prvočísla p_a a q_a . A jejich součin označí jako $n_a = p_a \cdot q_a$. Následně si zvolí číslo e_a takové, že je nesoudělné s $\varphi(n_a) = \varphi(p_a \cdot q_a) = (p_a - 1)(q_a - 1)$, kde $\varphi(n_a)$ je hodnota Eulerovy funkce pro n ($e_a < \varphi(n_a)$).

Nakonec vypočítá číslo d_a , které je multiplikativní inverzí čísla e_a .

Dvojice (n_a, e_a) tvoří veřejný klíč.

Dvojice (n_a, d_a) tvoří soukromý klíč.

Uživatel B postupuje analogicky a také si vytvoří veřejný a soukromý klíč.

Šifrování a dešifrování

Pokud nyní bude chtít uživatel A poslat uživateli B zprávu – přirozené číslo z , tak jej zašifruje pomocí veřejného klíče B_{pk} jako $x = z^{e_b} \cdot \text{mod}(n_b)$. Uživatel B přijatou zprávu dešifruje pomocí vzorce $z = x^{d_b} \cdot \text{mod}(n_b)$.

Příklad

Mějme dva uživatele: Monika a Karel

Monika volí dvě náhodná prvočísla, vypočte modulo (zbytek po dělení) a hodnotu Eulerovy funkce.

$$p_m = 17$$

$$q_m = 19$$

$$n_m = 17 \cdot 19 = 323$$

$$\varphi(323) = \varphi(17 \cdot 19) = (17 - 1)(19 - 1) = 288$$

Nyní si zvolí číslo $e_m = 37$ a pomocí rozšířeného Euklidova algoritmu vypočítá jeho multiplikativní inverzi e_m^{-1} v $\mathbb{Z}_{\varphi(n_m)}$.

$$d_m = 37^{-1} \text{mod}(288) = 109 \cdot \text{mod}(288)$$

Monika nyní zveřejní svůj veřejný klíč $(323; 37)$ a soukromý klíč $(323; 109)$ si ponechá.

Karel volí dvě náhodná prvočísla, vypočte modulo a hodnotu Eulerovy funkce

$$p_k = 7$$

$$q_k = 11$$

$$n_k = 7 \cdot 11 = 77$$

$$\varphi(77) = \varphi(7 \cdot 11) = (7 - 1)(11 - 1) = 60$$

Nyní si zvolí číslo $e_k = 23$ a pomocí rozšířeného Euklidova algoritmu vypočítá jeho multiplikativní inverzi e_k^{-1} v $\mathbb{Z}_{\varphi(n_k)}$.

$$d_k = 23^{-1} \bmod(60) = 47 \cdot \bmod(60).$$

Karel nyní zveřejní svůj veřejný klíč $(77; 23)$ a soukromý klíč $(77; 47)$ si ponechá.

Karel nyní chce Monice poslat zprávu 15 (pro jednoduchost).

Karel má přístup k veřejnému klíči Moniky a zašifruje zprávu jako:

$$x = 15^{37} \bmod(323) = 53$$

Zašifrovanou zprávu pošle Monice a ta je rozšifruje pomocí svého soukromého klíče jako:

$$z = 53^{109} \bmod(323) = 15$$

Útok RSA hrubou silou

- Nejjednodušší a nejméně efektivní útok
- Založen na faktorizaci prvočísel (obtížný matematický problém) – tedy zjištění soukromého klíče
- Faktorizace je možná pro dostatečně malá prvočísla \Rightarrow pro výpočet klíčů vždy volíme klíče délky 1024–2048 bitů, které se již nedají běžnými prostředky faktorizovat
 - *Je předpoklad, že kvantové počítače si s faktorizací prvočísel poradí.*

9.7 Hashovací funkce

- Algoritmus pro převod vstupních dat do tzv. hashe/haše (= číselná miniatura vstupu)
- Používá se
 - rychlejší prohledávání tabulky (porovnávání dat),
 - hledání položek v databázích a detekci duplicitních hodnot,
 - hledání malware antivirovým programem,
 - hledání podobných úseků DNA sekvencí v bioinformatice atd.
- Kryptografická hashovací funkce – např. pro vytváření a ověřování elektronického podpisu

Vlastnosti

- Pro libovolný vstup poskytuje stejně dlouhý hash
- Malá změna vstupu → velká změna výstupu (hash se od původního liší)

Vstup	Hash SHA-1
sírový	2090a57323a110f70c1e15e57dbaf190b4635354
sýrový	993183dca1300cd0b46431fe39887edc616da020

- Z hashe je nemožné (?) zrekonstruovat text původní zprávy
- V praxi nepravděpodobné, že dvěma různým zprávám bude odpovídat stejný hash (resp. pomocí hashe lze identifikovat právě jednu zprávu)
- Hashování dovoluje testovat shodu dat (rovnost), např. *ověřování hesel*
- Nezachovává podobnost dat ani uspořádání (viz výše)

