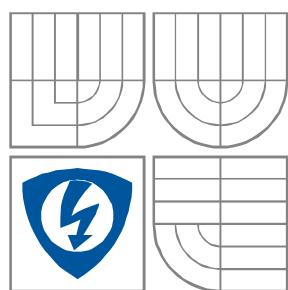


FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



Teoretická informatika

Garant předmětu:
Ing. Radim Burget, Ph.D.

Autoři textu:
Ing. Radim Burget, Ph.D.

BRNO * 2013



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Vznik téhoto skript byl podpořen projektem č. CZ.1.07/2.2.00/28.0096
Evropského sociálního fondu a státním rozpočtem České republiky.

Autor	Ing. Radim Burget, Ph.D.
Název	Teoretická informatika
Vydavatel	Vysoké učení technické v Brně Fakulta elektrotechniky a komunikačních technologií Ústav telekomunikací Purkyňova 118, 612 00 Brno
Vydání	první
Rok vydání	2013
Náklad	elektronicky
ISBN	978-80-214-4897-1

Tato publikace neprošla redakční ani jazykovou úpravou.

Obsah

1 ÚVOD DO PŘEDMĚTU.....	10
1.1 STRUKTURA PŘEDMĚTU.....	10
1.2 POPIS CVIČENÍ	11
1.2.1 <i>Nástroje</i>	11
1.2.2 <i>Zdroje informací pro jazyk JAVA:</i>	11
2 ÚVOD DO JAZYKA UML.....	13
2.1 ÚVOD DO OBJEKTOVĚ ORIENTOVANÉHO PROGRAMOVÁNÍ	13
2.1.1 <i>Třída vs. objekt</i>	14
1.1. <i>Základní pojmy</i>	15
2.2 DRUHY DIAGRAMŮ JAZYKA UML	16
2.2.1 <i>Diagram užití (Use case diagram)</i>	16
2.2.2 <i>Diagram Tříd (Class diagram)</i>	17
2.2.3 <i>Diagramu tříd vs. jazyk JAVA</i>	18
2.2.4 <i>ER diagram (Entity Relationship diagram)</i>	25
2.2.5 <i>Diagram sekvencí (Sequence Diagram)</i>	25
2.2.6 <i>Stavový diagram (State machine diagram)</i>	25
2.3 NÁVRHOVÉ VZORY	26
2.3.1 <i>Jedináček (Singleton)</i>	26
2.3.2 <i>Observer (pozorovatel)</i>	28
2.3.3 <i>Rozhraní (Interface)</i>	30
2.3.4 <i>Abstraktní Továrna (Abstract Factory)</i>	30
2.3.5 <i>Skladba (Composition)</i>	32
2.3.6 <i>Fasáda (Facade)</i>	32
2.4 LITERATURA	33
3 HODNOCENÍ ALGORITMŮ.....	34
3.1 DEFINICE ALGORITMU	34
3.2 HODNOCENÍ ALGORITMŮ.....	34
3.2.1 <i>Základní rády složitosti a jejich kvalifikace</i>	35
3.2.2 <i>Absolutní složitost</i>	36
3.2.3 <i>Asymptotická složitost</i>	37
3.3 PŘÍKLADY	37
3.3.1 <i>Příklad 1</i>	37
3.3.2 <i>Příklad 2</i>	38
3.3.3 <i>Příklad 3</i>	38
3.3.4 <i>Příklad 4</i>	38
3.3.5 <i>Příklad 5</i>	40
3.3.6 <i>Příklad 6</i>	41
3.4 LITERATURA	42

4 ÚVOD DO ABSTRAKTNÍCH DATOVÝCH TYPŮ.....	45
4.1 ÚVOD.....	45
4.2 ABSTRAKTNÍ DATOVÝ TYP.....	45
4.3 REPREZENTACE PROGRAMU V PAMĚTI	46
4.4 OBJEKTY VS. ELEMENTÁRNÍ DATOVÉ TYPY	47
4.5 POLE	48
4.5.1 <i>Vícerozměrná pole</i>	49
4.5.2 <i>Pole objektů</i>	50
4.6 LINEÁRNÍ SEZNAM	51
4.6.1 <i>Jednosměrně vázaný lineární seznam</i>	51
4.6.2 <i>Cyklický jednosměrně vázaný seznam</i>	53
4.6.3 <i>Reprezentace programem</i>	54
4.7 OBOUSMĚRNĚ VÁZANÝ LINEÁRNÍ SEZNAM	54
4.7.1 <i>Reprezentace programem</i>	56
4.8 SROVNÁNÍ ČASOVÉ SLOŽITOSTI POLÍ A JEDNOUSMĚRNĚ A OBOUSMĚRNĚ VÁZANÝCH SEZNAMŮ	57
4.9 FRONTA (FIFO, QUEUE).....	57
4.10 ZÁSOBNÍK (LIFO, STACK)	58
4.11 LITERATURA	58
5 STROMOVÉ DATOVÉ STRUKTURY	59
5.1 ADT STROM	59
5.1.1 <i>Základní pojmy</i>	59
5.1.2 <i>Reprezentace programem</i>	60
5.1.3 <i>Metody průchodu ADT strom</i>	61
5.1.4 <i>Nároky na implementaci</i>	62
5.2 N-ÁRNÍ STROMY	62
5.3 BINÁRNÍ STROMY.....	63
5.3.1 <i>Procházení binárními stromy s využitím rekurezivní funkce</i>	63
5.3.2 <i>Úplný binární strom</i>	64
5.3.3 <i>Binární vyhledávací stromy</i>	64
5.3.4 <i>Vložení prvku</i>	65
5.3.5 <i>Odstranění prvku</i>	65
5.3.6 <i>Implementace binárních vyhledávacích stromů</i>	65
5.3.7 <i>Nevyvážené stromy</i>	67
5.3.8 <i>Vyhledání prvku</i>	67
5.4 AVL STROMY	67
5.4.1 <i>Vkládání prvku</i>	67
5.4.2 <i>Odstraňování prvku</i>	70
5.4.3 <i>Příklady</i>	70
5.4.4 <i>Implementace základních operací ve stromových strukturách</i>	71
5.5 R STROMY	73

5.6 RED-BLACK STROMY	74
5.7 ČASOVÁ A PAMĚŤOVÁ SLOŽITOST	75
5.8 SOUHRNNÉ SROVNÁNÍ	76
5.9 LITERATURA	76
5.10 OTÁZKY K OPAKOVÁNÍ	77
6 TABULKOVÉ DATOVÉ STRUKTURY.....	78
6.1 DATOVÝ TYP TABULKA	78
6.1.1 <i>Tabulka s přímým adresováním</i>	78
6.1.2 <i>Hashovací tabulky</i>	79
6.1.3 <i>Řešení kolizi</i>	80
6.2 JAVA COLLECTIONS FRAMEWORK.....	82
6.2.1 <i>Collection</i>	82
6.2.2 <i>Množina - Set</i>	83
6.2.3 <i>Seznam - List</i>	84
6.2.4 <i>Fronta - Queue</i>	85
6.2.5 <i>Mapování</i>	86
6.2.6 <i>Iterátory</i>	87
6.2.7 <i>Řazení objektů – Rozhraní Comparable</i>	88
6.2.8 <i>Řazení objektů – Komparátory</i>	90
6.3 UKÁZKY IMPLEMENTACE	91
6.4 LITERATURA	92
7 TEORIE GRAFŮ.....	93
7.1 GRAFY.....	93
7.1.1 <i>Základní pojmy</i>	94
7.1.2 <i>Vlastnosti ADT graf</i>	98
7.1.3 <i>Vlastnosti množiny hran</i>	99
7.1.4 <i>Isomorfismus a podgrafy</i>	99
7.1.5 <i>Tranzitivní uzávěr</i>	100
7.2 PŘÍKLADY	101
7.3 ZPŮSOBY REPREZENTACE ADT GRAF.....	102
7.3.1 <i>Maticí sousednosti</i>	102
7.3.2 <i>Vektory sousednosti</i>	102
7.4 PRŮCHOD A VYHLEDÁVÁNÍ V ADT GRAF.....	103
7.4.1 <i>Úvod</i>	103
7.4.2 <i>Slepé prohledávání</i>	104
7.4.3 <i>Informované metody</i>	115
7.5 DALŠÍ VYUŽITÍ VYHLEDÁVÁNÍ V GRAFECH	118
7.6 LITERATURA	118
8 ALGORITMY ŘAZENÍ	120
8.1 ÚVOD	120

8.1.1	<i>Stabilita řadícího algoritmu</i>	121
8.2	PROČ JE NUTNÉ ALGORITMY ŘAZENÍ ZNÁT	121
8.3	JEDNODUCHÉ ŘADÍCÍ ALGORITMY	122
8.3.1	<i>Bubble sort</i>	122
8.3.2	<i>Insertion sort</i>	124
8.3.3	<i>Selection sort</i>	126
8.3.4	<i>Shell sort</i>	127
8.4	POKROČILEJŠÍ ALGORITMY ŘAZENÍ	129
8.4.1	<i>Merge sort</i>	129
8.4.2	<i>Heap sort</i>	130
8.4.3	<i>Quick sort</i>	131
8.5	SROVNÁNÍ ŘADÍCÍCH ALGORITMŮ	135
8.6	PROSTŘEDKY ŘAZENÍ JAZYKA JAVA	136
8.7	LITERATURA	136
9	ALGORITMY VYHLEDÁVÁNÍ	137
9.1	VYHLEDÁVÁNÍ V NESEŘAzeném POLI	137
9.2	VYHLEDÁVÁNÍ V SEŘAzeném POLI	137
9.3	VYHLEDÁVÁNÍ V ADT SEZNAM	139
9.4	VYHLEDÁVÁNÍ V ADT BINÁRNÍ VYHLEDÁVACÍ STROM	139
9.5	VYHLEDÁVÁNÍ V ADT HASHOVANÍ TABULKA	140
9.6	VYHLEDÁVÁNÍ V ADT GRAF	140
9.7	LITERATURA	141
10	OPTIMALIZACE	142
10.1	METODY PŘíméHO VYHLEDÁVÁNÍ	142
10.1.1	<i>Metoda Monte Carlo</i>	142
10.1.2	<i>Horolezecký algoritmus</i>	142
10.2	GENETICKÉ ALGORITMY	142
10.2.1	<i>Demonstrace efektivity genetických algoritmů</i>	143
10.2.2	<i>Základní pojmy</i>	144
10.2.3	<i>Mutace</i>	144
10.2.4	<i>Operace křížení</i>	145
10.2.5	<i>N-Bodové křížení (N-point crossover)</i>	145
10.2.6	<i>Rovnoměrné křížení (Uniform crossover)</i>	145
10.2.7	<i>Křížení vs. Mutace</i>	145
10.2.8	<i>Přirozený výběr</i>	146
10.2.9	<i>Genetické operátory</i>	147
10.2.10	<i>Uplatnění</i>	149
10.3	GENETICKÉ PROGRAMOVÁNÍ	150
10.3.1	<i>Základní princip</i>	150
10.3.2	<i>Zvyšování výpočetní výkonnosti</i>	151

10.3.3 Automaticky definované funkce	151
10.4 LITERATURA	153
11 VYČÍSLITELNOST A SLOŽITOST	155
11.1 ÚVOD	155
11.2 ZÁKLADNÍ POJMY	155
11.3 TURINGŮV STROJ.....	156
11.3.1 Univerzální Turingový stroj	157
11.3.2 Nedeterministický Turingův stroj	158
11.3.3 Paralelní Turingův stroj.....	158
11.3.4 Kvantový Turingův stroj.....	158
11.4 CHURCH-TURINGOVA TEZE.....	159
11.5 NEVYČÍSLITELNÉ PROBLÉMY	160
11.5.1 Rozhodovací problém.....	160
1.2. Nerozhodnutelný problém	161
11.5.2 Problém zastavení TS	161
11.6 TŘÍDY SLOŽITOSTI P, NP	162
11.6.1 NP-úplné problémy	164
11.6.2 Problém ekvivalence P vs. NP.....	164
11.7 LITERATURA	164
12 ANALÝZA DAT	166
12.1 ÚVOD	166
12.2 VSTUP: FORMÁTOVÁNÍ VSTUPNÍCH DAT.....	168
12.2.1 AML formát dat	171
12.2.2 Řídka data	172
12.2.3 Typy atributů	172
12.2.4 Chybějící hodnoty.....	174
12.2.5 Nepřesné hodnoty.....	174
12.3 VÝSTUP: REPREZENTACE ZNALOSTÍ	174
12.4 VĚROHODNOST: ZHODNOCENÍ, CO BYLO NAUČENO	175
12.5 TRÉNOVÁNÍ A TESTOVÁNÍ	176
12.5.1 Cross-validation a její varianty	177
12.5.2 Bootstrap	177
12.6 SROVNÁNÍ KVALITY UČÍCÍCH ALGORITMŮ.....	178
12.6.1 ROC křivky	180
12.6.2 Lift grafy.....	183
12.7 UČÍCÍ SE ALGORITMY	186
12.7.1 Rozhodovací stromy	187
12.7.2 Algoritmy podpůrných vektorů	189
12.7.3 k nejbližším sousedů (k-NN, K Nearest-Neighbor).....	191
12.7.4 Bayesovské sítě	192

12.7.5	<i>Neuronové sítě (Vícevrstvá perceptronová síť)</i>	193
12.8	ČÍSELNÁ PREDIKCE	193
12.9	SHLUKY	194
12.10	TRANSFORMACE	194
12.11	VÝBĚR ATRIBUTŮ (PŘÍZNAKŮ)	194
12.12	NORMALIZACE	194
12.13	KOMBINOVÁNÍ NĚKOLIKA MODELŮ	194
12.14	DALŠÍ INFORMACE	195
12.14.1	<i>Učení z rozsáhlých objemů dat</i>	195
12.14.2	<i>Dolování znalostí z textu a webu</i>	195
12.14.3	<i>Vizualizace</i>	196
12.14.4	<i>Dolování znalostí z multimediálních dat</i>	196
12.14.5	<i>Mezinárodní soutěž v dolování znalostí z báze dat</i>	196
12.14.6	<i>Nástroje</i>	196
12.15	LITERATURA	196

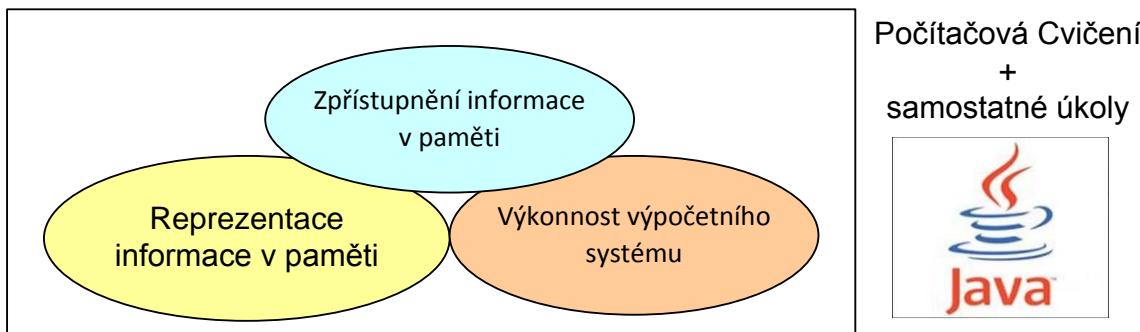
1 Úvod do předmětu

Toto skriptum slouží pro výuku předmětu Teoretická informatika, magisterského oboru Informační a výpočetní technika Fakultě elektrotechniky, Vysokého učení technické v Brně. Toto skriptum pokrývá oblast problematiky: 1) reprezentace informace, 2) zpřístupnění informace a 3) časové hledisko zpřístupnění informace v paměti počítače.

Teoretická informatika je rozsáhlý vědní obor, který se týká širokého spektra problematiky týkající se informace. Předmět má odlišné osnovy nežli předmět Teoretická informatika na Fakultě informatiky či také Teoretická informatika vyučovaná na Masarykově Univerzitě. Důvodem je, že vaše studium je zaměřeno jiným směrem a Fakulta elektrotechniky vychovává odborníky s (částečně) jiným zaměřením a důrazem. Přestože má předmět ve svém názvu slovo „teoretická“, je problematika tohoto kurzu probírána nejen z pohledu teorie, ale jsou současně i prakticky ověřena na vybraných příkladech. Zvolená platforma pro praktické ověření nabytých znalostí je programovací jazyk JAVA. Přestože je ale vše probíráno na konkrétním jazyce, probíraná látka je na jazyku nezávislá a je možné ji aplikovat jak v ostatních jazycích, jako je C, C++, C#, Ruby, atd., ale také v problematice návrhu hardware.

1.1 Struktura předmětu

Předmět se opírá o 3 stěžejní myšlenky, jsou to: jak zpřístupnit informaci v informačním systému (IS), jak reprezentovat informaci v IS a jak ovlivnit jejich výkonnost IS. Počítačová cvičení a samostatně řešené úkoly budou v programovacím jazyce JAVA (viz Obr. 1).



Obr. 1: Struktura předmětu

Na Obr. 2 je k vidění souvislost jednotlivých kapitol vůči témtu tematickým bodům. A na posledním obrázku Obr. 3 je k vidění diagram souvislostí s některými ostatními vědními obory, se kterými MTIN souvisí. Na **Obr. 3** je potom znázorněna souvislost s jinými tematickými celky v oblasti počítačové vědy.