Využití

- Kontrola integrity dat (shodnost velkých souborů)
- Autentizace, ukládání a kontrola přihlašovacích hesel
- Prokazování autorství
- Jednoznačná identifikace dat (digitální podpis)

Hashovací algoritmus je bezpečný, pokud je velmi obtížné (tj. se současnými prostředky prakticky nemožné):

- 1) najít zprávu, která odpovídá svému otisku
- 2) najít dvě rozdílné zprávy, které mají stejný otisk

9.7.1 MD5

MD (Message-Digest algorithm) = skupina hashovacích algoritmů (prof. Ronald L. Rivest, MIT, 1990)

- MD5 byl v r. 1991 navržen jako bezpečná náhrada za MD4, který byl označen za nedostatečně zabezpečený
- Využíván v softwaru, který rozpozná nedotčenost přenášených dat, příp. pro kontrolu integrity souborů u operačních systémů založené na Unixu
- 1996 – objeveny chyby, přestože nebyly podstatné, kryptologové začali doporučovat jiné algoritmy, např. SHA
- 2004 – nalezeny větší chyby, MD5 se zásadně nedoporučuje

MD5(„Každý nemůže být chytřej. Ty hloupí musej dělat výjimku, protože kdyby byl každý chytřej, tak by bylo na světě tolik rozumu, že by z toho byl každý druhý člověk úplně blbej.“)

= 62047e3f413ac220eff75f911fd66d84

Ukázka hashe MD5

9.7.1.1 Výpočet hashe

- Vstup = jakákoliv zpráva libovolné délky
- Výstup = 512 bitů velké bloky \Rightarrow 16 slov o velikost 32 bitů
 - 448 bitů = samotná zpráva
 - 64 bitů = délka původní zprávy
- Každý blok je počítán samostatně – pro výpočet jsou použity 4 proměnné (A, B, C, D), inicializační hodnoty jsou předem stanoveny (hexadecimální soustava)

- Každá z těchto funkcí má
 - na vstupu tři 32bitová slova a
 - na výstupu jedno 32bitové slovo

```

F(X, Y, Z) = (X AND Y) OR (NOT(X) AND Z)
G(X, Y, Z) = (X AND Z) OR (Y AND NOT(Z))
H(X, Y, Z) = X XOR Y XOR Z
I(X, Y, Z) = Y XOR (X OR NOT(Z))

```

Každý z těchto funkcí je postupně vykonávána 16krát ve 4 kolech. Výslednou hash tvoří proměnné A, B, C, D , ke kterým je přičtena jejich inicializační hodnota.

```

public class MD5 {
    public static String getMD5(String input) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] messageDigest = md.digest(input.getBytes());
            BigInteger number = new BigInteger(1, messageDigest);
            String hashtext = number.toString(16);
            while (hashtext.length() < 32) {
                hashtext = "0" + hashtext;
            }
            return hashtext;
        }
        catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws NoSuchAlgorithmException {
        System.out.println(getMD5("Javarmi.com"));
    }
}

```

9.7.1.2 Zvýšení bezpečnosti

Algoritmus MD5 se často používal pro ukládání hesel. Pro navýšení bezpečnosti se využívá tzv. kryptografická sůl (několik náhodných bitů, slouží pro doplnění vstupu při použití hashovací funkce). Přidáním soli k heslu se ztíží útoky na získání hesla a účinnost slovníkových stoků s využitím přepočítaných tabulek (rainbow table attack). Tento postup je do jisté míry univerzální – lze použít i na jiné hashovací funkce, které lze dopředu „*předpočítat*“, jejich výsledný hash uložit a pak rychleji hledat v databázi srovnáváním s hashem na který se útočí.

Příklad v Programovacím jazyce Java

Pro určité navýšení bezpečnosti lze též použít kombinaci hesla a uživatelského jména. V případě, kdy si dva uživatelé zvolí stejné heslo, jejich hash se bude zásadně lišit, protože jejich uživatelská jména budou odlišná.

Příklad v Programovacím jazyce Java

Další eventualitou je kombinace hashovacích funkcí, například použití MD5 a SHA současně. Tento postup zajistí chráněné informace i v případě, že bude na jedné funkci nalezena kolize.

Příklad v Programovacím jazyce Java

9.7.1.3 Použití v praxi

- MD5 v softwarovém světě ⇔ jistota, že přenášený soubor dorazí beze změny
- Unixové operační systémy obsahují aplikace pro výpočet *MD5 sumy* v jejich distribučních balíčcích, zatímco uživatelé Windows jsou nuceni použít aplikace třetích stran. 😊

–

9.7.2 SHA

SHA (secure hash algorithm) společný název pro skupiny rozšířených hašovacích algoritmů (*SHA-0*, *SHA-1*, *SHA-2* a *SHA-3*), které převádí vstupní data na otisk dané délky.

- Ze znalosti otisku je nemožné rekonstruovat vstupní data
- Malá změna na vstupu vede k velké změně na výstupu, tj. k vytvoření diametrálně odlišného otisku

Algoritmus (chcete-li hashovací funkci) SHA navrhla Národní bezpečnostní agentura USA (NSA) ve spolupráci s Národním institutem pro standardy v USA (NIST).