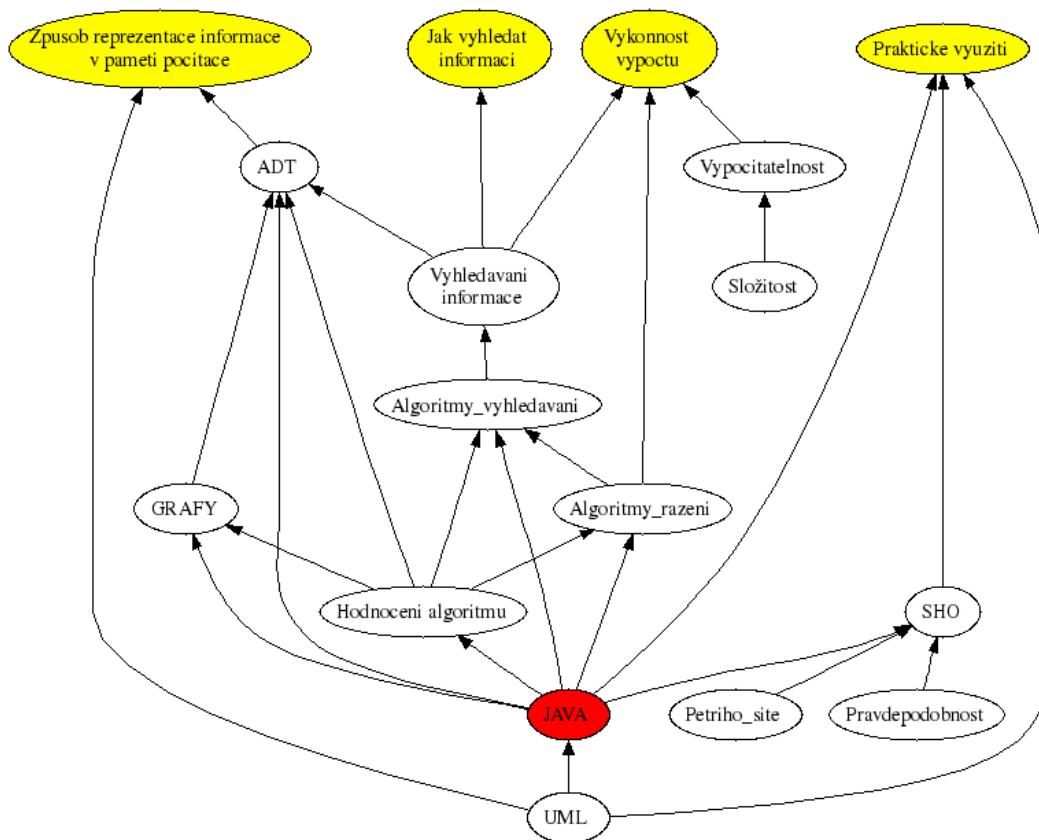
1.2 Popis cvičení

1.2.1 Nástroje

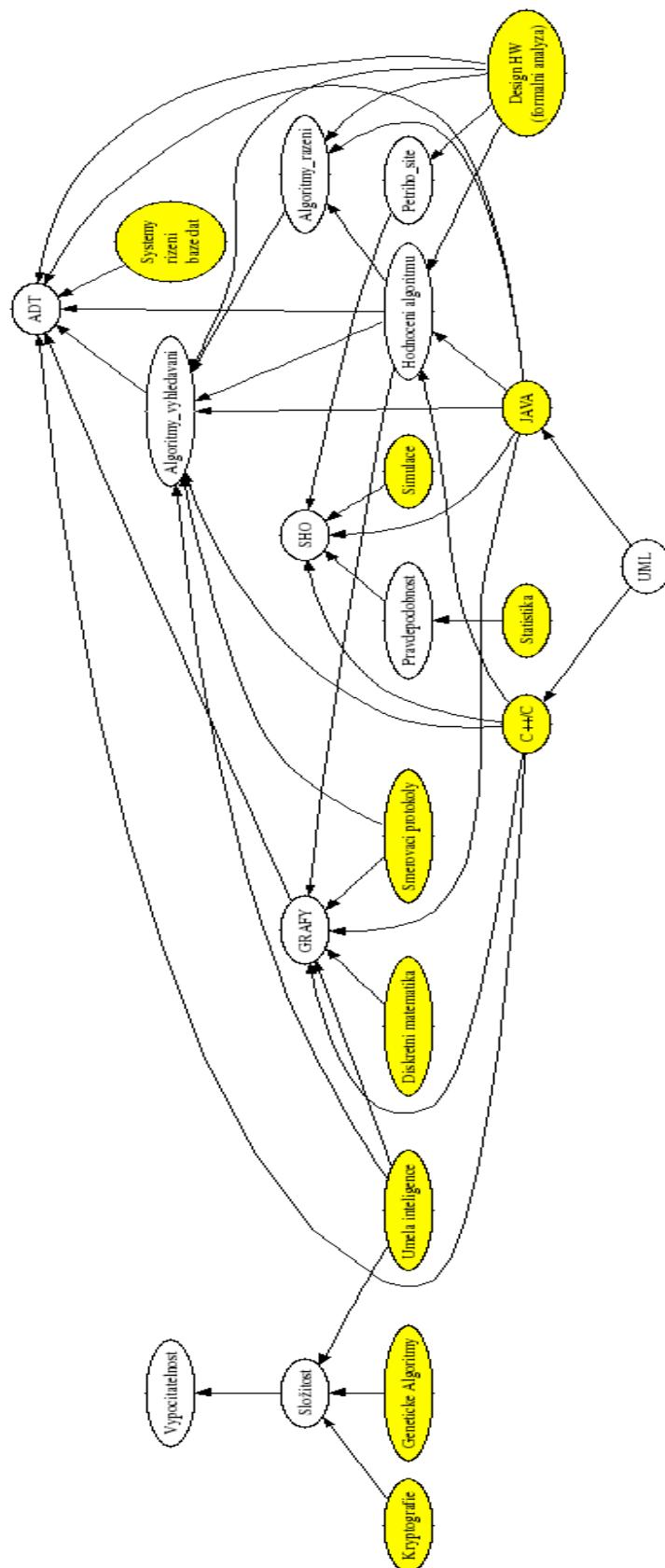
- JDK (Java Development Kit)
<http://java.sun.com/javase/downloads/index.jsp>
- Eclipse IDE for Java Developers,
<http://www.eclipse.org/downloads/>

1.2.2 Zdroje informací pro jazyk JAVA:

- Dřívější předměty Bc. studia
- <http://java.sun.com/docs/books/tutorial/>
- Brett Spell, Java programujeme profesionálně
- Bruce Eckel, Myslíme v jazyce JAVA I, II



Obr. 2: Souvislost jednotlivých kapitol, probíraných v rámci předmětu.



Obr. 3: Souvislost jednotlivých probíraných tematických celků s ostatními vědními disciplínami v počítačové vědě.

2 Úvod do jazyka UML

UML je zkratkou pro Unified Modeling Language, což je **grafický jazyk pro vizualizaci, specifikaci, návrh a dokumentaci programových systémů**.

Důvody proč je dobré znát jazyk UML jsou hned dva. V rámci předmětu MTIN budeme na odborné úrovni komunikovat ohledně struktury projektů a programů a struktur. To se dá provést v každém případě prostřednictvím zdrojových kódů, nicméně raději nežli toto dělat prostřednictvím zdrojového kódu, který je určen pro komunikaci člověk-počítač, je vhodnější užití UML, což je grafickou alternativou určenou **pro komunikaci člověk-člověk**. Pokud se jí člověk naučí rozumět, je neocenitelným nástrojem v komunikaci. Druhým důvodem je vysoká pravděpodobnost, že jako absolventi technického oboru zaměřeného na informační a telekomunikační technologie se řada z vás v praxi stane součástí nějakého vývojového týmu, kde se touto formou bude komunikovat. V průmyslu u nás i ve světě se jazyk UML používá ve většině společností, které vyvíjí na objektové úrovni a myslí to s vývojem vážně. Atž z pohledu vývojáře, analyтика či vedoucího týmu se s programováním budete přímo či nepřímo setkávat bude nezbytné znát alespoň základy takových systémů.

Pro zevrubný popis vlastností jazyka UML bychom si vystačili v rámci celého jednoho semestrálního kurzu. V této přednášce budou představeny typy diagramů, se kterými se setkáte nejčastěji.

Při každém vytváření modelu mějte na paměti, že hlavní myšlenkou jazyka UML je, aby vše bylo jednoduché a snadno pochopitelné. Nemusíte se řídit přesně daným standardem, nicméně vše musí být dobře pochopitelné a neměly by vaše diagramy zmást jinou osobu, která je s tímto jazykem seznámena. Z toho důvodu by základní schéma vždy mělo být dodrženo.

K notaci jazyka UML existuje dostatek literatury jak v knižní formě [1], [2], [3], [5], [6], [7], [8] tak ve formě elektronické [5] a to v anglickém jazyce ale i v českém. Existují také nástroje a to jak komerční tak zdarma dostupné pro vytváření diagramů [9], [10].

Tato kapitola předpokládá alespoň základní orientaci v programovacích jazycích a pojmy objektového programování a jejím cílem je seznámit Vás s relevancí zdrojového kódu vůči UML diagramů a některými základními technikami pro návrh programů. Po té, co si představíme nějaké základní struktury, představíme si i nějaké základní pravidla, jak je dobré programy psát. I přestože každý program je svou podstatou unikátní a má různé požadavky, za roky praxe se ustálily některá pravidla, o kterých se dozvítě něco více. Ty vám mohou pomoci psát kvalitnější programy a v budoucnu snáze porozumět i cizímu kódu.

2.1 Úvod do objektově orientovaného programování

Základním pojmem pro objektově orientované programování (OOP) je **třída a objekt**. Objekt je abstraktní reprezentant nějakého reálného prvku, „pamatuje“ si svůj stav (v podobě dat čili **atributů**) a poskytuje rozhraní **operaci**, aby se s ním mohlo pracovat (nazývané **metody**). Při používání objektu nás zajímá, **jaké operace (služby) poskytuje**, ale ne, jakým způsobem to provádí - to je princip **zapouzdření**. Díky tomu dosahujeme vysoké míry abstrakce.

Programátor, uživatel některé již existující třídy, jediné co musí znát je rozhraní této třídy. Například: třída MPEG4Encoder bude mít veřejné rozhraní s metodami Encode(File YUVFile, File MPEGFile), kde programu předáváme informaci o vstupním souboru, výstupním souboru a nic dalšího nemusíme znát. Nemusíme se zabývat I-Framy P-Framy či B-Framy. Toto vše je implementováno v rámci třídy, ale jako uživatelé této třídy nic o tom nemusíme znát. Tedy alespoň do doby, dokud nechceme na funkci třídy něco změnit (opravit chybu, vylepšit, urychlit atd.).

Abstrakce objektu, která v architektuře programu podchycuje na obecné úrovni podstatu všech objektů podobného typu, se nazývá **třída**. **Třída je předpis, jak vyrobit objekt daného typu**. Například třída Student (chápejme ji jako objekt) má nějaké jméno a příjmení, má bydliště, umí navštěvovat přednášky a cvičení. Totéž platí i pro přednášejícího. Při modelování těchto dvou tříd, tedy Student a Přednášející, mohu generalizovat společné vlastnosti a díky této abstrakci vytvořit obecnou třídu Člověk, která bude mít **atributy** jméno a příjmení (obojí je nějaký řetězec znaků) a **metody** chodit na přednášky a cvičení. Pomocí relace **dědičnosti** potom mohou třídy Student a Přednášející dědit popřípadě přidat další vlastnosti třídy, které již platí výhradně pro ni (např. zadej půlsemestrální test).

2.1.1 Třída vs. objekt

V rámci předmětu se budete velice často setkávat s pojmy třída, objekt a je proto nutné uvědomit si rozdíl mezi těmito dvěma pojmy. **Třídou rozumíme popis, na základě kterého lze vytvořit objekt** (neboli instanci třídy). Třída sama o sobě nezabírá v paměti běžícího programu **žádný prostor**. Objektů nějaké třídy **může existovat libovolný počet**, ale k jakémukoli objektu existuje pouze jedna třída (v rámci dědičnosti tříd se můžeme setkat s tzv. vícetypovostí objektu).

Příklad třídy:

```
class Osoba {
    public String firstName;

    public String surname;

}
```

V třídě je definováno, že její atributy jsou jméno a příjmení a žádné metody.

V tomto případě se k atributům přistupuje přímo, nikoli prostřednictvím metod. Důvod je kvůli přehlednosti a jednoduchosti. V profesionálně navrženém programu by se toto vyskytnout nemělo a přináší to v případě ladění a hledání chyb programu nemalé komplikace, zejména tedy u velkých aplikací.

Příklad vytvoření a použití objektů typu třída:

```
// vytvoreni prvního objektu
Osoba o1 = new Osoba();
o1.firstName = "Jan";
o1.surname = "Novák";

// vytvoreni dalšího objektu
```

```

Osoba o2 = new Osoba();
o2.firstName = "Karel";
o2.surname = "Novák";

// vypis do konzole
System.out.println("Osoba1:" + o1.firstName + " "
+ o1.surname);
System.out.println("Osoba2:" + o2.firstName + " "
+ o2.surname);

```

Objekty si po dobu svého života pamatují svůj stav. V tomto případě své jméno a příjmení. Objektů může existovat libovolný počet, zatímco všechny mohou být stejného datového typu.

1.1. Základní pojmy

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

Abstrakce – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skřínka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

Zapouzdření – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistence. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

Skládání – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

Dědičnost – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).

Polymorfismus – odkazovaný objekt se chová podle toho, jaký je jeho skutečný typ. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší. V praxi se tato vlastnost projevuje např. tak, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy (třídy, která přímo či nepřímo z této třídy dědí), která se může chovat jinak, než by se chovala instance rodičovské třídy, ovšem v rámci mantinelů, daných popisem rozhraní.

Konstruktor – konstruktor je speciální metoda (její název je identický s názvem třídy a nevrací žádný datový typ), která je spuštěna vždy při vytvoření nového objektu.

Atribut – je to proměnná, jejíž platnost je na úrovni objektu. Udržuje stav objektu a je sdílena mezi metodami.

Metoda – funkce, která je definována na úrovni třídy. V plně objektově orientovaných jazycích se lze setkat pouze s tímto druhem funkcí.

Některé z těchto vlastností jsou pro OOP unikátní, jiné (např. abstrakce) jsou běžnou vlastností i jiných vývojových metodik. OOP je někdy označováno jako programátorské

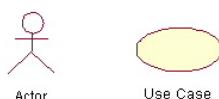
paradigma, neboť popisuje nejen způsob vývoje a zápisu programu, ale i způsob, jakým návrhář programu o problému přemýšlí.

2.2 Druhy diagramů jazyka UML

Diagramy jsou nejznámější a nejpoužívanější částí standardu. Následuje přehled diagramů v UML 2.0 včetně jejich rozčlenění do skupin:

- **strukturní diagramy:**
 - **diagram tříd (class diagram)**
 - diagram komponent (component diagram)
 - composite structure diagram
 - diagram nasazení (deployment diagram)
 - diagram balíčků (package diagram)
 - **diagram objektů** (object diagram), též se nazývá diagram instancí
- **diagramy chování:**
 - diagram aktivit (activity diagram)
 - **diagram užití (use case diagram)**
 - stavový diagram (state machine diagram)
 - diagramy interakce:
 - **sekvenční diagram** (sequence diagram)
 - diagram komunikace (communication diagram, dříve collaboration diagram)
 - interaction overview diagram
 - diagram časování (timing diagram)

2.2.1 Diagram užití (Use case diagram)

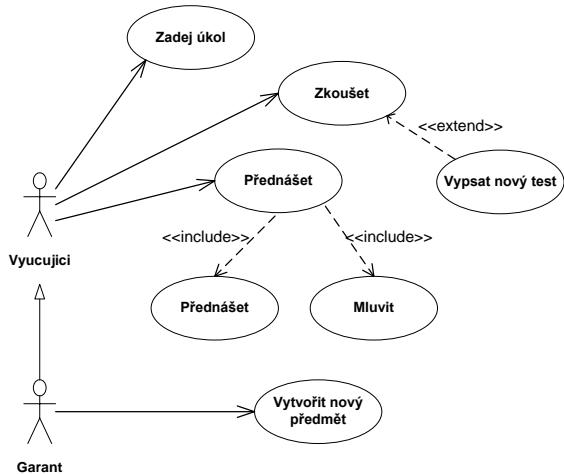


Obr. 4: Prvky užívané v diagramech užití

Účastník (actor) zastupuje nějaký druh role, který v systému vystupuje (například v IS FEKT – student, učitel, garant, administrátor, atd.).

Diagramy užití jsou používány v první fázi vývoje systému analytikem a slouží pro uvědomění si veškerých souvislostí a požadavků na systém. Díky své názornosti a jednoduchosti je vhodný pro komunikaci se zákazníkem, který není s notací UML seznámen vůbec.

Vazba, která existuje mezi účastníky Vyucujici a Garant se nazývá dědičnost či generalizace a význam je takový, že Garant přejímá veškeré vlastnosti nadřazeného prvku.



Obr. 5: Příklad jednoduchého diagramu užití.

V diagramu se také nalézají vztahy <<include>>, <<extend>>. Vztah <<extend>> použijeme pro vyjádření faktu, že případ užití může být **nepovinně rozšířen o chování z jiného případu užití**. Zdůrazňuji slovo nepovinně, kterým se od <<include>> liší. Vztah <<include>> naopak použijí v případě, že se jedná o **povinnou součást**, kterou například sdílí více případů užití. Všimněte si také, že mají opačnou orientaci.

2.2.2 Diagram Tříd (Class diagram)

Základním stavebním kamenem diagramu tříd je třída samotná (viz Obr. 6). Je zde uveden název třídy, atributy této třídy a jejich viditelnost. Dále potom metody a jejich viditelnost. V případě viditelnosti se používá buď grafická reprezentace (viz Obr. 6) anebo se velice často používají značky: pro vyjádření **privátní viditelnosti (-)**, **chráněné (#)**, **veřejné (+)**.

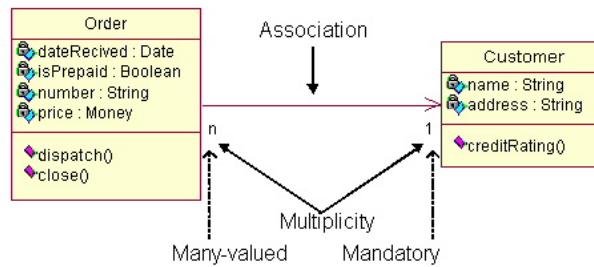


Obr. 6: Příklad třídy v notaci jazyka UML

Mezi třídami mohou být **asociace** (viz Obr. 7). Asociace je **vztah mezi dvěma třídami** a dokonce může být pojmenována. Ke každé asociaci může být přidána **násobnost**, která může nebýt libovolných hodnot. Nejběžnější případy jsou 1, 0..1, 1..n, n. Číslo n zde v tomto případě značí libovolný počet a často se používá také symbol „*“.

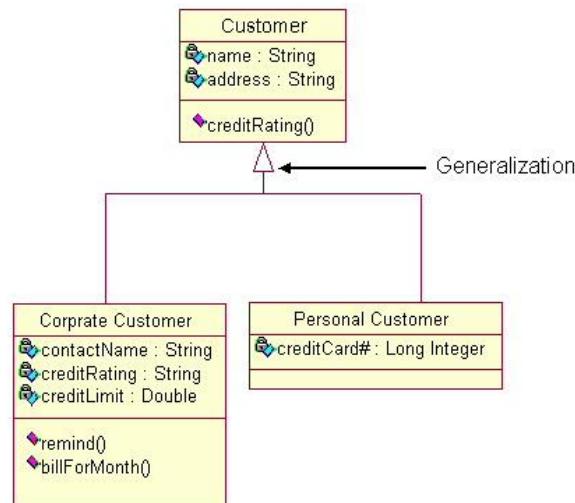
Všimněte si, že informace o násobnosti platí do kříže – tedy, že pro jednoho zákazníka existuje n objednávek, zatímco k objednávce je asociován vždy pouze jeden konkrétní zákazník.

Asociace může být orientovaná, či neorientovaná. V našem případě se jedná o asociaci orientovanou, tj. z třídy Order (objednávka) ke třídě Customer (zákazník). Tato informace nám říká, že vazbu bude udržovat pouze třída Order a nikoli třída Customer. Kromě jednoduché asociace existuje ještě agregace (viz Obr. 14) a kompozice (viz Obr. 15).



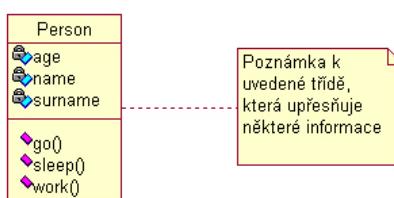
Obr. 7: Asociace mezi objekty a jejich kardinalita

Poslední vztah, který mezi třídami může existovat, je **vztah dědičnosti**. Třída Customer (zákazník), ze které dědí třídy CorporateCustomer a Personal Customer říká, že tyto dvě třídy budou dědit vlastnosti z třídy Customer – tedy jméno a adresu. Tento přístup je velice výhodný z toho důvodu, že tím **redukujeme množství duplicitního kódu na minimum** a tím zajistíme i menší chybovost. Vztah dědičnosti je znázorněn na Obr. 8.



Obr. 8: Vztah dědičnost (generalizace) mezi objekty

V případě složitějších diagramů a asociací může nastat potřeba přidat komentář k určitému prvku. Poznámka se vklásá s pomocí speciálního čtverce se zahnutým rohem (viz Obr. 9), relevance k místu této **poznámky se dle standardu značí přerušovanou čarou**.

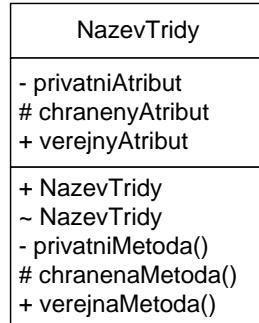


Obr. 9: Poznámka v notaci jazyka UML

2.2.3 Diagramu tříd vs. jazyk JAVA

Cílem této kapitoly je lépe objasnit souvislost mezi zdrojovým kódem a notací jazyka UML. Na Obr. 10 je znázorněna třída s názvem „NazevTridy“. Všimněte si, že nyní je použita

varianta s textově vyjádřenou násobností. Níže následuje zdrojový kód, který odpovídá tomuto diagramu.



Obr. 10: Třída ke zdrojovému kódu v notaci jazyka UML, diagramu tříd.

```

public class NazevTridy {
    NazevTridy() {
        // konstruktor bez argumentu
    }
    /*
     * POZN: Jazyk JAVA nevyžaduje používání
     * destrukturů, proto zde v kódu ani není uveden.
     * Pokud budete nutně potřebovat destruktur i v
     * jazyce java, použijte metodu finalize.
     *
     * Destruktory jsou užívány velice často v jazyce C++.
     */
    private String privatniAtribut;
    protected int chranenyAtribut;
    public int verejnyAtribut;

    public void verejnaMetoda() {
        // nějaký kód
    }
    private int privatniMetoda() {
        // nějaký kód
    }
    protected int chranenaMetoda() {
        // nějaký kód
    }
}
  
```

K tomu, abychom si mohli vysvětlit další asociace mezi třídami, je nutné, abychom mírně předběhli a použili některý z abstraktních datových typů [11] pro reprezentaci kolekce prvků. K tomu použijeme třídu Vector, která je součástí standardu jazyka JAVA. Níže je demonstrováno použití této třídy:

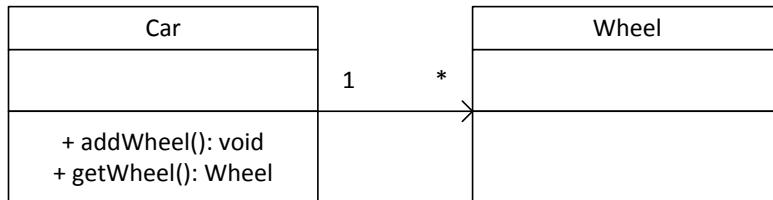
```

Vector<Item> itemVector = new Vector<Item>();
Item i = new Item();
itemVector.add(i); // vloží prvek na konec
itemVector.add(new Item()); // vloží prvek na konec
  
```

```
itemVector.removeElement(i); // odstrani prvni vyskyt objektu
```

2.2.3.1 Orientovaná asociace - násobnost 1:n

Na obrázku Obr. 11 je znázorněna orientovaná asociace mezi dvěma třídami. Níže je pak odpovídající zdrojový kód v jazyce JAVA. Díky tomu, že asociace je orientovaná, bude reference na asociované objekty třídy Wheel obsahovat pouze třída Car.



Obr. 11: Asociace 1:N v notaci jazyka UML

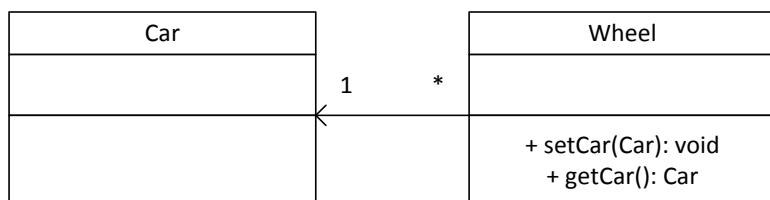
```

import java.util.Vector;
public class Car {
    private Vector<Wheel> wheelList = new Vector<Wheel>();
    public void addWheel(Wheel wheel) {
        wheelList.add(wheel);
    }
    public Wheel getWheel(int i) {
        return wheelList.get(i);
    }
}
public class Wheel {
}
  
```

Všimněte si, že přestože je v kódu atribut, třída Car žádný atribut vyznačen nemá. Je to z toho důvodu, že tento atribut je reprezentován orientovanou asociací 1:N mezi třídami Car a Wheel.

2.2.3.2 Orientovaná asociace - násobnost n:1

Násobnost N:1, jak je vyobrazena na Obr. 12, říká, že pro třídu Car (Auto) existuje libovolný počet kol. V tomto případě by bylo možné omezit maximální počet, popřípadě i pevně určit konkrétní číslo, nicméně i tento návrh je zcela vyhovující. Díky tomu, že asociace je orientovaná, tak referenci na objekty bude udržovat pouze třída Wheel (Kolo).



Obr. 12: Asociace N:1 v notaci jazyka UML

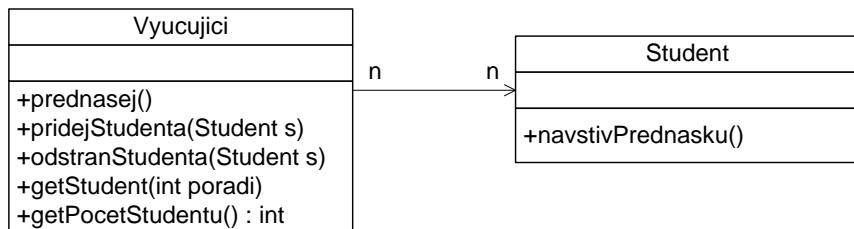
```

public class Wheel {
    private Car car;
    public Car getCar() {
        return car;
    }
    public void setCar(Car car) {
        this.car = car;
    }
}
public class Car {
    // auto nedisponuje referenci na kolo
}

```

2.2.3.3 Orientovaná asociace - násobnosť N:N

Asociace typu N:N, jak je znázornená na Obr. 13, značí, že pro jakýkoli objekt Vyucujici může existovat libovolný počet objektů typu Student a naopak, pro jeden objekt typu Student může existovat libovolný počet objektů typu Vyucujici.



Obr. 13: Asociace typu N:N

```

import java.util.Vector;

public class Vyucujici {
    private Vector<Student> studentList= new Vector<Student>();
    public void pridejStudenta(Student student) {
        studentList.add(student);
    }
    public void odstranStudenta(Student student) {
        studentList.remove(student);
    }
    public Student getStudent(int i) {
        return studentList.get(i);
    }
    public void prednasej() {
    }

    public int getPocetStudentu() {
        return studentList.size();
    }
}
public class Student {
    // student nedisponuje referenci na vyucujiciho
    public void navstivPrednasku() {
        // nejaký kod
    }
}

```

```
}
```

2.2.3.4 Neorientovaná asociace - násobnost N:N

U neorientovaných asociací platí, že **každá strana disponuje referencemi na obou stranách**. Je tedy kombinací dříve zmíněných případů.

Opět existují následující tři typy:

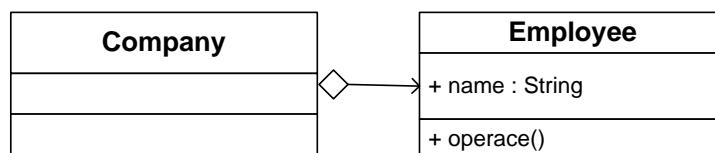
- násobnost 1:n
- násobnost n:1
- násobnost n:n

2.2.3.5 Agregace

Na Obr. 14 je vyobrazena agregace třídy Employee (Zaměstnanec) třídou Company (Společnost). Agregace má velmi podobný význam jako asociace (orientovaná či neorientovaná). V tomto případě ale posilujeme význam vztahu, kde Company obsahuje objekty typu Employee.

Namísto agregace může být vždy použita obyčejná asociace a nebude to chybou, pokud ale použijeme agregaci, agregující třída musí být výhradním správcem aggregovaných objektů.

Stejně jako u agregace je možné, aby vztahy měly název a násobnost.



Obr. 14: Příklad asociace typu agregace v notaci jazyka UML

```

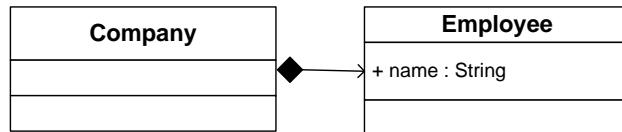
class Employee {
    private String name;
    // :
}

class Company {
    private Employee[] employees;
    // :
}
  
```

Je to forma asociace, jež vyjadřuje vztah celek – část. Element může přetrvat v paměti i po jejím odstranění z kontejneru, příp. stát se součástí jiného kontejneru.

2.2.3.6 Kompozice

Kompozice (viz Obr. 15) je ještě silnější obdobou agregace. V tomto je třída definována přímo v těle předchozí třídy.



Obr. 15: Kompozice třídy v notaci jazyka UML

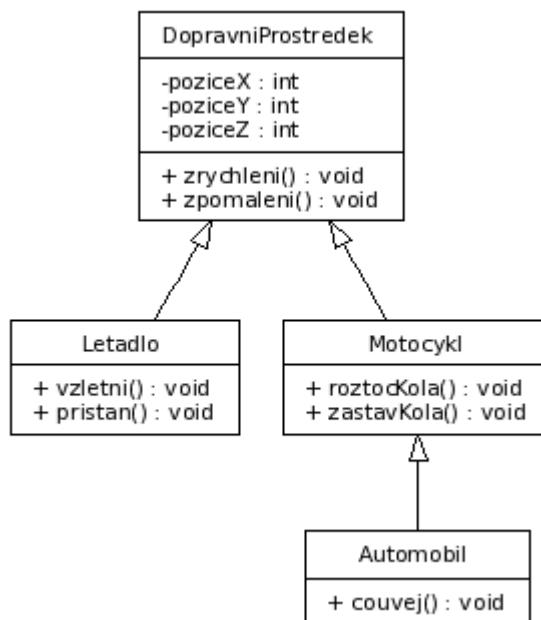
```

class Company {
    private Employee[] employees;
    // :
    class Employee {
        private String name;
        // :
    }
}
  
```

Zrušením kontejneru automaticky zrušíme i obsažený element. Daný element může být součástí právě jednoho kontejneru.

2.2.3.7 Dědičnost a generalizace

Díky generalizaci může několik tříd sdílet stejnou vlastnost. Výhody, které to přináší, jsou eliminace opakování kódu, přehlednější architektura a větší odolnost vůči chybám. Generalizace je vyobrazena na Obr. 16 a níže následuje odpovídající programový kód.



Obr. 16: Dědičnost a generalizace v notaci jazyka UML

```

public class DopravniProstredek {
    private int poziceX;
  
```

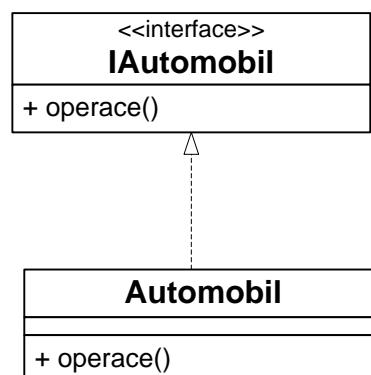
```

    private int PoziceY;
    private int poziceZ;

    public void zrychleni() {
        // nejaký kod
    }
    public void zpomaleni() {
        // nejaký kod
    }
}
public class Motocykl extends DopravniProstredek {
    public void roztocKola() {
        // nejaký kod
    }
    public void zastavKola() {
        // nejaký kod
    }
}
public class Automobil extends Motocykl {
    public void coupej() {
        // nejaký kod
    }
}
public class Letdalo extends DopravniProstredek {
    public void vzletni() {
    }
    public void pristan() {
    }
}

```

2.2.3.8 Realizace



Obr. 17: Vztah implementace

Realizace je souhrn všech veřejně přístupných metod dané třídy. Umožňuje vícenásobné využití operací, aniž bychom museli zavádět dědičnost mezi třídami.

2.2.3.9 Závislost

Závislost je vztah mezi dvěma elementy modelu, v němž změna jednoho (nezávislého) elementu ovlivní druhý (závislý) element.



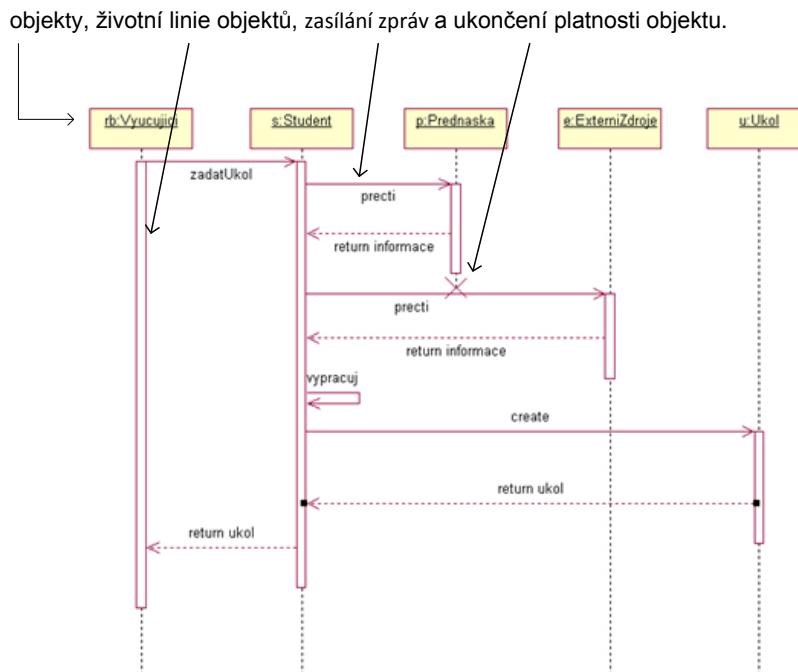
Obr. 18: Vztah závislost

2.2.4 ER diagram (Entity Relationship diagram)

ER-diagram je téměř nerozeznatelnou dvojčetem diagramu tříd. Jediný rozdíl, který mezi těmito dvěma typy diagramů je, jsou data, která reprezentuje. Zatímco v případě diagramu tříd se jednalo o vzájemné vztahy tříd, v tomto případě se jedná o vztahy mezi perzistentně ukládanými daty prostřednictvím databází. Neexistují zde metody. Vzájemné asociace mezi třídami vyjadřují cizí klíče a atributy entity vyjadřují názvy položek v databázi.

2.2.5 Diagram sekvencí (Sequence Diagram)

Na Obr. 19 je znázorněn diagram sekvencí. Jsou na něm znázorněny objekty, životní linie objektů, zasílání zpráv a ukončení platnosti objektu.

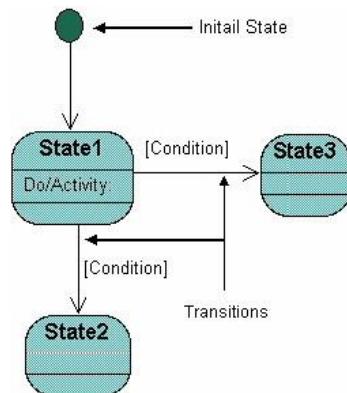


Obr. 19: Diagram sekvencí v notaci jazyka UML

2.2.6 Stavový diagram (State machine diagram)

Na Obr. 20 je znázorněn stavový diagram, jeho počáteční stav, vzájemně propojené přechody s definovanými stavami. V rámci jednotlivých stavů je definováno aktuální status daného místa,

přechody (které mohou být podmíněné nějakým předpokladem) potom značí přechod z jednoho stavu do stavu dalšího.



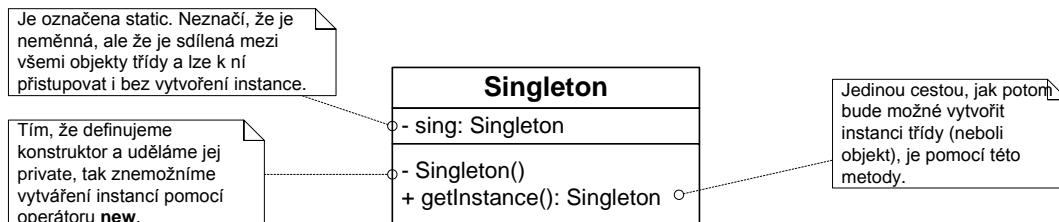
Obr. 20: Stavový diagram

2.3 Návrhové vzory

Návrhové vzory jsou jistá schémata uspořádání tříd, které se postupem doby ustálily a velice často se používají. Těchto uspořádání (tzn. návrhových vzorů) je celá řada [4], [12]. My se zmíníme o několika nejdůležitějších z nich.

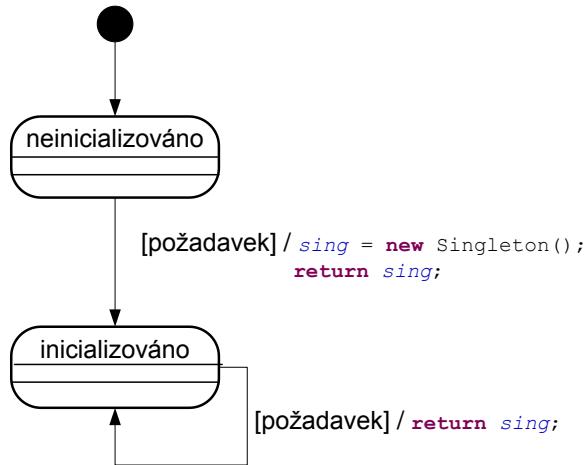
2.3.1 Jedináček (Singleton)

Singleton je velice častým návrhovým vzorem a používá se tehdy, **je-li zapotřebí, aby v celé aplikaci existovala jediná instance dané třídy**. Dejme tomu databázové připojení.



Obr. 21: Diagram tříd - Schéma singleton.