Jedná se o skupinu pěti algoritmů:

- SHA-1: Obraz dlouhý 160 bitů
 - SHA-224
 - SHA-256
 - SHA-384
 - SHA-512
- Tyto varianty se souhrnně uvádějí jako SHA-2.

Délka výstupního hashe ostatních algoritmů skupiny SHA-2 je určena číslem u konkrétního názvu, *např. SHA-224 má hash dlouhý 224 bitů.*

Algoritmy rodiny SHA se používají u několika různých protokolů a aplikací, včetně TLS a SSL, SSH, S/MIME a IPsec. Dále je možné SHA využít i pro kontrolu integrity souborů či ukládání hesel.

Je považována za nástupce hashovací funkce MD5.

9.7.2.1 Zpochybnění bezpečnosti*

Bezpečnost SHA-1 byla poněkud zpochybněna kryptografickými odborníky^[1] nebo například i firmou Google^[2] ačkoli nebyly oznámeny žádné útoky na varianty SHA-2. Varianty SHA-2 jsou algoritmicky stejné s algoritmy SHA-1 (ten v roce 2014 používá 90 % webů). Proto jsou snahy o vývoj vylepšených hašovacích funkcí^{[3][4]} Veřejná výběrová soutěž pro novou SHA-3 funkci byla formálně oznámena 2. listopadu 2007 ve Federal Register^[5] – „NIST usiluje o

zavedení jednoho nebo více nových hašovacích algoritmů pomocí veřejné konkurence stejně jako u vývoje pro Advanced Encryption Standard (AES)^[6].“ Roku 2012 zvítězil algoritmus Keccak.

[1] http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[2] <https://konklone.com/post/why-google-is-hurrying-the-web-to-kill-sha-1> - Why Google is Hurrying the Web to Kill SHA-1

[3] http://www.schneier.com/blog/archives/2005/11/nist_hash_works_4.html

[4] <http://www.heise-security.co.uk/articles/75686/2>

[5] http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf

{title}. www.csrc.nist.gov [online]. [cit. 10-01-2008]. Dostupné v archivu pořízeném dne 05-02-2008.

9.7.2.2 SHA-0

SHA-0, jakož i SHA-1 vytváří 160bitový hash zprávy s délkou $2^{64} - 1$ bitů.

- Založen na stejných principech jako MD4 a MD5
- Vydán 1993 s pův. označ. SHA
- stažen krátce po vydání

9.7.2.3 SHA-1

- Vydán 1995
- liší se od SHA-0 jednou bitovou rotací provedené pomocí jednosměrné funkce¹⁸
- změna podle NSA
- oprava zvyšuje kryptografickou bezpečnost
- 2005: útok na SHA-1 \Rightarrow budoucí použití SHA-1 není bezpečné
- 2017: prohlášeny Microsoftem za zastaralé

¹⁸ funkce, kterou lze snadno vyčíslit, ale je velmi obtížné z výsledku funkce odvodit její vstup. Matematicky: Ze zadaného x tedy lze snadno získat $f(x)$, avšak výpočet inverzní funkce $g^{-1}(x)$ při znalosti $g(x)$, je sice teoreticky možné, ale prakticky velmi obtížné.

9.7.2.4 SHA-2

- Souhrnný název pro čtyři algoritmy: SHA-224, SHA-256, SHA-384, SHA-512
- Vydány 2001
- Podobnost s SHA-1 \Rightarrow jisté riziko(?)
- Dosud bez úspěšných útoků
- SHA-256 – ověřování softwarových balíků linuxové distribuce Debian
- SHA-256 a SHA-512 – DNSSEC (*viz předměty síťového charakteru*)
- Datové schránky v České republice

9.7.2.5 SHA-3 (pův. Keccak [ketʃak])

- Představen, standardizován 2012, na základě soutěže
 - zvítězil v konkurenci s 63 dalšími algoritmy
- Nástupce SHA-1 a SHA-2
- **Autoři:* Guido Bertoni, Joan Daemen, Michaël Peeters a Gilles Van Assche
 - Keccak (catch-ack)
- Běh možný na mnoha různých zařízeních
- Rychlost při implementaci v hardware \Rightarrow
 \Rightarrow při běhu na procesoru *Core 3* má rychlost cca 13 cyklů na byte
- Zcela odlišný princip od SHA-2 \Rightarrow případný průlom by neohrozil bezpečnost obou funkcí

10 Složitost algoritmu

Výběr vhodného algoritmu → v jakém kontextu je zamýšleno algoritmus spouštět?

- „Domácí“ použití, resp. algoritmus / program jen pro osobní účely
- Výuka programování

Upřednostňujeme *jednoduchost* a *srozumitelnost*

Jak přemýšlet ve chvíli, kdy daný algoritmus zamýšlím prodat, příp. publikovat?