Chování tohoto návrhového vzoru se dá stavovým diagramem popsat následovně:



Jednoduchý příklad:

```

public class Singleton {
    private static Singleton sing;
    // Private constructor suppresses generation of a (public)      //
    default constructor
    private Singleton() {
    }

    public static Singleton getInstance() {
        if (sing == null) {
            sing = new Singleton();
        }
        return sing;
    }
}
  
```

Složitější příklad, který bere v potaz výkonnost:

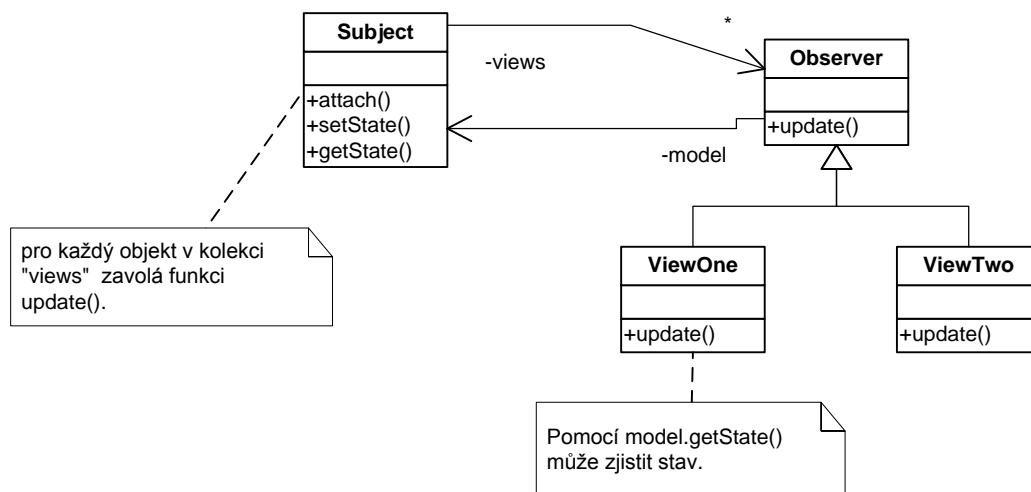
```

public class Singleton {
    private static Singleton sing;
    // Private constructor suppresses generation of a (public)
    // default constructor
    private Singleton() {
    }

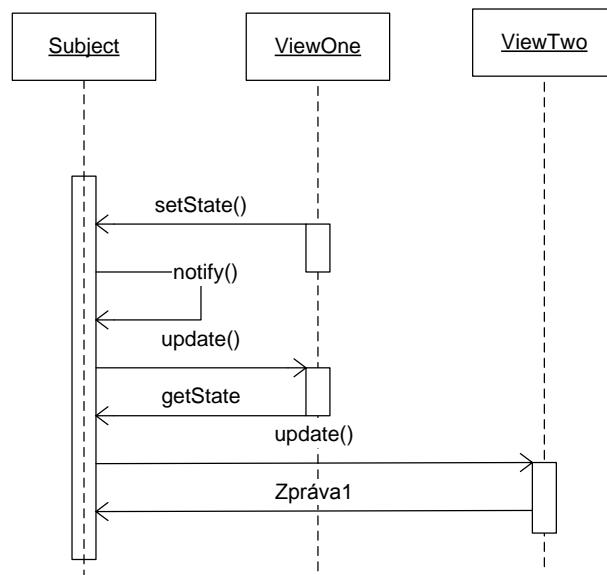
    public static Singleton getInstance() {
        if (sing == null) {
            // Not guaranteed to work
            synchronized (Singleton.class) {
                if (sing == null) {
                    sing = new Singleton();
                }
            }
        }
        return sing;
    }
}
  
```

2.3.2 Observer (pozorovatel)

Mějme následující problém: na stavu jednoho objektu závisí řada jiných objektů. Při změně stavu objektu potřebují být závislé objekty automaticky vyrozuměny. Pro řešení tohoto problému existuje právě návrhový vzor observer (pozorovatel). Na Obr. 22 je znázorněn diagram tříd. Na dalším Obr. 23 je znázorněn příklad fungování v diagramu sekvencí.



Obr. 22: Diagram tříd návrhového vzoru "pozorovatel".



Obr. 23: : Diagram sekvencí návrhového vzoru "pozorovatel".

Jazyk Java v tomto případě poskytuje připravené rozhraní pro implementaci tohoto návrhového vzoru. Třída `Observable` je rozhraní pro implementaci třídy `Subject` a třída `Observer` zajišťuje metody pozorovatele.

Swing¹ používá Observer pro komunikaci mezi komponentami. Nevyužívá ovšem standardní rozhraní poskytované jazykem Java, ale definuje vlastní. Observer je nazýván listener.

```
import java.util.Observable;
import java.util.Observer;

public class Demo {
    public static void main(String[] av) {
        MySubject model;

        // vytvoreni modelu
        ViewOne viewOne = new ViewOne();
        ViewTwo viewTwo = new ViewTwo();

        // pridani modelu k pozorovani
        model = new MySubject();
        model.addObserver(viewOne);
        model.addObserver(viewTwo);

        // z teto pozice oznamim ze byl zmenen stav
        // na zaklade toho se informuji viewOne, viewTwo,
        model.changeSomething();
    }
}

/** Pozorovatel. Kontroluje stav pozorovaneho objektu. */
class ViewOne implements Observer {
    /** For now, we just print the fact that we got notified. */
    public void update(Observable obs, Object x) {
        System.out.println("update(" + obs + ", " + x + ")");
    }
}

/** Pozorovatel. Kontroluje stav pozorovaneho objektu. */
class ViewTwo implements Observer {
    /** For now, we just print the fact that we got notified. */
    public void update(Observable obs, Object x) {
        System.out.println("update(" + obs + ", " + x + ")");
    }
}

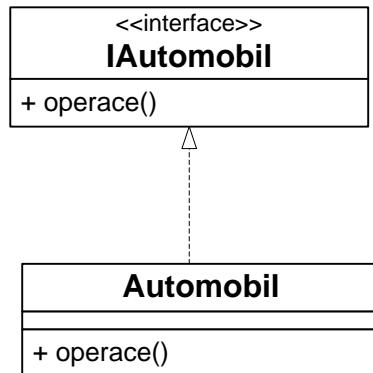
/** Pozorovany objekt. */
class MySubject extends Observable {
    public void changeSomething() {
        setChanged();
        Integer novaHodnota = 1;
        notifyObservers(novaHodnota);
    }
}
```

¹ Swing je knihovna pro tvorbu grafického uživatelského rozhraní na platformě JAVA.

V případě, že bychom se nechtěli opřít o vlastnosti jazyka, dokázali byste říci, co metody `addObserver`, `notifyObservers` dělají a nahradit je vlastním kódem? (POZN: Jedná se o jednoduché řešení v rozmezí několika málo řádků).

2.3.3 Rozhraní (Interface)

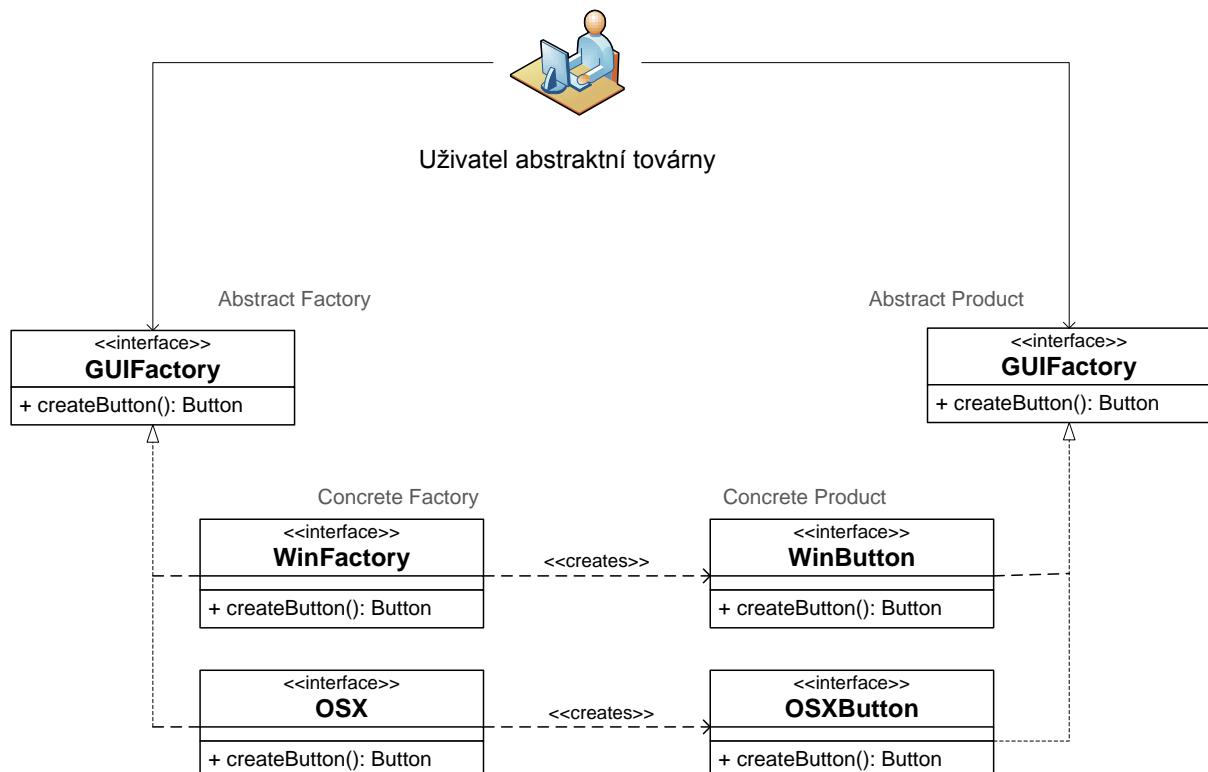
Rozhraní slouží zejména pro potřeby vynutit si jednotnou podobu metod v třídách, které je implementují a současně k definici společného datového typu.



Obr. 24: Příklad – implementace rozhraní.

2.3.4 Abstraktní Továrna (Abstract Factory)

Abstraktní továrna zjednoduší vytváření objektů, které by bylo v složité vytvářet pouze pomocí konstruktoru.



Obr. 25: Návrhový vzor abstraktní továrna.

```
/*
 * GUIFactory example
 */
abstract class GUIFactory
{
    public static GUIFactory getFactory()
    {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0)
        {
            return new WinFactory();
        }
        else
        {
            return new OSXFactory();
        }
    }

    public abstract Button createButton();
}

class WinFactory extends GUIFactory
{
    public Button createButton()
    {
        return new WinButton();
    }
}

class OSXFactory extends GUIFactory
{
    public Button createButton()
    {
        return new OSXButton();
    }
}

abstract class Button
{
    public abstract void paint();
}

class WinButton extends Button
{
    public void paint()
    {
        System.out.println("I'm a WinButton: ");
    }
}

class OSXButton extends Button
{
    public void paint()
    {
        System.out.println("I'm an OSXButton: ");
    }
}

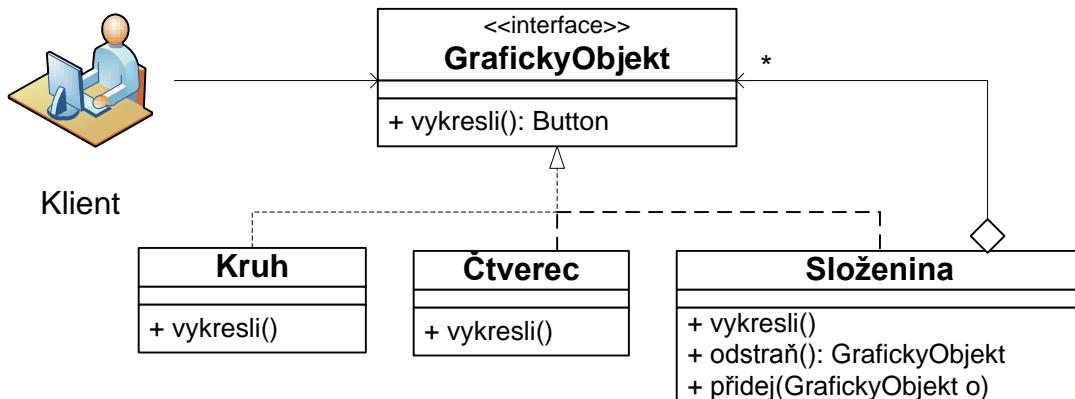
public class Application
{
    public static void main(String[] args)
```

```
{
    GUIFactory factory = GUIFactory.getFactory();
    Button button = factory.createButton();
    button.paint();
}

// Output is either:
//   "I'm a WinButton:"
// or:
//   "I'm an OSXButton:"
}
```

2.3.5 Skladba (Composition)

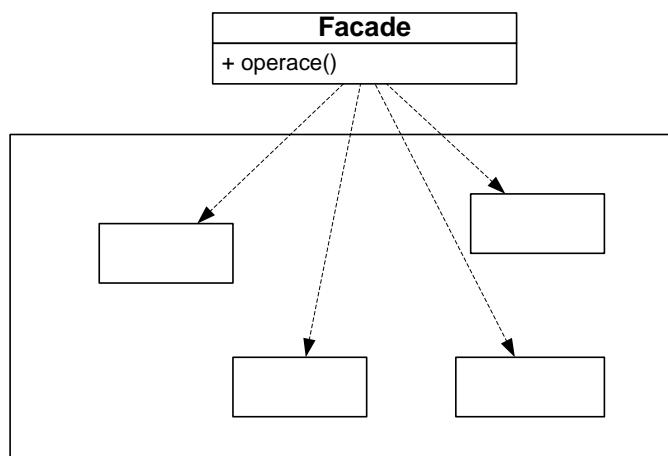
Skladba umožňuje hierarchické zanořování objektů.



Obr. 26: : Diagram tříd pro návrhový vzor "skladba".

2.3.6 Fasáda (Facade)

Fasáda poskytuje jednotné rozhraní k různým třídám, které byly původně navrženy s odlišným rozhraním.



Obr. 27: Schéma pro návrhový vzor "fasáda", který sjednocuje přístup k různým třídám.

2.4 Literatura

- [1] ARLOW J., NEUSTADT I., „UML 2 a unifikovaný proces vývoje aplikací“, Computer Press, a.s., ISBN: 80-251-1503-9
- [2] Schmuller, J.: Myslíme v jazyce UML. Grada Publishing, Praha, 2001.
- [3] Meilir Page-Jones. : Základy objektově orientovaného návrhu v UML. Grada Publishing, Praha, 2001.
- [4] Bruce Eckel, Thinking in Patterns Revision 0.9, May 20, 2003, <http://www.odioworks.com/46-Bruce_Eckel's_Free_Electronic_Books.html>
- [5] Bruce Eckel, Thinking in Java, 2nd edition, <http://www.odioworks.com/46-Bruce_Eckel's_Free_Electronic_Books.html>
- [6] PILONE D., UML 2.0 in a Nutshell, O'Reilly Media, Inc.; 2 edition
- [7] ROSENBERG D., STEPENS M. „Use Case Driven Object Modeling with UML: Theory and Practice“, Apress
- [8] René Stein, „Návrh aplikací v jazyce UML - Unified Modeling Language“, 5. 11. 2003.
- [9] Pavel Tišnovský, „Nástroje pro tvorbu UML diagramů“, ROOT.CZ, <<http://www.root.cz/clanky/nastroje-pro-tvorbu-uml-diagramu/>>
- [10] Martin Fowler, „Programy pro práci s UML diagramy“, <<http://java.vse.cz/Main/UMLNastroje>>
- [11] BURGET R., „Abstraktní datové typy 3“, Přednáška kurzu MTIN
- [12] GOOCH T., „UML Tutorial“, Kennesaw State University <http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/>
- [13] Design Patterns, SourceMaking <http://sourcemaking.com/design_patterns>

3 Hodnocení algoritmů

3.1 Definice algoritmu

Algoritmus můžeme definovat jako jednoznačně určenou posloupnost konečného počtu elementárních kroků vedoucí k řešení daného problému, přičemž musí být splněny základní vlastnosti každého algoritmu:

1. **Hromadnost a univerzálnost** - algoritmus musí vést k řešení celé třídy úloh, vzájemně se lišících pouze vstupními údaji (argumenty programu). Musí řešit úlohu pro libovolnou přípustnou kombinaci vstupních dat a musí pokrývat všechny situace, které mohou při výpočtu nastat.
2. **Determinovanost (jednoznačnost)** - v každém kroku algoritmem popsaného postupu musí být jednoznačně určeno, co je výsledkem tohoto kroku a jak má algoritmus dále pokračovat. Důsledkem této vlastnosti je, že pro stejná vstupní data vydá algoritmus vždy stejný výsledek.
3. **Konečnost** - algoritmus v konečné době skončí (jinak by byl nepoužitelný). (S tím částečně souvisí problém zastavení Turingova stroje [6])
4. **Rezultativnost** - algoritmus při zadání vstupních dat vždy vrátí nějaký výsledek (může se jednat i o chybové hlášení).
5. **Korektnost** - výsledek vydaný algoritmem musí být správný.
6. **Opakovatelnost** - při použití stejných vstupních údajů musí algoritmus dospět vždy k témuž výsledku.

Algoritmus označujeme jako **efektivní**, jehož složitost je maximálně polynomiální (např. n^{127}) nikoliv však exponenciální 2^n .

Efektivita algoritmu může být zajímavým kritériem ve vědních disciplínách, jako je kryptografie, kde naopak usilujeme o nalezení takového problému, kde neexistuje efektivní algoritmus. Příkladem je asymetrické šifrování pomocí RSA (iniciály autorů Rivest, Shamir, Adleman), kde se jedná o **problém rozkladu čísla na prvočísla** (faktorizace) [1]. Dalším problémem, ke kterému není dosud znám (není jisté, zdali vůbec existuje) efektivní algoritmus je **problém diskrétního logaritmu**. Ten se s výhodou používá pro **Diffie-Hellman výměnu klíčů**.

3.2 Hodnocení algoritmů

Složitost algoritmů se klasifikují na základě dvou kritérií: **paměťová náročnost a časová náročnost**. Popis, který se k tomu může použít je: **absolutní složitost a asymptotická složitost**.

Algoritmus a jeho implementace jsou rozdílné věci. Teoreticky může být algoritmus výkonný, ale pokud k jeho implementaci nepoužijeme správné metody a techniky, můžeme celý jeho potenciál ztratit a degradovat jej na nějakou formu méně výkonného algoritmu. S něčím

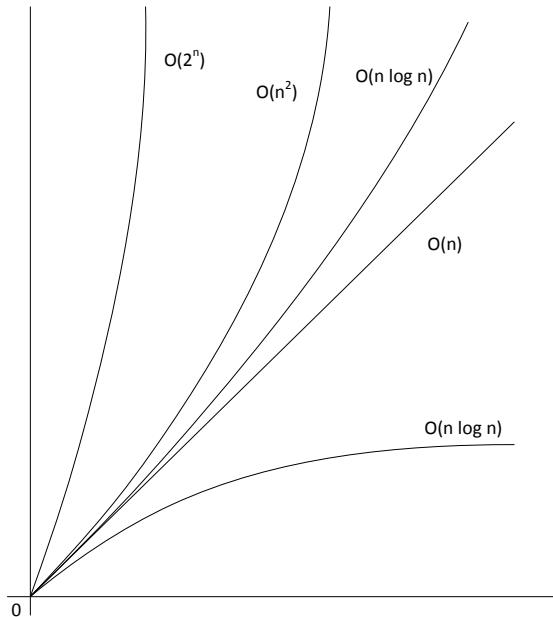
podobným se můžeme často setkat i v praxi. Na internetu existuje mnoho zdrojových kódů a implementací programů a ne vždy se jedná o skutečně kvalitní kód a správnou implementaci. Příklad algoritmu se stejnou funkcí je například množina, kterou lze reprezentovat v jazyku JAVA pomocí tříd HashSet, TreeSet popřípadě i (ne příliš vhodně) pomocí seznamu LinkedList. I když chování je stejné, operace vkládání a vyhledávání mají výrazně jiné vlastnosti.