Koncový uživatel (klient) se primárně soustředí na to, k čemu má bezprostřední přístup

- Propracované systémy nabídek
- Kontextová nápověda
- Kvalita prezentace výsledků v grafické formě atd.

Málokdy (pokud vůbec) se klient zajímá o vnitřní úpravu programu, srozumitelnost a estetiku použitých algoritmů apod.

Vývojář by tyto aspekty měl zohlednit. V opačném případě riskuje neúspěch produktu na trhu (konkurenceschopnost programu)

Je zřejmé, že i v tomto odvětví se projevuje typický střet zájmů typu výrobce × klient. Zatímco politikou výrobce je vytvořit produkt (software / algoritmus) co *nejlevněji* a prodat co *nejdraž*, základem politiky klienta je získat daný produkt s *nejvyšší kvalitou* s minimálním *nákladem*.

Celá problematika lze zjednodušit do základních kritérií hodnocení programu: (Wróblewski, 2015)

- Způsob komunikace s uživatelem
- Rychlost provádění základních funkcí programu
- Stabilita

Tato kapitola bude vyšetřovat efektivní činnost programů, resp. výpočetní složitost algoritmů, tedy rychlost provádění základních funkcí programu. Ostatní kritéria řeší jiné předměty. Znalosti z této kapitoly nám umožní správně reagovat na otázku *Který ze dvou algoritmů, plnící stejnou úlohu odlišnými způsoby je efektivnější a proč?*

Složitost algoritmu často čerpá ze znalostí středoškolské (příp. vysokoškolské) matematiky. Metody, které si představíme však budou značně zjednodušené a na místo teoretických studií se budeme orientovat spíše na praktické aplikace.

10.1 Definice

Analýza efektivity programů rozlišuje dva základní faktory, které přispívají ke spokojenosti koncových uživatelů:

- Čas provádění – sleduje nároky algoritmu na čas. Jak dlouho trvá výpočet s tímto algoritmem?
- Náročnost na paměť – sleduje nároky algoritmu na paměť. Je schopen provést tento algoritmus můj počítač s omezenou pamětí?

Druhý faktor, a sice náročnost na paměť, postupně ztrácí význam s ohledem na digitální vývoj a s ním související pokles cen.

Základní otázkou analýzy algoritmů je výběr vhodné míry výpočetní složitosti. Zvolená míra pak musí mít stejnou vypovídající hodnotu pro všechny uživatele daného algoritmu bez ohledu na hardwarové vybavení stanice.

Tedy tvrzení, že „*Můj program je rychlý, protože skončil po jedné minutě*“ nemá samo o sobě žádnou vypovídající hodnotu z oblasti analýzy složitosti algoritmu. Aby tvrzení bylo úplné, bylo by nutné přidat další informace: *typ počítače, taktovací frekvence CPU, počet zpracovávaných dat, priorita programu v paměti, kompilátor zdrojového kódu, ...*

Tato analýza je zřejmě dosti nešťastná, je tedy nutná *univerzální metrika*, která nijak nesouvisí s detaily hardwarové povahy.

Čas dokončení daného algoritmu je nejčastěji určen, kromě jiných činitelů, rozměrem dat n , která tento algoritmus zpracovává.

Pojem rozměr dat není jednoznačný. Zatímco pro funkci pro třídění pole to bude jednoduše velikost (délka) pole, v případě programu na výpočet funkce faktoriál se bude jednat o velikost vstupní hodnoty. Podobně funkce, která vyhledává data v seznamu (viz kap. [Chyba! Nenalezen zdroj odkazů.](#) [Chyba! Nenalezen zdroj odkazů.](#), str. [Chyba! Záložka není definována.](#)) bude silně závislá na délce seznamu. Ve všech uvedených případech se jedná o rozměr vstupu dat.

Obecně můžeme říct, že n je nejvýznamnějším parametrem, který ovlivňuje čas dokončení algoritmu.

	10	20	30	40	50	60
n	0,000 01 s	0,000 02 s	0,000 03 s	0,000 04 s	0,000 05 s	0,000 06 s
n^2	0,000 1 s	0,000 4 s	0,000 9 s	0,001 6 s	0,002 5 s	0,003 6 s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
2^n	0,001 s	1,0 s	17,9 min	12,7 dní	35,7 roků	366 stol.
3^n	0,59 s	58 min	6,5 roku	3855 stol.	cca 10^8 stol.	cca 10^{13} stol.
$n!$	3,6 s	768 stol.	cca 10^{16} stol.	cca 10^{32} stol.	cca 10^{48} stol.	cca 10^{66} stol.