3.2.1 Základní řády složitosti a jejich kvalifikace

Níže jsou v Tab. 1 uvedeny třídy (řády) složitostí, které jsou nejčastěji používány při popisu algoritmů. Pro všechny platí, že hodnota proměnné n je v intervalu $<0, \infty>$, c reprezentuje konstantní číslo. Na Obr. 28 jsou pak vyobrazeny tvary funkcí a jejich vzájemné srovnání.

Notace	Název	Příklad
$\mathcal{O}(1)$	konstantní	Rozhodnutí, zda-li je číslo sudé či liché
$\mathcal{O}(\log n)$	logaritmická	Nalezení prvku v ADT binárním vyhledávacím stromě [4]
$\mathcal{O}(n)$	lineární	Nalezení prvku v neseřazeném poli
$\mathcal{O}(n \log n)$	lineárně logaritmická, neboli kvazilineární	Serazení seznamu pomocí heapsort, výpočet FFT
$\mathcal{O}(n^2)$	kvadratická	Řazení pomocí insertion sort, výpočet DFT
$\mathcal{O}(n^c)$, $c > 1$	polynomická	Floyd-Warshallův algoritmus
$\mathcal{O}(c^n) = 2^{\mathcal{O}(n)}$	exponenciální	Přesné řešení problému obchodního cestujícího [3].
$\mathcal{O}(n!)$	faktoriální (kombinatorická)	Řešení NP úplného problému [3] hrubou silou (problém obchodního cestujícího).

Tab. 1: Základní složitostní funkce v pořadí od nejvíce výhodné po nejméně výhodnou



Obr. 28: Přehled funkcí základních složitostí

3.2.2 Absolutní složitost

Začněme první skupinou - **absolutní** složitostí. Úvodem je nutno říci, že se z praktických důvodů nepoužívá příliš často, jelikož absolutní složitost je možné určit pouze u jednoduchých případů. Níže je ukázka funkce v jazyce JAVA, která vrací jako výstupní hodnotu mocninu vstupního argumentu funkce:

Příklad algoritmu (JAVA):

```
public long mocnina(int cislo) {
    return cislo*cislo;
}
```

Určit paměťovou složitost je v tomto případě triviální – není zapotřebí žádné další paměti, proto je její absolutní složitost konstantní rovna 2 (jedna pro vstupní argument, jedna pro návratovou hodnotu).

U absolutní složitosti se mnohdy uvádí také její jednotka, kde bývá nejčastěji uváděn **počet instrukcí** v asemblerovém kódu.

Z pohledu časové složitosti se taktéž jedná o jednoduché řešení. Instrukcí, které procesor pro výpočet mocniny musí provést, je vždy konstantní, proto je i složitost tohoto algoritmu konstantní a rovna 3 – zřejmě by se ve strojovém kódu ve finální podobě jednalo o jednu instrukci *MOV* (přesun v registrech), 1 instrukci *MUL* (násobení) a jedna instrukce *RET* (návrat z volání funkce)[2]. Pro různé architektury se počet instrukcí může případně i lišit, nicméně i tak zůstane doba výpočtu vyjádřitelná konstatntním číslem.

```
.globl _mocnina
_mocnina:
    movl 4(%esp), %eax
    mull 8(%esp), %eax      # Nasobeni
```

```
ret          # Return value is in %eax
```

3.2.3 Asymptotická složitost

Ne vždy je ale možno určit přesnou složitost algoritmu. Složitost algoritmu může například záviset na hodnotách vstupních dat. Z toho důvodu byla zavedena takzvaná asymptotická složitost, která approximuje chování funkce a to z pohledu buď nejlepšího možného chování, průměrného chování či nejhoršího chování.

3.2.3.1 Notace O (Omkron, big-O)

$O(f(n))$ – Omikron notace (často se jí říká i "velké-O notace", big-O notace, worst-case notace)

$O(f(n))$ značí horní asymptotický odhad. Tedy, jinými slovy, v žádném případě nemůže nastat případ, kdy by skutečná složitost algoritmu byla větší než hodnota $O(f(n))$.

S tímto druhem notace se setkáte v 98% případů. Přesto i tak nemusí být vždy směrodatná, například řadící algoritmus Quick sort, který je nejčastější implementací na dnešních architekturách počítačů, má hodnotu funkce O kvadratickou složitost. (tj. jednu z nejhorších v kategorii řadících algoritmů), přesto jeho průměrná časová složitost je natolik dobrá, že ve většině případů vykazuje nejlepší výsledky.

3.2.3.2 Notace Θ (Theta)

$\Theta(f(n))$, neboli theta notace je průměrný odhad chování funkce. Algoritmus se může ve skutečnosti chovat lépe i hůře, ale v průměru by se měl co nejvíce blížit hodnotě této funkce. Tato notace je mnohem méně často používaná v porovnání s notací O .

3.2.3.3 Notace Ω (Omega)

$\Omega(f(n))$, neboli Omega notace je dolní odhad aneb „lepší už to nebude“.

Tato notace je mnohem méně často používaná v porovnání s notací O .

3.2.3.4 Další notace

Existují ještě další rozdelení [5], která se však v praxi téměř nepoužívají. V rámci kurzu se jimi nebudeme zabývat.

3.3 Příklady

3.3.1 Příklad 1

Následující algoritmus provádí výpočet mocniny čísla 2.0 / e. Pokuste se určit jeho vlastnosti z hlediska absolutních asymptotických složitostí z pohledu času i prostoru (tj. paměti).

```
// N.... je vstupní parametr algoritmu a určuje počet prvků pole;
int sum = 1;
for (i=0; i< N; i++) {
    // výpočet
    sum = sum * 2.0 / e;
}
```

3.3.2 Příklad 2

Výpočet sumy prvků v matici:

```
// N.... počet prvků;
final int N = ???;
int matrix[][] = new int[N][N];
int sum = 0;
for (i=0; i< N; i++) {
    for (j=0; j< N; j++) {
        // nějaký výpočet
        sum += matrix[i][j];
    }
}
```

3.3.3 Příklad 3

```
final int N = ???;
int matrix[][] = new int[N][1000];
int sum = 0;
for (i=0; i< N; i++) {
    for (j=0; j< 1000; j++) {
        // nějaký výpočet
        sum += matrix[i][j];
    }
}
```

I přestože je v kódu dvakrát vnořený cyklus **for**, nejedná se o složitost kvadratickou ale pouze lineární. Druhý cyklus je konstanta (1000). Výsledná složitost je tedy $\mathcal{O}(1000n)$, $\Theta(1000n)$, $\Omega(1000n)$. Zpravidla je dostačující získat pouze řád složitosti, z toho důvodu lze notaci zjednodušit na $\mathcal{O}(n)$, $\Theta(n)$, $\Omega(n)$.

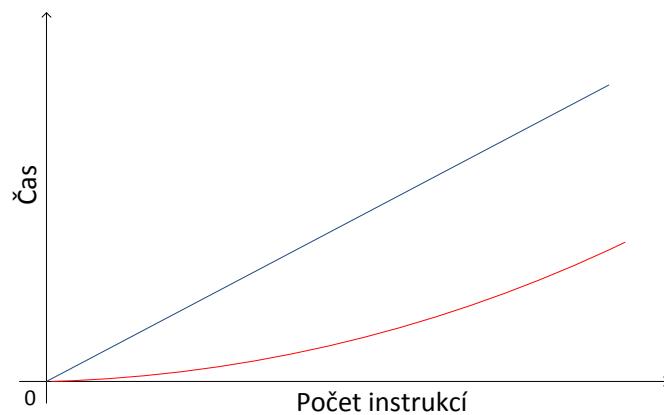
3.3.4 Příklad 4

Předpokládejme, že máme 2 algoritmy, které operují nad N prvky. První algoritmus má časovou závislost na počtu prvků N kvadratickou a zpracování jediného prvku trvá 10 taktů procesoru. Druhý algoritmus má časovou závislost na počtu prvků lineární a zpracování jediného prvku trvá 10000 taktů procesoru. Pokuste se oba dva algoritmy analyzovat a pokuste se rozhodnout, zda-li je možné jednoznačně určit, který z algoritmů je lepší.

Frekvence procesoru: $f = 3,4\text{GHz}$, perioda provedení jedné instrukce je potom: $T = 1/3,4\text{GHz} = 2.9412e-10$ sec pro jeden takt procesoru (pro jednoduchost zanedbejme superskalární procesor a předpokládejme skalární procesor).

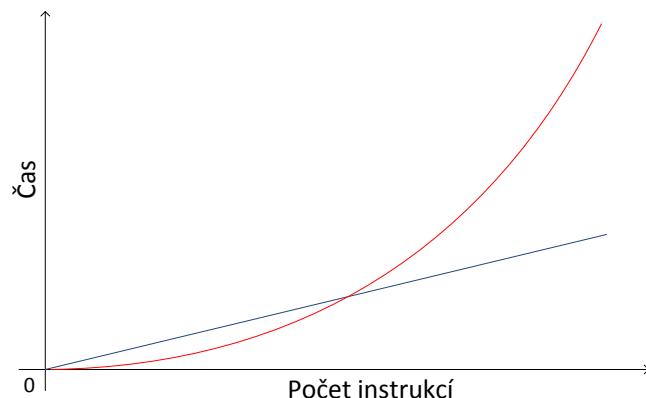
3.3.4.1 $N = 1..60$ prvků

- Algoritmus s lineární časovou složitostí: 17 usec
- Algoritmus s kvadratickou časovou složitostí: 10 usec



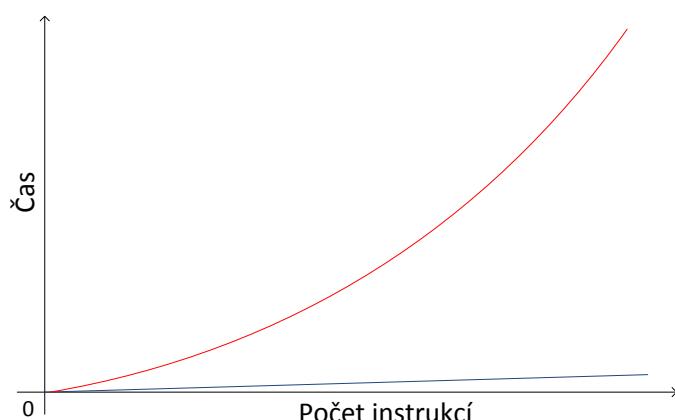
3.3.4.2 $N = 1..200$ prvků

- Algoritmus s lineární časovou složitostí: 59 usec
- Algoritmus s kvadratickou časovou ložitostí: 116 usec



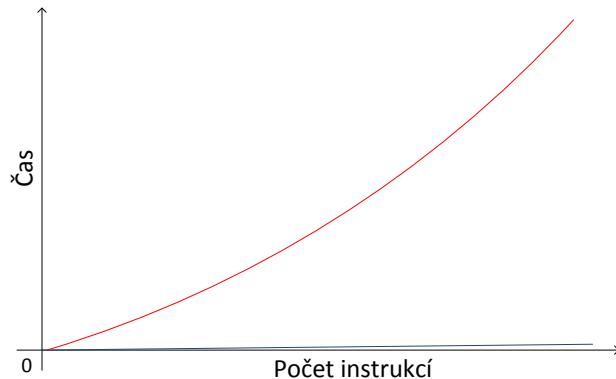
3.3.4.3 $N = 1..10'000$ prvků

- Algoritmus s lineární časovou složitostí: 2,882 msec
- Algoritmus s kvadratickou časovou ložitostí: 282,471 msec



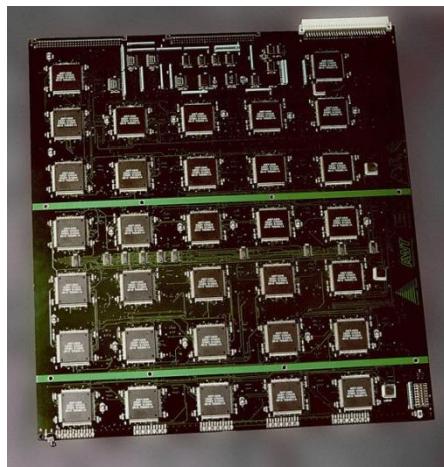
3.3.4.4 $N = 1..10'000'000$ prvků

- Algoritmus s lineární časovou složitostí: 2.935294 sec
- Algoritmus s kvadratickou časovou ložitostí: 292942.411647 sec = 3 dny, 9.3 hodin



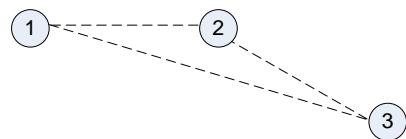
3.3.4.5 Závěr příkladu

Jak je z předešlého příkladu patrné, nevhodnou volbou algoritmů a chybně zvoleným přístupem k problému je možné se snadno dostat za hranici spočítatelnosti. Pro analytika je proto nezbytná dobrá znalost stěžejních algoritmů. Na základě nich je možné optimalizovat, případně i vytvářet nové algoritmy a výrazně tak ušetřit cenu výsledného produktu.



Obr. 29: Elektronický obvod, navržený v roce 1998, pro útok hrubou silou na šifru DES. Konstrukce obvodu přišla na méně než \$250 000 a chybná volba algoritmů by byla naprosto fatální pro výpočetní výkon (zdroj: www.eff.org).

3.3.5 Příklad 5



Obr. 30: Uzly grafu v euklidovském prostoru.

Mějme matici vzdáleností (je tedy symetrická):

	1	2	3
1	0	3	5.5
2	3	0	3
3	5.5	3	0

Úkol:

- Vytvořte algoritmus, který seče vzdálenosti mezi uzly pro obecný graf (tj. s proměnným počtem vrcholů N, nikoli konstantním).
- Určete složitost takového algoritmu.

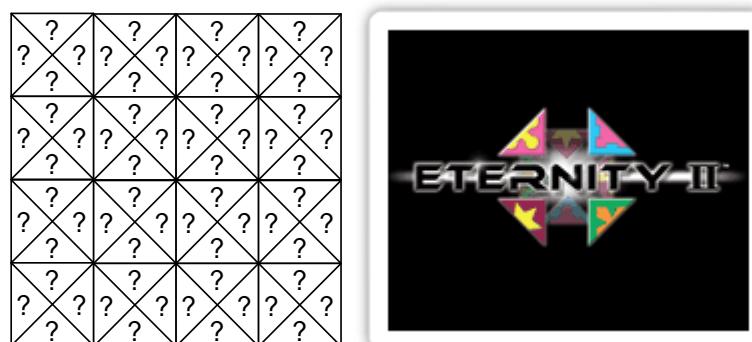
(Odpověď na konci dokumentu)

Demo: Lokalizace uzlů v síti – Global Network Positioning

3.3.6 Příklad 6

Před nedávnem, přesněji tedy dne 28.7.2007, byla na celosvětový trh uvedena hra Eternity II [6], což je puzzle s výhrou \$2'000'000. Je to následníkem předchozí varianty hry Eternity, která již byla rozluštěna britským studentem a doba řešení trvala údajně dobu 18 měsíců. Cílem této hry je uspořádat do 2D pole o jistých rozměrech jednotlivá políčka tak, aby symboly i barvy v každém dílku navazovaly na sousedící dílky a současně šedé dílky rámovaly okraje. Každé políčko lze samozřejmě otáčet a mohou mít tedy každé 4 různé polohy. Příklad nevyřešeného hracího pole 4x4 je uveden na Obr. 31. Tato hra se prodává v několika složitostech a to o počtu 32 polí, 72 polí a nejtěžší s 256, za jejíž vyřešení je přislíbena finanční odměna.

POZN: Bohužel není k dispozici přesné zadání puzzle a podobu jejich políček, nicméně předpokládejme, že existuje pouze jediná kombinace, která je správným řešením.



Obr. 31: Hra Eternity II o velikosti pole 4x4 a 16 dílky.

- Dokážete posoudit složitost algoritmu z hlediska proměnných AxB, kde A, B je počet sloupců a řádků v 2D poli AxB.
- Dokážete posoudit složitost algoritmu z hlediska proměnné N, kde N je počet sloupců a řádků v 2D poli NxN.
- Dokážete navrhnout algoritmus pro řešení problému?
- Jak dlouhou dobu zabere řešení pole 16x16, budeme-li řešit problém hrubou silou na některém běžně dostupném HW?

- Do jaké třídy složitosti tento problém lze zařadit?
- Napadl vás nějaký způsob optimalizace?
- Už jste přišli na řešení :-) ?

Spočtěme si nyní, kolik možností rozložení tato hra v sobě poskytuje. Máme k dispozici 4 rohové délky, 56 zbylých okrajových a zbylých 196 pro vnitřní okraje. Kolik je tedy celkem možností? Můžeme to vyjádřit jako:

$$C = \frac{4!}{4} \cdot 56! \cdot 196! \cdot 4^{196} = 2.186 \cdot 10^{559} \quad (1)$$

To je bezesporu velké číslo. Co si ale pod ním představit? Představme si, že průměrná lidská hlava obsahuje přibližně $4.56 \cdot 10^{26}$ atomů. Představme si nyní, že každý atom bude reprezentovat jedno řešení. Lidská hlava je tedy rozhodně málo, přidejme tedy celou naši planetu Zemi, dostáváme se na $8.87 \cdot 10^{49}$. To je sice lepší, ale stále to nestačí, vezměme tedy raději Slunce, které je svým průměrem 100x větší – dostáváme se na $9 \cdot 10^{56}$ atomů. Raději tedy vezměme naši galaxii, kde je 100 až 120 miliard hvězd. Dostaneme se přibližně na hodnotu $1.2 \cdot 10^{68}$, což je bohužel pořád málo. Vezměme tedy celý vesmír a to jak včetně zářící hmoty, tak včetně veškeré tmavé hmoty – dostáváme se v maximálním případě na hodnotu 10^{81} , což je stále málo. Vzít dva vesmíry nic neřeší, tím bychom se dostali na počet atomů $2 \cdot 10^{81}$. No, a pokud bychom předpokládali, že každý atom v našem vesmíru bude představovat další vesmír, dostaneme se na hodnotu 10^{162} , což také nestačí.