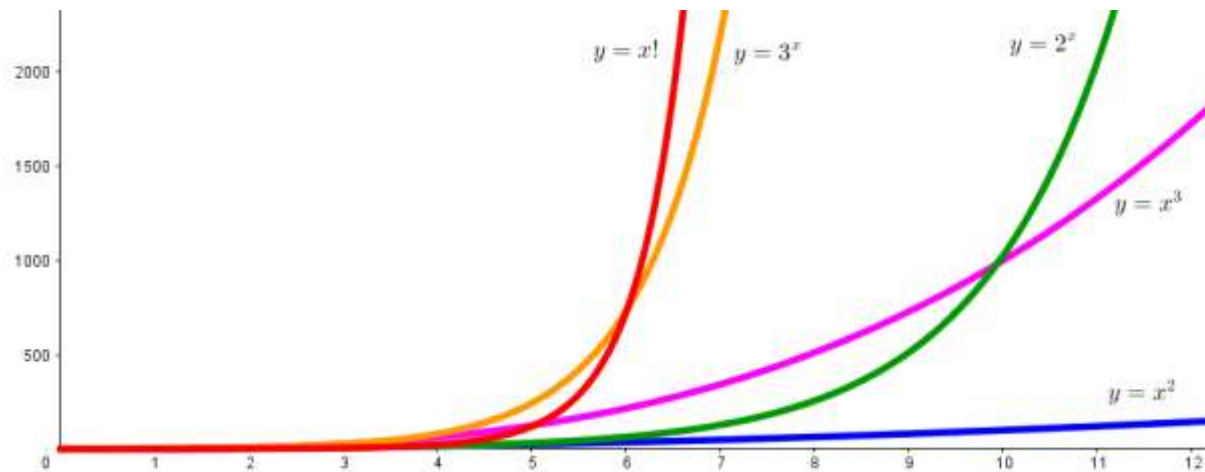
TABULKA 4: ČAS PROVÁDĚNÍ PROGRAMŮ PRO ALGORITMY RŮZNÝCH TŘÍD

Tabulka reflektuje mnoho časových údajů o provádění jednotlivých algoritmů v závislosti na daných předpokladech:

- Čas provádění algoritmu A je úměrný vybrané matematické funkci
např. pro vstupních hodnotu velikosti x a funkci $n!$ Odpovídá čas provádění hodnotě $x!$
- Jedna elementární operace trvá 1 μ s.
např. pro $n = 10$ a funkci $x!$ dostáváme $10! = \frac{3\,628\,800}{1\,000\,000} = 3,6$ s.

Z uvedených předpokladů, pak můžeme dospět k hodnotám, které jsou v tabulce *Tabulka 4 Čas provádění programů pro algoritmy různých tříd*. Zejména pak její pravá dolní část může být značně překvapivá.

Pojďme si z matematiky připomenout základní vlastnosti funkcí. Zejména se zaměříme na růst hodnot funkcí v závislosti na růstu proměnné x . Jak lze vyčíst z obrázku níže, některé funkce mají tendenci růst *rychleji*.



TABULKA 5: GRAFICKÉ ZNÁZORNĚNÍ DOMINANCÍ FUNKCÍ

Osa x reprezentuje rozměr dat n . Osa y představuje čas provedení algoritmu $T(n)$.

Z edukativně-didaktických důvodů je záměrně graf znázorněn pro *malá* n . Pro reálné rozměry dat by grafické znázornění nemělo smysl. Dále z analogických důvodů je zvoleno nestandardní měřítko grafu, a sice: $\frac{x}{y} = \frac{1}{500}!$

Řekneme, že funkce $f(x)$ je dominantní vůči $g(x)$ a píšeme $f(x) \gg g(x)$, pokud platí, že $f(x)$ roste rychleji, než $g(x)$.

$$x! \gg 3^x \gg 2^x \gg x^3 \gg x^2 \gg x \gg \ln x \gg k$$

Jinými slovy:

faktoriálová \gg exponenciální \gg kubická \gg kvadratická \gg lineární \gg logaritmická \gg
 \gg konstantní

Důsledkem našeho zkoumání, pak může být zjištění, že pro třídy s nízkou složitostí lze v praxi řešit komplexnější problémy. Dále toto tvrzení můžeme prohlubovat

jednoduchou simulací algoritmů. U časově nákladnějších funkcí nezískáme zrychlením počítače tolik času, abychom mohli zpracovat více vstupních dat.

Před zrychlením			Po zrychlení (1 000×)	
n	$100n$	2^n	$100n$	2^n
10	1 000	1 024	1	1
20	2 000	1 048 576	2	1 049
30	3 000	1 073 741 824	3	1 073 742
40	4 000	1 099 511 627 776	4	10 955 116 278
50	5 000	...	5	...
100	10 000	...	10	...
1 000	100 000	...	100	...
10 000	1 000 000	...	1 000	...
100 000	10 000 000	...	10 000	...
1 000 000	100 000 000	...	100 000	...

TABULKA 6: KLAMNÁ DŮVĚRA V MOŽNOSTI POČÍTAČŮ A REALITA (WRÓBLEWSKI, 2015)

Tabulka ukazuje, že počet problémů, které lze vyřešit za daný čas silně vzrůstá (o tři řády) v případě programu třídy $100n$, zatímco u druhého algoritmu jich dokážeme zpracovávat pouze o 50 % více. Faktický rozsah 10-20 se rozšiřuje stěží na 10-30, což je v podstatě zanedbatelné.