Předpokládejme nyní, že každý z obyvatelů naší planety včetně novorozeňat (6.6 miliard) má počítač, který dokáže za jednu sekundu spočítat 4 miliardy možností (), tedy za jeden rok $1.26 \cdot 10^{17}$. Všichni dohromady tedy spočteme za jeden rok $8.325 \cdot 10^{26}$ možností a za dobu trvání vesmíru $1.665 \cdot 10^{47}$. Pravděpodobnost, že bychom tedy prohledáváním všech možností za tuto dobu našli řešení, se s čistým svědomím může považovat za 0%. Tedy v případě že si vystačíme s přesností 10^{-512} ☺.

Z tohoto pohledu by to možná vypadalo značně skepticky. Naštěstí teoretická informatika poskytuje několik cest, jak výpočetně náročné problémy řešit. Nebo alespoň výrazně zvýšit pravděpodobnost nalezení řešení.

3.4 Literatura

- [1] ARJEN K. LENSTRA, „Integer Factoring“. Kluwer Academic Publishers, Boston.
- [2] DJ Delorie, „Function Calling Conventions“,
Dostupné z <http://www.delorie.com/djgpp/doc/ug/asm/calling.html>
- [3] Held, Michal & Richard M. Karp (1962), "A Dynamic Programming Approach to Sequencing Problems", Journal of the Society for Industrial and Applied Mathematics 10 (1): 196-210
- [4] BURGET, R. Abstraktní datové typy, Přednáška MTIN.

- [5] William B. Frakes, Information Retrieval: Data Structures and Algorithms, Prentice Hall; 1 edition (June 22, 1992)
- [6] Eternity II, <http://www.eternityii.com/>

3.4.1.1 Odpovědi k některým příkladům

Příklad 4.3

```
public class MaticeVzdalenosti {
    public static void main(String[] args) {
        // konstanty se znaci final a pisou zpravidla velkymi písmeny
        final int MATRIX_SIZE = 3;
        double matrix[][] = new double[MATRIX_SIZE][MATRIX_SIZE];
        // naplnení hodnotami ...

        double sum = 0.0;
        for(int i=0; i<MATRIX_SIZE; i++) {
            for(int j=0; j<i; j++) {
                sum += matrix[i][j];
                System.out.println("I:" + i + " J:" + j);
            }
        }
    }
}
```

Složitost tohoto algoritmu je vyšší než lineární, nicméně nižší než kvadratická. Přesné vyjádření složitosti by vypadalo asi jako kombinace bez opakování :

$$\binom{N}{2} = \frac{N^2 - N}{2} \quad (2)$$

Jelikož se ale zabýváme asymptotickými složitostmi a pokud nespecifikujeme jinak, téměř vždy mluvíme o nejhorší možné složitosti. Odpověď tedy může znít: $O(N^2)$.

4 Úvod do abstraktních datových typů

4.1 Úvod

Každý problém, který má být řešen či realizován za pomocí počítače musí být nějakým způsobem reprezentován v paměti. Existuje mnoho přístupů a v rámci kapitol o abstraktních datových typech budou shrnuty základní možnosti a základní datové struktury, které se k tomu účelu nejčastěji používají.

Všechny z probíraných datových struktur mají své kladné stránky a současně i záporné stránky. Úkolem analytika je, aby zvolil vhodnou datovou reprezentaci. Jejich nevhodná volba může výrazně zpomalit chování celého systému, ale na druhou stranu může dojít i k případům, kdy problém trvající stovky let lze optimalizovat na 4 sec [1]. Popřípadě z pohledu paměťové složitosti, algoritmus s náročností gigabytů lze optimalizovat na náročnost v rádu kilobytů.

ADT přináší vyšší míru abstrakce. To dává možnost využití základních stavebních bloků a tím i vyšší rychlosť, přehlednost a v neposlední řadě i vyšší spolehlivost výsledné aplikace.

4.2 Abstraktní datový typ

(DEFINICE) Abstraktní datový typ je množina dat a množina k nim asociovaných operací, které jsou přesně určeny a nezávislé na jakékoli konkrétní operaci. Často se název zkracuje na **ADT**.

Příkladem jednoduchého abstraktního datového typu může být typ **Boolean** (viz Tab. 2).

Typ	Boolean
Obor hodnot	$\beta = \{\text{True}, \text{False}\}$
Operace	konjunkce: $\beta \times \beta \rightarrow \beta$ disjunkce: $\beta \times \beta \rightarrow \beta$ negace: $\beta \times \beta \rightarrow \beta$
Předpoklady	= rovnost: $\beta \times \beta \rightarrow \beta$ \neq nerovnost: $\beta \times \beta \rightarrow \beta$

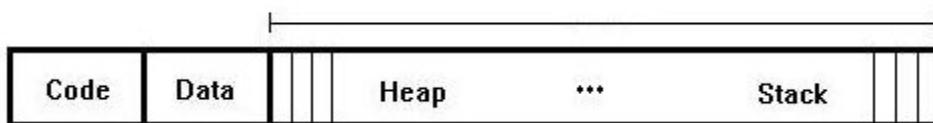
Tab. 2: Abstraktní datový typ Boolean

Relace $\beta \times \beta \rightarrow \beta$ představuje kartézský součin množin β , který je zobrazením do oboru hodnot množiny β . Obdobně je to i s reálnými čísly, celými čísly, atd. Tyto elementární ADT jsou součástí každého programovacího jazyka. Následující odstavce se budou věnovat vyšším abstraktním datovým typům, jako je pole, lineární seznamy, fronta, zásobník, stromy, binární stromy, binární vyhledávací stromy, AVL stromy, tabulky.

4.3 Reprezentace programu v paměti

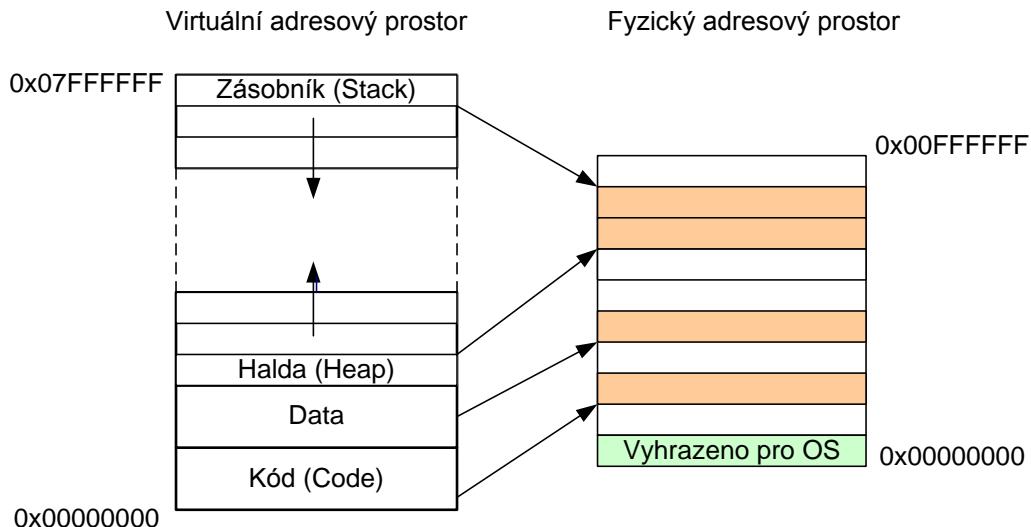
K tomu, abychom mohli vytvářet ADT a realizovat je v paměti počítače, je nutné dobře porozumět, jak jsou programy reprezentovány v paměti počítače. Každá paměť přiřazená procesu v počítači se skládá ze nejméně 4 logických částí – segment instrukcí prováděného programu (Code), datový segment (Data), segment haldy (Heap) a segment zásobníku (Stack) (viz Obr. 32).

Instrukce prováděného programu jsou zřejmě – podle něj počítač postupuje při výpočtu. Je to množina strojových instrukcí, které naprosto jednoznačně provádí program. V sekci „Data“ mohou být uložena data, která jsou známy v době překladu programu, to znamená textové řetězce, hodnoty polí, hodnoty konstant, proměnných atp. Tyto dvě sekce jsou známy v době překladu programu a jejich velikost se během provádění programu nikterak nemění. Zbylé dvě sekce „Heap“ a „Stack“ jsou naopak dynamické a jejich velikost roste/zmenšuje u každé z jiného směru během provádění programu. Sekce „Heap“ neboli halda slouží k alokaci dynamické paměti. Stack, neboli zásobník, je nezbytný ke správnému fungování při volání funkce. Při zavolání programu se změní pozice registru PC (program counter) na místo, kde jsou k dispozici instrukce funkce. Proto, aby po dokončení funkce bylo možné vrátit se zpět na původní místo provádění programu, je nutné pamatovat si adresu PC registru. Ten je ukládán do zásobníku. Spolu s tím jsou sem také ukládány hodnoty argumentů této metody (zálohované hodnoty registrů).



Obr. 32: Rozčlenění paměti programu.

Mohou nastat dva případy problému nedostatku paměti – v případě alokace nové paměti, popřípadě přetečení zásobníku v případě velkého zanoření volání funkcí programu. Na Obr. 33 je vyobrazen vztah fyzické a virtuální paměti, která je rozdělena za pomocí stránkování.



Obr. 33: Vztah fyzické a virtuální paměti.

Pokud chceme vytvořit nový paměťový prostor, je nezbytné, abychom tak učinili prostřednictvím operátoru **new**.

V jazyce JAVA je obrovskou výhodou, že není nezbytné, abychom mazali vytvořené objekty. Pro správu paměti tu existuje tzv. **Garbage Collector**. Abychom smazali alokovanou paměť, stačí, abychom hodnotu proměnné nastavili na hodnotu null:

```
// alokace paměťového prostoru
byte[] nazevPromennePole = new byte[100];
// pokud nastavíme referenci „nazevPromennePole“ na null,
// garbage collector automaticky pozná, že již neexistuje
// žádná reference na místo v paměti, a paměť automaticky uvolní
nazevPromennePole = null;
```

Ohledně Garbage Collectoru toho bylo řečeno mnoho a dočtete se o něm spoustu pravd, ale i nepravd. Je pravdou, že na určité množině je výkonnost provádění programu výrazně pomalejší v porovnání s jazyky komplikovanými do strojového kódu. Co ovšem spousta autorů opomíjí, že jazyk JAVA může díky vyšší abstrakci dělat důmyslnější optimalizace kódu. Na jisté množině aplikací vykazuje jazyk JAVA i lepší výsledky než jazyk C++ [2], [3].

4.4 Objekty vs. Elementární datové typy

Jazyk JAVA není zcela objektově orientovaný jazyk a obsahuje tzv. elementární datové typy. Jsou to např. byte, short, int, long, float, double, char, boolean.

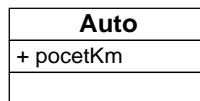
Chování těchto tříd je potom následující:

```
int neinicializovana; // neinicializovana=0
int x = 1;
int y = x;           // x=1, y=1
y = y+2;            // x=1, y=3
```

Všimněte si, že jsme přiřadili proměnnou y hodnotě x, následně jsme změnili hodnotu y a x zůstala nezměněna. V tomto případě se nejedná o reference na objekty, ale jsou to primitivní

datové typy. Také si všimněte, že přestože jsme neinicializovali jeho hodnotu, jazyk JAVA nastaví hodnotu na 0 a nikoli na hodnotu, která zůstala v paměti z dřívějška – tj. téměř nějaké náhodné číslo (na rozdíl například od C/C++).

Jiná situace nastane v případě použití objektů namísto jednoduchých datových typů:



Obr. 34: Třída auto (UML diagram tříd)

Takový kód je reprezentovaný následujícím kódem:

```

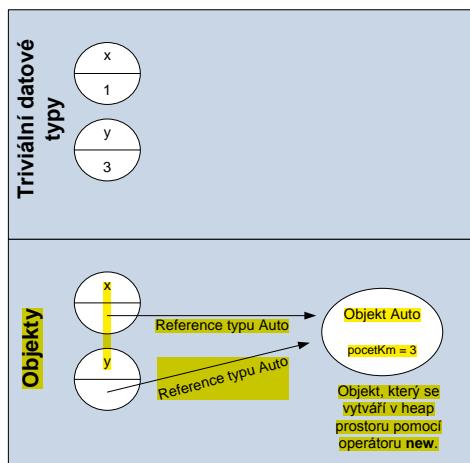
public class Auto {
    public int pocetKm; // implicitní hodnota je 0
}
  
```

A pokusíme se vytvořit příklad obdobný tomu předchozímu, tentokrát ale nikoli s využitím triviálních datových typů ale objektů a referencí na ně:

```

Auto neinicializovany;           // neinicializovany=null
Auto x = new Auto();
Auto y = x;                      // x.pocetKm=1, y.pocetKm=1
y.pocetKm = y.pocetKm + 2;       // x.pocetKm=3, y.pocetKm=3
  
```

Aby bylo zřejmé proč tomu tak je, je nutné uvědomit si, jak jsou tato data reprezentována v paměti a jaký je rozdíl mezi triviálními datovými typy a objekty, resp. referencemi na objekty. V případě objektů se jedná o **reference na objekty**. Zatímco v případě triviálních datových typů se jedná přímo o místo v paměti a nikoli o referenci (viz Obr. 35).



Obr. 35: Rozdíl reprezentace v paměti triviálních datových typů a referencí na objekty.

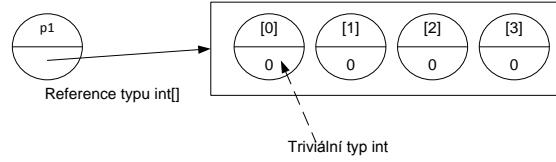
4.5 Pole

ADT pole je jedním z nejběžněji používaných abstraktních datových typů. Je součástí všech programovacích jazyků a jeho realizace je velice jednoduchá.

Nevýhodou pole je, že jeho **délka je po vytvoření neměnná**. V případě neznalosti dalších datových typů (pro tento účel se hodí ADT seznam) **by docházelo ke zbytečnému plýtvání**

paměti anebo, v opačném případě, k vyčerpání jeho kapacity. Níže je příklad vytvoření pole o 4 prvcích typu **int**:

```
// pole o velikosti 4 prvků
int[] p1 = new int[4];
```



Obr. 36: Model reprezentace pole v paměti.

Pokud budeme chtít změnit některou položku pole, využijeme k tomu referenci na pole (pole je objekt) a zadáme jeho **index**:

```
p1[2] = 4;
```

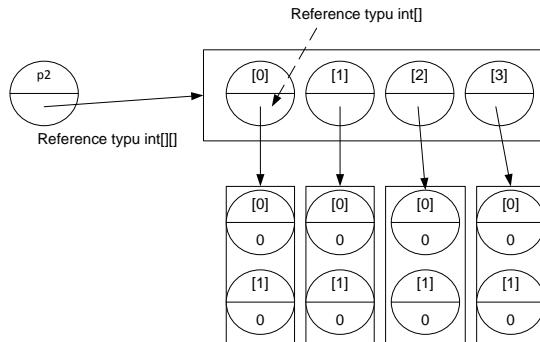
V případě čtení jednoduše zpřístupníme potřebnou položku:

```
p1[2];
```

4.5.1 Vícerozměrná pole

Vícerozměrné pole je možné vytvořit jako "pole polí".

```
// dvourozměrné pole o velikosti 4x2 prvků
int[][] p2 = new int[4][2];
```



Obr. 37: Reprezentace v paměti dvojrozměrného pole.

Zápis do kterékoli položky pole bude analogický:

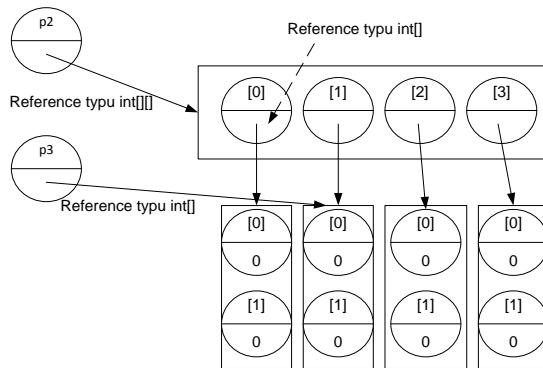
```
p1[2][1] = 4;
```

Anebo čtení:

```
p1[2][1];
```

Pokud chci zpřístupnit pole ve druhém sloupci, stačí k tomu něco jako:

```
int[] p3 = p2[1];
```



Obr. 38: Zpřístupnění podčásti pole pomocí reference typu int[].

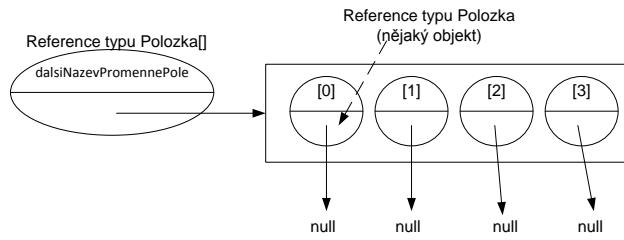
Pozn: Třídy **Integer**, **Double**, **Byte**, **Boolean**, **Float**, atd. (zde si naopak všimněte velkého počátečního písmene) jsou sice objekty, ale tvoří výjimku v chování objektů. Správa paměti je organizována tak, že se navenek tváří jako triviální datové typy (int, double, byte, float, boolean, ...)

4.5.2 Pole objektů

V případě pole objektů je situace mírně odlišná než u triviálních typů. Příčina plyne z rozdílu reprezentace v paměti (viz Obr. 35). Pokud vytvoříme pole objektů, tak pole obsahuje pouze reference na objekty – a ty jsou implicitně nastaveny na hodnotu **null**. Kromě vytvoření pole, tak proto musíme vytvořit i jednotlivé objekty. Příčinou této odlišnosti je skutečnost, že jazyk JAVA není plně objektově orientovaný jazyk. Někteří tuto skutečnost tvůrcům jazyka vyčítají, ale důvod k tomu je ryze pragmatický - datové typy jako **int**, **double**, **byte**, **boolean** (všimněte si počátečního malého písmene) jsou velice často používané a nemají žádné zvláštní vlastnosti kromě vložení hodnoty a jeho přečtení. Kromě toho existují i typy **Integer**, **Double**, **Byte**, **Boolean**, což jsou plnohodnotné objekty a disponují i některými metodami – jako příklad „parsing“ (rozkódování) z textu a tedy proměnné tohoto typu mohou nabývat hodnot **null**.

POZN: U těchto typů existuje výjimka – navenek se chovají stejně jako triviální typy a to přestože jsou skutečně v paměti organizovány jako objekty.

```
public class Polozka {
    public static void main(String arg) {
        // pole o velikosti 4 prvků s objekty
        // - kazda polozka je typu null !!!
        Polozka[] dalsiNazevPromennePole = new Polozka[4];
        //naplneni daty
        for(int i = 0; i< dalsiNazevPromennePole.length; i++) {
            // vytvoreni noveho objektu a nastaveni
            // reference v poli na jeho hodnotu
            dalsiNazevPromennePole[i] = new Polozka();
        }
    }
}
```



Obr. 39: Stav pole objektů před naplněním za pomocí cyklu „for“. Každá reference ukazuje na speciální hodnotu „null“, což představuje neinicializovanou hodnotu. Cyklus „for“ poté vytvoří pro každou referenci nový objekt a nastaví hodnotu reference na něj.