10.1.1 Faktoriál

! Připomeňte si definici faktoriálu a možnosti výpočtů – kap. 2.3.3.1 Faktoriál, str. 55 v kap. 2 Algoritmy.

Rekurzivní funkce faktoriál v jazyce Java:

```
public static int factorialRek(int number){
    if(number < 0) return -1;
    if(number == 0 || number == 1) return 1;
```

```

    return number * factorialRek(number - 1);
}

```

Pro zjednodušení budeme vycházet z předpokladu, že časově nejnáročnější operací bude v tomto případě podmíněný příkaz `if()` (což je pro úlohy tohoto typu příznačné). Čas provedení tedy můžeme rovněž popsat rekurzivním zápisem:

$$T(0) = t_c,$$

$$T(n) = t_c + T(n - 1) \text{ pro } \forall n \in \mathbb{N}_1.$$

Pro vstupní hodnotu $n = 0$ se čas provedení funkce $T(0)$ rovná času provedení jedné instrukce, kterou symbolicky označujeme t_c .

Pro vstupní hodnoty $n \geq 1$ je čas provedení funkce $T(n)$ dán v souladu s rekurzivním vzorcem $T(n) = t_c + T(n - 1)$.

Výše uvedený zápis je vhodný spíše pro pochopení obecné problematiky, nikoliv pro konkrétní výpočty. Těžko bychom okamžitě dokázali říci, jak dlouho bude trvat výpočet funkce `faktorialrek(100)`. Zkusme na to jít jiným způsobem, a sice si výše uvedený rekurzivní zápis faktoriálu přepsat do obecné soustavy rovnic:

$$T(n) = t_c + T(n - 1)$$

$$T(n - 1) = t_c + T(n - 2)$$

$$T(n - 2) = t_c + T(n - 3)$$

...

$$T(1) = t_c + T(0)$$

$$T(0) = t_c$$

Nyní sečteme levé a pravé strany, dostáváme vztah:

$$T(n) + T(n - 1) + \dots + T(1) + T(0) = (n + 1)t_c + T(n - 1) + T(n - 2) + \dots + T(0)$$

Nechť $w = T(n - 1) + T(n - 2) + \dots + T(0)$, pak:

$$T(n) + w = (n + 1)t_c + w$$

Nyní odečteme w a získáváme závislost: $T(n) = (n + 1)t_c$.

Z výsledku je zřejmé, jakým způsobem ovlivňuje *velikost vstupní hodnoty* počet instrukcí, které program provádí – *čili v důsledku čas provedení algoritmu*. Za předpokladu, že známe parametr t_c a hodnotu n , můžeme velmi přesně určit, za jaký čas dokončí algoritmus svoji činnost. Výsledek těchto přesných výpočtů označujeme termínem praktická složitost algoritmu a značíme symbolem T .

V praxi je však využití praktické složitosti diskutabilní. U *dostatečně velkých* n totiž jsou časy dosti podobné, např. pro $(n + 1)t_c \cong (n + 15)t_c$ apod.

V dalších úvahách tedy budeme primárně hledat typ matematické funkce, který nejlépe vystihuje podstatu algoritmu, resp. takovou matematickou funkci, která nejvýrazněji ovlivňuje čas provádění programu.

Tuto funkci budeme označovat teoretickou složitostí nebo třídou algoritmu. Taková funkce se označuje symbolem \mathcal{O} (tzv. Omikron notace).

Teoretickou složitost algoritmu lze najít dvěma způsoby:

- Lze vycházet z určitých matematických tvrzení a aplikovat na konkrétní situace
- Intuitivní metodou

Intuitivní metoda je v mnoha případech rychlejší a zároveň, pro neznalé vyšší matematiky, značně přístupnější. Pro názornost uvádím nějaké příklady, jak lze z rovnic vyjadřující praktickou složitost $T(n)$ zjistit jeho teoretickou složitost \mathcal{O} .

$T(n)$	\mathcal{O}
6	$\mathcal{O}(1)$
$3n + 1$	$\mathcal{O}(n)$
$n^2 - n + 1$	$\mathcal{O}(n^2)$
$2^n + n^2 + 4$	$\mathcal{O}(2^n)$

Lze pozorovat, že teoretická složitost vybírá z praktické složitosti funkci s nejvyšším dominancí (tedy funkci s nejrychlejším růstem).

10.1.2 Přehled nejčastějších funkcí \mathcal{O} složitosti

Konstantní – třída $\mathcal{O}(1)$

- Rozsah operací prováděných není závislý na rozsahu dat (neznamená to však, že bude malý: číslo 1 je často mylně zaměňováno s jedinou instrukcí)
- Příklad algoritmu: Skok na prvek v poli dle indexu

Logaritmická – třída $\mathcal{O}(\log n)$

- Připomeňme si definici logaritmu: $\log_a n = x \Leftrightarrow a^x = n$
- Aritmetický výklad logaritmické složitosti naznačuje, že pokud n roste geometrickou posloupností, pak výpočetní náročnost bude vzrůstat aritmeticky

Nechť $a_n = 100a_{n-1}$

	a_1	a_2	a_3	a_4	a_5	a_6
n	1	100	10 000	10^6	10^8	10^{10}
$\mathcal{O}(\log n)$	0	2	4	6	8	10

- Pokud n neroste drasticky rychle, algoritmus má sice tendence zpomalovat, nikoliv však výrazně.
- Příklad algoritmu: prohledávání setříděného pole; binární vyhledávání

Lineární – třída $\mathcal{O}(n)$

- Lineární složitost označuje algoritmus, který funguje v čase úměrném velikosti problému (n)
- Příklad algoritmu: sekvenční zpracování znakového řetězce; obsluha fronty; hledání prvku v neseřazeném poli

Kvadratická – třída $\mathcal{O}(n^2)$

- Objevuje se v aritmetických a kombinatorických úlohách, kde se uplatňuje pravidlo „každý s každým“
- Příklad algoritmu: sčítání matic typu $n \times n$ pro „malá“ n .
- *Poznámka*: Analogicky u kubické složitosti $\mathcal{O}(n^3)$ lze jako příklad uvést násobení matic $n \times n$

Exponenciální – třída $\mathcal{O}(2^n)$

- Použitelná jen pro malá n .
- V praxi se algoritmy této složitosti dají používat – pokud vrací výsledky ve snesitelném čase
- Příklad algoritmu: Problém obchodního cestujícího¹⁹ pomocí dynamického programování

Faktoriálová – třída $\mathcal{O}(n!)$

- Příklad algoritmu: Problém obchodního cestujícího hrubou silou

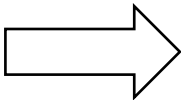
¹⁹ Problém obchodního cestujícího (anglicky Travelling Salesman Problem – TSP) je obtížný diskretní optimalizační problém, matematicky vyjadřující a zobecňující úlohu nalezení nejkratší možné cesty procházející všemi zadanými body na mapě.

10.2 Příklady

10.2.1 Nulování matice

Jak lze vynulovat část dvourozměrného pole, resp. matice pod hlavní diagonálou?

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1



0	1	1	1	1	1
0	0	1	1	1	1
0	0	0	1	1	1
0	0	0	0	1	1
0	0	0	0	0	1
0	0	0	0	0	0

Zápis příslušného kódu v jazyce C++:

```
int tab[n][n];
void nulovani()
{
    int i,j;
    i = 0;                                // ta
    while (i < n){                          // tc
        j = 0;                              // ta
        while (j <= i){                     // tc
            tab[i][j] = 0;                  // ta
            j = j + 1;                      // ta
        }
        i = i + 1;                          // ta
    }
}
```

t_a – čas provedení instrukce přiřazení

t_b – čas provedení instrukce porovnání

Jak funguje cyklus `while()`?

```
j = 0;
while (j <= n){
    instrukce;
    j = j + 1;
}
```

Princip činnosti cyklu `while()` spočívá v n -krát provedení instrukce v těle cyklu.

Podmínka cyklu se tedy provede $(n - 1)$ -krát.

Tedy v reakci na komentáře lze sestavit rovnici pro výpočet složitosti.

Jednu periodu vnitřního cyklu můžeme zapsat jako:

$$t_c + t_a + t_a = t_c + 2t_a$$

Tato perioda se i -krát opakuje, tedy:

$$\underbrace{(t_c + 2t_a) + \dots + (t_c + 2t_a)}_{i - \text{krát}} = \sum_{j=1}^i (t_c + 2t_a)$$

Jednu periodu vnějšího cyklu můžeme zapsat jako:

$$2t_a + 2t_c + \sum_{j=1}^i (t_c + 2t_a)$$

Perioda vnějšího cyklu se opakuje N -krát, tedy:

$$\sum_{i=1}^N \left(2t_a + 2t_c + \sum_{j=1}^i (t_c + 2t_a) \right)$$

A přičítáme instrukce vně cyklu:

$$T(n) = t_c + t_a + \sum_{i=1}^N \left(2t_a + 2t_c + \sum_{j=1}^i (t_c + 2t_a) \right)$$

Sumační notace lze (zaplaťpánbůh) rozumně odstranit. Stačí si uvědomit, že argument sumací je neměnný čili lze argument triviálně násobit. Dostáváme rovnici:

$$T(n) = t_c + t_a + \sum_{i=1}^N 2t_a + 2t_c + i(t_c + 2t_a)$$

Nyní vzpomeňme na vzorec pro výpočet součtu posloupnosti přirozených čísel od 1 do N :

$$1 + 2 + 3 + \dots + N = \frac{N(N+1)}{2}$$

Pomocí tohoto vzorce lze rovnici

$$T(n) = t_c + t_a + \sum_{i=1}^N 2t_a + 2t_c + i(t_c + 2t_a)$$

upravit do tvaru:

$$T(n) = t_c + t_a + 2N(t_a + t_c) + \frac{N(N+1)}{2}(t_c + 2t_a)$$

Za pomoci elementárních úprav zjednodušíme rovnici:

$$T(n) = t_a(1 + 3N + N^2) + t_c\left(1 + 2,5t_c + \frac{N^2}{2}\right)$$

Z toho tvaru je již patrné, že analyzovaný algoritmus má třídu složitosti podle nejrychleji rostoucí funkce – tedy $\mathcal{O}(n^2)$.