4.6 Lineární seznam

Problém se **statickou velikostí pole** řeší ADT seznam. ADT seznam umožňuje, aby počet jeho **prvků rostl či klesal do libovolné délky**, která je aktuálně **nutná**. To přináší **efektivní využití z hlediska nároků na paměť**, nicméně je **náročnější na implementaci a operace nad ním mají horší časovou složitost**.

Ačkoli název „lineární seznam“ může znít odborně, jedná se o jednoduchou myšlenku a netřeba na tom hledat žádné složitosti či tajuplnosti. Jedná se zkrátka o způsob, jak vytvořit **pole prvků proměnné délky** a organizovat tyto prvky v řadě za sebou (tj. lineárně).

4.6.1 Jednosměrně vázaný lineární seznam

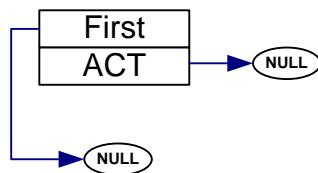
Definice lineárního seznamu je uveden v Tab. 3.

Typ	Jednosměrně vázaný seznam
Obor hodnot	Množina všech lineárně seřazených posloupností všech délek.
Operace	isEmpty addFirst removeFirst getFirst setActFirst setActNext removeAfterAct addAfterAct getAct getSize

Tab. 3: Jednosměrně vázaný lineární seznam.

Jednotlivé operace dělají to, co naznačují jejich anglické názvy a podrobněji budou vysvětleny v následujícím textu.

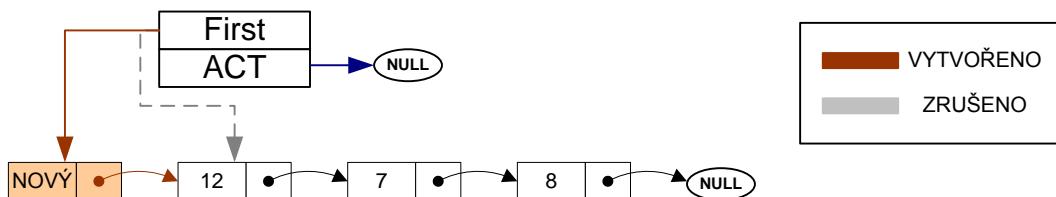
Na Obr. 40 je ukázka nejjednoduššího seznamu. Seznam je prázdný (neobsahuje žádný prvek) a připraven na veškeré zmíněné operace.



Obr. 40: Prázdný jednosměrně vázaný seznam. Neobsahuje žádnou datovou položku.

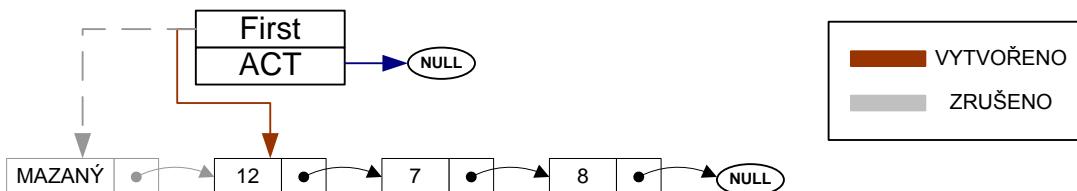
isEmpty: vrací pravdu v případě, že seznam, nad kterým je tato operace volána neobsahuje žádný prvek. Existuje mnoho cest jak zjistit, že je seznam prázdný, jednou z nich a nejjednodušší je zjištění, zdali „First“ odkazuje na null či nikoli.

addFirst: vloží nový prvek na první pozici v seznamu.



Obr. 41: Do již tříprvkového jednosměrně vázaného seznamu je metodou addFirst vložen nový prvek na první pozici seznamu.

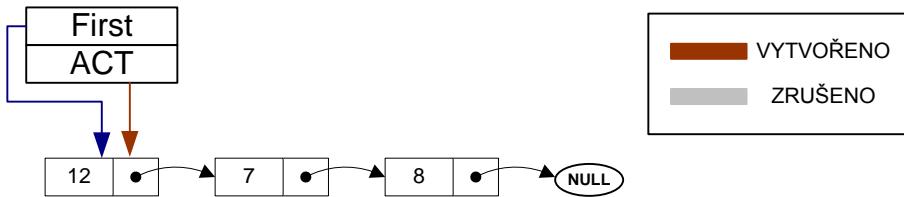
removeFirst: odstraní první prvek ze seznamu. Pokud je seznam prázdný, neprovede žádnou operaci (viz).



Obr. 42: Ze čtyřprvkového jednosměrně vázaného seznamu je metodou removeFirst smazán prvek z první pozice.

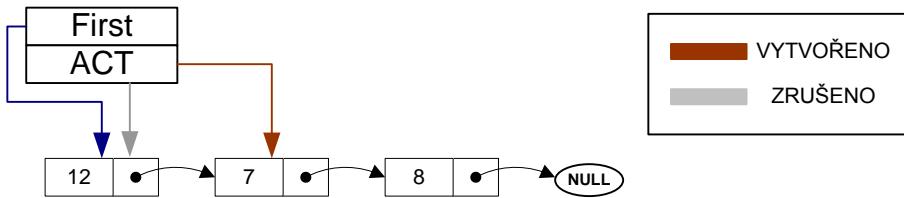
getFirst: funkce vrací referenci na první položku v seznamu. Je-li seznam prázdný, vrací null. Nežli se dostaneme k metodám, které operují s položkou ACT by mělo být nejprve řečeno, k čemu vlastně slouží. Jedná se o jakýsi iterátor, tedy pomocnou proměnnou, kterou se dá přistupovat i k datům uvnitř seznamu. Tedy nejen na první pozici, jak bylo zmíněno v předešlých metodách, ale kdekoli. Pokud tuto položku umístíme za třetí prvek seznamu, můžeme například daleko snadněji smazat všechny prvky za touto pozicí. Hodnota ACT je implicitně nastavena na null. Proměnná ACT musí vždy odkazovat na prvek, který je přítomen v seznamu. Pokud by byla některá položka odstraněna a ACT na ni zůstala odkazovat, jednalo by se o nekonzistence seznamu a hrubou chybu programátora.

setActFirst: nastaví hodnotu proměnné ACT na první prvek v seznamu. Pokud je seznam prázdný, nastaví hodnotu na null (viz Obr. 43).



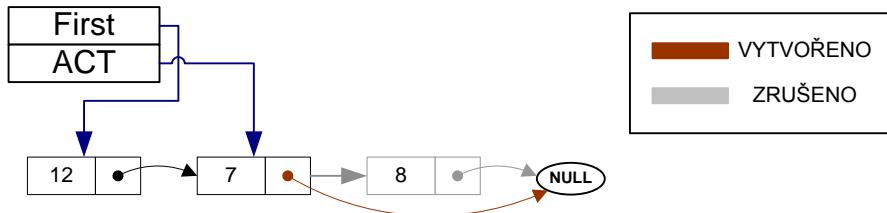
Obr. 43: Reference ACT byla nastavena na první pozici v seznamu.

setActNext: posune proměnnou ACT na následující položku v seznamu. Pokud je hodnota ACT nespecifikována (rovna null), neprovede se žádná operace.



Obr. 44: Reference ACT byla nastavena na následující pozici v seznamu.

removeAfterAct: Metoda odstraní prvek, který následuje za prvkem, na který ukazuje reference ACT v seznamu. Pokud ACT je nespecifikováno (ACT == null), neprovede se žádná operace.



Obr. 45: Reference ACT odkazovala na druhou položku v seznamu a byla zavolána metoda removeAfterAct.

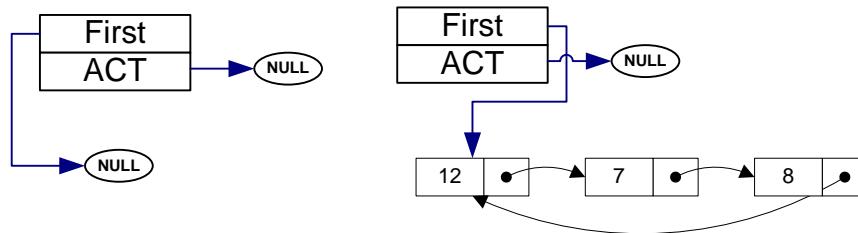
addAfterAct: Opakem předchozí operace – tedy vloží prvek za aktuální pozici. Pokud ACT je nespecifikováno (ACT == null), neprovede se žádná operace.

getSize: Vrací velikost seznamu, tedy počet prvků, které obsahuje.

getAct: vrací položku ACT.

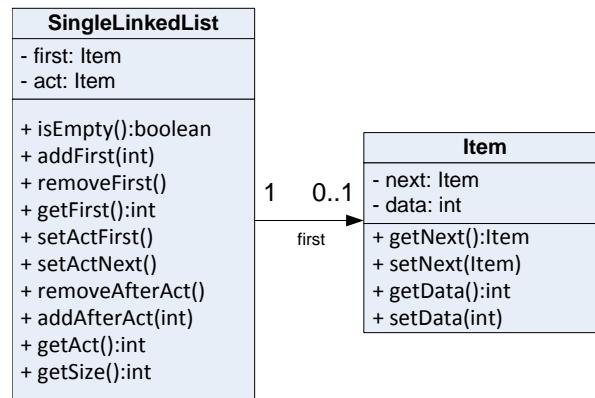
4.6.2 Cyklický jednosměrně vázaný seznam

Cyklický jednosměrně vázaný seznam je ve všech ohledech identický s klasickým jednosměrně vázaným seznamem s jedinou výjimkou a to, že **poslední položka seznamu odkazuje na první prvek**. V necyklickém seznamu tato reference ukazuje na null. V případě cyklického seznamu s jedním prvkem odkazuje položka sama na sebe.



Obr. 46: Na obrázku jsou vyobrazeny dva cyklicky vázané seznamy. Jeden je prázdný, druhý obsahuje 3 prvky.

4.6.3 Reprezentace programem



Obr. 47: Návrh class diagramu pro jednosměrně vázaný seznam v notaci jazyka UML[4].

```

public class Item {
    private Item prev;
    private int data; // může být libovolný datový typ
}

public class SingleLinkedList {
    private Item first;
    public void addFirst(Item newItem) {
        // kod pro vložení prvku
    }
    // .... ostatní metody obdobně
}
  
```

POZN.: Přestože jsme si definovali, že v případě kompozice by měla být třída *Item* vnořena v třídě *SingleLinkedList*, je možné je nechat i odděleně. Jediné co je nutné zaručit, aby si položky *Item* třída *SingleLinkedList* spravovala sama. Příklad bude demonstrován v rámci cvičení.

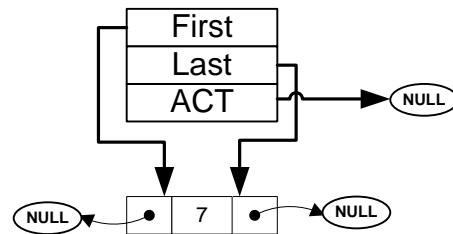
4.7 Obousměrně vázaný lineární seznam

Další variantou lineárního seznamu je **obousměrně vázaný seznam**. Přináší s sebou větší složitost provádění operací, nicméně umožňuje pohyb plovoucí proměnné ACT oběma směry – na předchůdce i následníka.

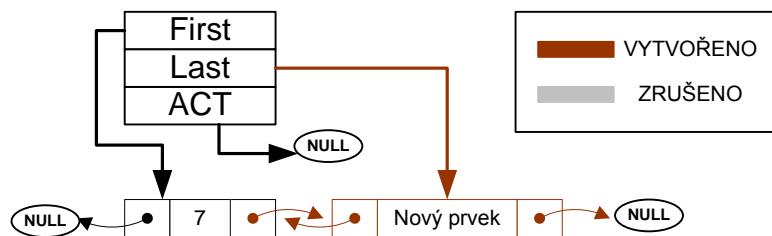
Typ	Obousměrně vázaný seznam
Obor hodnot	Množina všech lineárně seřazených posloupností všech délek.
Operace	Obsahuje veškeré operace, které obsahuje i jednosměrně vázaný seznam, navíc ale ještě metody: addLast removeLast getLast setActLast setActNext setActPrev removeAfterAct removeBeforeAct addBeforeAct

Tab. 4: Obousměrně vázaný lineární seznam.

Význam těchto nových operací je obdobný operacím z jednosměrně vázaného seznamu s tím rozdílem, že jsou zrcadlově otočeny. Výhodou je, že se pomocí ACT můžeme pohybovat v obou směrech a díky tomu u některých algoritmů například řazení získat významné urychlení (z kvadratické na lineární časovou složitost).

**Obr. 48: Obousměrně vázaný seznam o velikosti jednoho prvku.**

addLast: Přidá nový prvek na konec seznamu. Reference Last musí vždy ukazovat na poslední prvek seznamu. Naopak reference First musí vždy ukazovat na první pozici v seznamu.

**Obr. 49: Stav po vložení prvku do seznamu.**

removeLast: Odstraní poslední prvek seznamu. V případě, že je seznam prázdný, nestane se nic.

getLast: vrací referenci na poslední prvek seznamu.

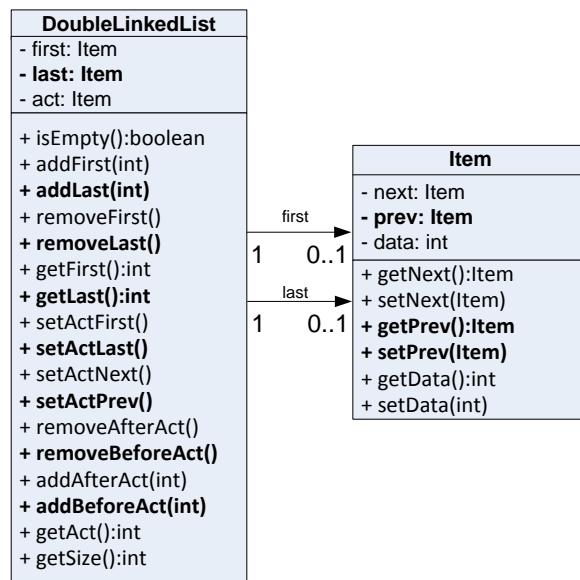
setActLast: nastaví referenci ACT na poslední prvek seznamu.

setActPrev: Nastaví proměnnou ACT na předchůdce aktuálního prvku. Pokud je jeho hodnota ne definovaná (nastavena na null) nenastane žádná změna.

removeBeforeAct: Odstraní prvek předešlý aktuálnímu. Pokud je ACT nedefinováno (== null) anebo ACT odkazuje na první prvek v seznamu, neprovede se žádná akce.

addBeforeAct: Přidá prvek za aktuální pozici reference ACT. Pokud je ACT rovno null, neprovede se žádná akce.

4.7.1 Reprezentace programem



Obr. 50: Návrh diagramu tříd pro obousměrně vázaný seznam v notaci jazyka UML[4]. Tučně jsou vyznačeny metody a atributy, které jsou navíc oproti jednosměrně vázanému lineárnímu seznamu.

```

public class Item {
    private Item next;
    private Item prev;
    private int data; // může být libovolný datový typ
}

public class DoubleLinkedList {
    private Item first;
    private Item last;
    private Item act;
    // ...
    public void addLast(Item newItem) {
        // kod pro vložení prvku
    }
    // .... ostatní metody
}
  
```

POZN.: Kompozice bude opět řešena prostřednictvím přístupu, kde si třída *DoubleLinkedList* bude spravovat své položky výhradně sama.

4.8 Srovnání časové složitosti polí a jednosměrně a obousměrně vázaných seznamů

Operace	Pole	Jednosměrně vázaný seznam	Obousměrně vázaný seznam
Vložení prvku na začátek	$O(n)$	$O(1)$	$O(1)$
Vložení prvku na konec	$O(1)$	$O(n)$	$O(1)$
Odstranění libovolného prvku	$O(n)$	$O(n)$	$O(n)$
Vyhledání prvku	$O(n)$	$O(n)$	$O(n)$
Reverze	$O(n)$	$O(n^2)$	$O(n)$

Tab. 5: Srovnání základních operací nad abstraktními datovými typy v notaci Omikron (nejhorší případ).

4.9 Fronta (FIFO, queue)

Fronta (neboli FIFO = First In First Out) je ADT, který pracuje stejně, jak napovídá název: na její konec přicházejí prvky a na druhé straně (tj. na začátku fronty) jsou prvky odebrány. Operace, které lze provádět nad ADT fronta tedy jsou:

- public void isEmpty();
- public void push(Item);
- public Item pop();

K implementaci ADT fronty můžeme s výhodou použít ADT oboustranně vázaný seznam.

Zásobníky jsou užity například pro:

- Moderní architektury HW
- Směrovače
- VM (virtuální stroje) obsahují zásobník (Sun Microsystems™ JVM)
- Algoritmy pro procházení stavového prostoru (Grafy/Umělá inteligence)
- Překladače jazyků
- ...

Dalším případem mohou být prioritní fronty. Prioritní fronty jsou modifikované tak, aby se prvky vkládaly na jistou pozici. K tomu aby se dalo určit na jakou, musí existovat nějaká operace, která dokáže porovnat prioritu hodnoty daného prvku.

Příklad: Implementujte vyrovnavací buffer pro audio/video přehrávač, který bude vyrovňávat časové zpoždění příchozích paketů.

4.10 Zásobník (LIFO, stack)

Zásobník neboli LIFO (Last In First Out) je ADT, který se používá velice často. Při volání programů a jeho funkcí a podfunkcí je vždy nutné, aby **program** věděl, do které části programu se má uskutečnit návrat.

Jeho funkce je naprosto totožná s **funkcí klasického zásobníku** například od pistole. Definované operace jsou:

- public void insert(Item);
- public Item remove();
- public boolean isEmpty();

Na jejich základě je možné **vkládat nové prvky na vrchol zásobníku, odstraňovat prvky z vrcholu zásobníku a zjišťovat, zdali je prázdný či nikoli.** K implementaci ADT zásobník si ve skutečnosti vystačíme s ADS **seznam a to i jednoduše vázaným**, pomocí kterého lze implementovat všechny tři metody.

Příklad: Napište program, který vypíše argumenty příkazové řádky v opačném pořadí (argumentů může být libovolný počet).

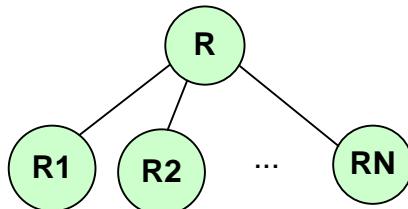
4.11 Literatura

- [1] BURGET, R. Hodnocení algoritmů, Přednáška MTIN.
- [2] J. M. Bull, L. A. Smith, L. Pottage and R. Freeman, „Benchmarking Java against C and Fortran for Scientific Applications“. Edinburgh Parallel Computing Centre.
[<http://www.philippes.com/JGI2001/finalpapers/18500097.pdf>](http://www.philippes.com/JGI2001/finalpapers/18500097.pdf)
- [3] J.P.Lewis and Ulrich Neumann, Performance of Java versus C++, Computer Graphics and Immersive Technology Lab, University of Southern California, 2004
[<http://www.idiom.com/~zilla/Computer/javaCbenchmark.html>](http://www.idiom.com/~zilla/Computer/javaCbenchmark.html)
- [4] BURGET, R. Úvod do UML, Přednáška MTIN.