11 Umělá inteligence a strojové učení (Oracle Academy)

11.1 Umělá inteligence

a

11.1.1 Data, informace

b

11.1.2 Kategorizace dat

c

11.2 Strojové učení

d

11.2.1 Workflow strojového učení

e

11.2.2 Entropie a ID3 algoritmy

f

12 Poznámky

Zde jsou uvedeny poznámky autora k aktuální verzi učebnice

- Zdroje a citace obrázků!!
- Doplnit anglické (originální) názvy
- Algoritmy – Rekurze: Doplnit informace o hw, omezení RAM, správa paměti SWAP (Ing. Zděněk),
 - Stack overflow!, Cesta na věčnost, Typy rekurzivních algoritmů, Rekurzivní myšlení, Odstranění rekurze (Wroblewski)
- Euklidův algoritmus, rozšířený Euklidův algoritmus

13 Citace a zdroje literatury

b.r.

ALGORTIMY.NET, 2019. Algoritmy.net. *Algortimy.net*.

BENEŠ, Libor a Martin ŠIMEČEK, 2012. Programujte.com. *Vývojové diagramy - Selection, Insert a Bubble sort - 21.díl*. Dostupné také z:

<http://programujte.com/clanek/2012011800-vyvojove-diagramy-selection-insert-a-bubble-sort-21-dil/>

ČÁPKA, David, 2012. UML - Class diagram. *ITnetwork.cz*.

ČÁPKA, David, 2019. *ITnetwork.cz*.

ČÁPKA, David, David ČÁPKA, ed., 2019. *ITnetwork.cz*. *UML state machine diagram*.

HORDĚJČUK, Vojtěch, 2019. *Voho = Vojta Hordějčuk*.

JANČAŘÍK, Antonín, 2014. Diskrétní matematika. *Grafy*.

JIROVSKÝ, Lukáš, 2008. *Vybrané problémy z teorie grafu ve výuce na střední škole*. Praha: Matematicko-fyzikální fakulta UK.

KOLÁŘ, Josef, 2000. *Teoretická informatika*. Praha: Česká informatická společnost.

KOVÁŘ, Petr, 2012. *Úvod do teorie grafů*. Ostrava: Technická univerzita Ostrava.

LAFORE, Robert, 2003. InformIT. *Simple Sorting in Java*.

LESSNER, Dan, 2014. Učíme informatiku. *Rekurzivní kreslení*.

MASEHIAN, Ellips, 2013. ResearchGate. *A solution to the 8-queens problem*.

MATOUŠEK, Radomil, 2006. *Metody kódování*. Brno: FSI VUT v Brně.

NECKÁŘ, Jan, 2015. Algoritmy: příklady algoritmů v jazyce Java, Perl, Python, řešení složitých matematických úloh". *Algoritmy.net*.

PAVUS, , 2005. UML: class diagram - diagram tříd. [Http://mpavus.wz.cz/index.php](http://mpavus.wz.cz/index.php).

PIANDWHIPPEDCREAM, 2007. Wikipedia Commons. *File:SierpinskiTriangle.svg*.

PŘISPĚVOVATELÉ WIKIPEDIE, 2016. Wikimedia Commons. *File:P3A.jpg*.

SEDGEWICK, Robert a Kevin WAYNE, 2007. Javac Queens.java. *Klaus Ostermann*.

SCHMIDT-CORNELIUS, Hanson, 2014. LiveCode. *Recursion for Runaways*.

S, Mikhail, 2018. Wikimedia Commons. *Aktivita_example.png* [online]. Dostupné také z: https://upload.wikimedia.org/wikipedia/commons/c/c2/Aktivita_example.png

STEYN, Barry, 2019. Doctrina. *Maximum Height Of A Red-Black Tree*.

VISUALPARADIGM, 2017. VisualParadigm. *Selection Sort Flowchart Example*.

WIKIPEDIE, Přispěvovatelé, ed., 2018. Wikipedie, otevřená encyklopedie. *Diagram*.

WRÓBLEWSKI, Piotr, 2015. *Algoritmy*. Brno: Computer Press.

YANG, Herong, 2014. UML Tutorials - Horong's Tutorial Examples. *Activity Diagram - Frame Notation and Parameters*.

ZIMMEROVÁ, B., 2008. *PV167 Projekt z obj. návrhu IS*.

14 Rejstřík

Bude vytvořen nakonec

15 Obsah

Bude vytvořen nakonec