5 Stromové datové struktury

5.1 ADT Strom

(**definice**): Strom T je konečná množina nula nebo více prvků (uzlů), z nichž jeden je označen jako kořen r (root) a zbývající uzly jsou rozděleny do $n \geq 0$ disjunktních podmnožin T_1, T_2, \dots, T_k , které jsou také stromy a jejichž kořeny r_1, r_2, \dots, r_k jsou následníky kořene r (viz Obr. 51).



Obr. 51: ADT strom

Typ	ADT Strom
Obor hodnot	Prázdný strom, jednoprvkový strom, dvouprvkový strom, ... N-prvkový strom. Pro každý z těchto stromů o počtu uzlů N může poté každá hodnota uzlu nabývat libovolných hodnot (tedy variace s opakováním). Obor hodnot je zadán pro každý strom samostatně.
Operace	Vkládání prvku Mazání prvku Vyhledávání prvku

Tab. 6: ADT strom

Existuje více definicí stromu, my si však vystačíme s touto jedinou.

Příklady použití stromů:

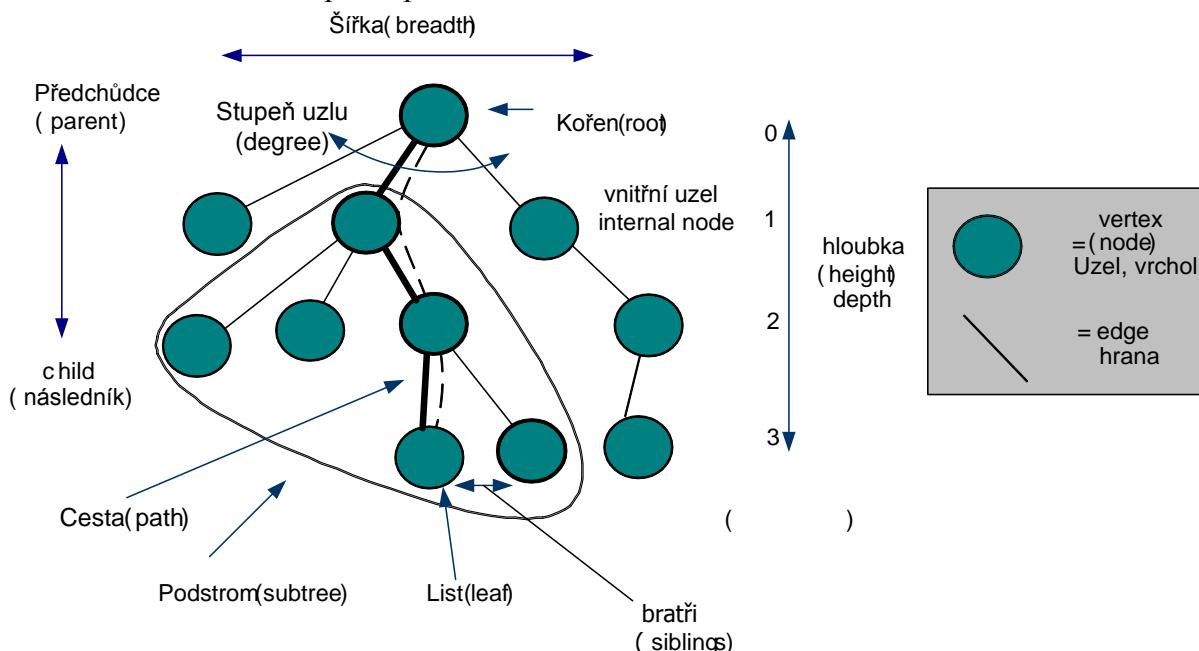
- jeden z možných způsobů indexování klíčů v databázích (systémech řízení báze dat)
- reprezentace znalostí, stavového prostoru v umělé inteligenci
- metody distribuce klíčů v kryptografii (broadcast encryption)
- jakékoli řazené struktury, množiny, atp.
- popis scény v oblasti zpracování a analýza obrazu, počítačová grafika
- vyhledávací stromy v databázových systémech
- rozhodovací stromy – expertní systémy
- organizace adresářů a souborů v souborovém systému OS,
- komprese dat (Hufmannovy kódovací stromy, fraktálová komprese)
- atd.

5.1.1 Základní pojmy

ADT strom je speciálním případem grafu [1], který je acyklický a uzly (vrcholy grafu) jsou uspořádány hierarchicky ve vztahu předchůdce-následovník.

Kořen je uzel, který nemá předchůdce a v celém stromu může být pouze jediný kořen. **Listy** (vnější uzly) jsou uzly, které nemají žádného následníka. **Vnitřní uzly** jsou uzly, které mají alespoň jednoho následníka. **Cesta** je: je-li n_1, n_2, \dots, n_k množina uzelů ve stromu takových, že n_i je předchůdce n_{i+1} , pro $1 \leq i \leq k$, pak se tato množina nazývá cesta z uzlu n_1 do n_k . **Délka cesty** je počet hran, které spojují uzly cesty. **Hloubka stromu** je maximální délka cesty ve stromu. **Sousedé** (siblings) jsou takové uzly, které mají společného předka (přímého). Počet přímých potomků uzlu se nazývá **stupeň uzlu**. **Stupeň stromu** je maximální stupeň uzlu v celém stromu. **Hloubka uzlu** je délka cesty od kořene do uzlu. **Výška stromu** je největší délka cesty od kořene k uzlu ve stromu. **Velikost uzlu** (size) je počet následníků, které uzel má + 1 (počet uzelů podstromu). Viz Obr. 52. Strom je **vyvážený**, jestliže v celém stromu neexistuje rozdíl v absolutní hodnotě kterýchkoli dvou cest od uzlu ke kořenu stromu větší než 1. Alternativní definice může být, že pro každý uzel je rozdíl hloubky levého a pravého podstromu v intervalu $<-1;1>$.

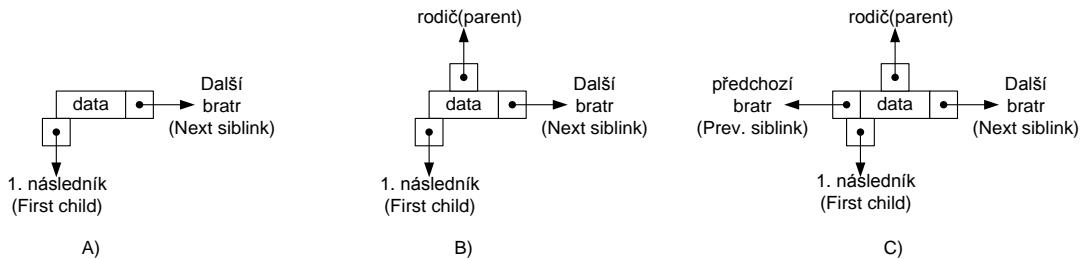
Konvence: Pokud v textu bude zmíněna časová složitost, např. $O(n)$, jedná se o složitost v závislosti na celkovém počtu prvků n .



Obr. 52: Příklad stromu a popis jeho vlastností

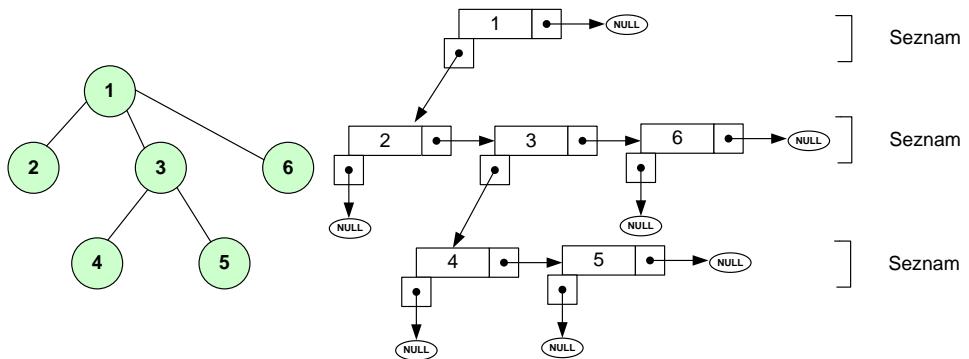
5.1.2 Reprezentace programem

Stromy je možné reprezentovat více způsoby a analogicky jako u ADT seznam je možné použít jednosměrně vázané stromy či obousměrně vázané stromy. Základní stavební blok je zobrazen na Obr. 53.



Obr. 53: Příklady základních stavebních bloků pro ADT stromy.

Příklad A) zobrazuje nejjednodušší variantu kde i stromová hierarchie i bratři (siblings) jsou spojeny jednosměrně. V příkladu B) jsou hierarchie stromu provázány obousměrně a v případě C) jsou obousměrně provázány jak bratři, tak i hierarchie (viz Obr. 54).



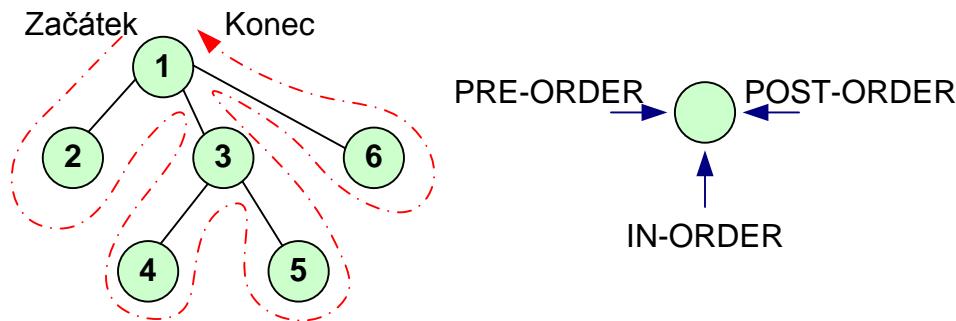
Obr. 54: Příklad implementace stromu využitím stavebního bloku z Obr. 3 varianta A).

Takový strom je nejobecnější variantou stromu. Je možné, aby stupeň stromu libovolně rostl a libovolně se zmenšoval. Naneštěstí obsahuje některé negativní vlastnosti lineárního seznamu a z toho důvodu není vždy vhodné používat tento přístup.

5.1.3 Metody průchodu ADT strom

Zatímco průchod lineárním seznamem je triviální a jednoduchý, u ADT strom existuje několik přístupů, jak pocházet celý strom a navštívit každý prvek právě jednou. Ze základních metod to jsou Pre-order, Post-order a In-Order.

Před zahájením průchodu stromem je nutné představit si obálku stromu (viz Obr. 55). Metodu pre-order si lze představit jako výčet prvků, které míjíme zleva. Pro nás příklad by to odpovídalo uzelům v pořadí 1, 2, 3, 4, 5, 6 a říká se mu prefixový zápis. Další metodu, In-Order, si lze představit jako výčet prvků v pořadí, jak bychom je míjeli zespod, tedy 2, 1, 4, 3, 5, 6. Všimněte si, že je v tomto pořadí zachováno levo-pravé pořadí. Poslední metodou je post-order, která by prvky míjela zprava a výsledná posloupnost by byla 2, 4, 5, 3, 6, 1 - postfixový zápis.



Obr. 55: Průchody stromem pre-order, in-order, post-order.

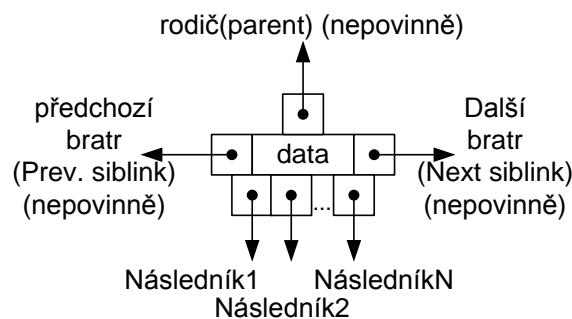
5.1.4 Nároky na implementaci

Implementaci průchodu stromem lze provést rekurzivní funkcí. V tomto případě bychom se opřeli o přítomnost **ADT zásobníku** přímo v překladači.

Druhým přístupem může být, že použijeme vlastní zásobník. V tom případě máme větší přehled o množství konzumované paměti a můžeme lépe analyzovat jednotlivé stavy výpočtu.

5.2 N-árni stromy

N-árni stromy jsou speciálním případem ADT strom. Platí pro ně, že stupeň stromu je roven **maximálně N**, tedy jinak řečeno, žádný uzel nemá více nežli N potomků. V tomto případě již číslo N není dynamická proměnná a za dobu existence stromu se nemění. Pokud si můžeme dovolit takové omezení, není nutné, abychom implementovali sousedící uzly jako ADT seznam, ale můžeme s výhodou použít přístupu přímého, pomocí N odkazů (Obr. 56).



Obr. 56: Příklad implementace N-árniho stromu.

Díky této optimalizaci redukujeme časovou složitost průchodu stromem na úrovni bratrů (siblings) z lineární složitosti O(n) na konstantní O(1) složitost.

N-árni stromem je implementována například adresářová struktura v OS Linux s využitím i-nodů (index-node).

5.3 Binární stromy

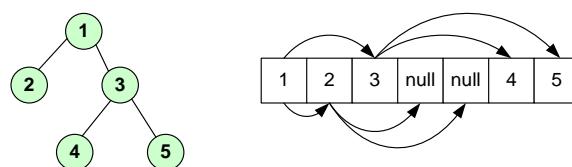
Binární stromy jsou speciálním případem N-árního stromu. Jsou to nejčastěji používanou datovou strukturou v počítačové vědě a zbytek textu bude věnován výhradně jím, nebude-li explicitně řečeno jinak. Jejich výhoda spočívá v redukci časové složitosti vyhledávání prvků z lineární $O(n)$ na (v ideálním případě) logaritmickou $O(\log_2 n)$.

Reprezentace binárních stromů v paměti je možná (a i doporučena) způsobem jakým se reprezentují N-árni stromy. Další možností je taktéž s využitím pole, jak je znázorněno na Obr. 57. Vztah, kterým lze určit levého a pravého potomka se dá vyjádřit vztahem (1), (2). Jak je ale zřejmé, z paměťového hlediska se nevždy musí jednat o efektivní řešení.

$$Next_{Left} = 2 * ACT + 1 \quad (1)$$

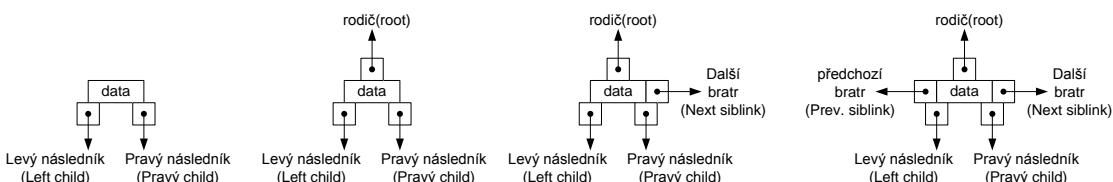
$$Next_{Right} = 2 * ACT + 2 \quad (2)$$

Kde $Next_{Left}$ je index levého potomka v poli, $Next_{Right}$ je index pravého potomka ve stromu a Act je aktuální index prvku, kde první pozice začíná hodnotou nula.



Obr. 57: Reprezentace binárního stromu v poli.

Na Obr. 58 jsou znázorněny datové struktury, pomocí kterých je možné reprezentovat binární strom v paměti počítače, jak je vidě



Obr. 58: Binární stromy jako speciální případ N-árních stromů, kde N=2.

5.3.1 Procházení binárními stromy s využitím rekurzivní funkce

5.3.1.1 Pre-order

```
public void printPreOrder(Node node) {
    if (node == null) return;

    System.out.print(" " + node.getData() + " ");
    printPreOrder (node.getLeft());
    printPreOrder (node.getRight());
}
```

5.3.1.2 In-order

```
public void printInOrder(Node node) {
    if (node == null) return;

    printInOrder(node.getLeft());
    System.out.print(" " + node.getData() + " ");
    printInOrder(node.getRight());
```

}

5.3.1.3 Post-order

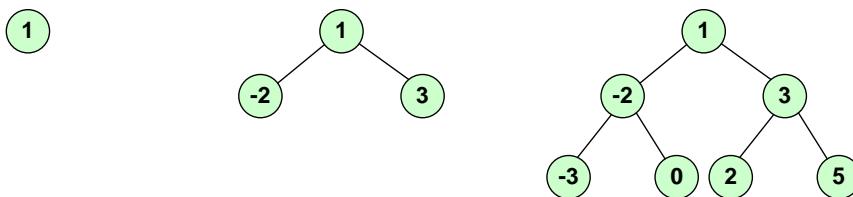
```
public void printPostOrder(Node node) {
    if (node == null) return;

    printPostOrder (node.getLeft());
    printPostOrder (node.getRight());
    System.out.print(" " + node.getData() + " ");
}
```

Všimněte si, že všechny tři implementace jsou téměř identické. Jediné co se liší, je pořadí části kódu, která vypisuje do standardního výstupu.

5.3.2 Úplný binární strom

(Definice) Úplný binární strom (Complete Binary Tree) je speciální případ binárního stromu, kde uzly v každé úrovni s výjimkou té nejhlubší vrstvy mají všechny uzly právě dva potomky. Na úrovni n (n = výška stromu) nemá žádný z uzlů žádného potomka (viz Obr. 59).



Obr. 59: Příklad kompletních binárních stromů s výškou 0, 1 a 2.

Obdobná definice platí i pro úplné n-ární stromy, ve většině případů je však binární strom lepší volbou.

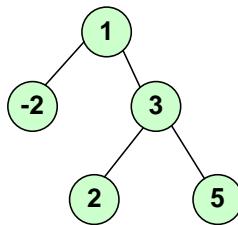
Příkladem použití úplných binárních stromů může být broadcastová kryptografie (konkrétně Subset Difference Tree[2]), která je základem pro šifrování HD-DVD a Blu-ray disků.

5.3.3 Binární vyhledávací stromy

V obecných binárních stromech není žádný požadavek na pořadí prvků. Z toho důvodu stromy nemohou urychlit přístup k prvku. Pokud v obecném binárním stromu chceme vyhledat některý prvek, musíme stejně některou z metod projít všechny prvky. Časová složitost je tedy ekvivalentní průchodu lineárním seznamem. Je tedy patrné, že pro vyhledávání prvků se výše zmíněné metody příliš nehodí. S řešením přichází binární vyhledávací stromy.

(Definice) Binární vyhledávací stromy jsou stromy, kde musí pro každý uzel platit, že hodnota všech potomků z levého podstromu je menší než hodnota rodiče, a hodnota všech potomků z pravého podstromu je větší než hodnota rodiče.

Pro přehlednost budou v rámci kurzu použita celá čísla jako hodnoty jednotlivých uzlů. Avšak v praxi mohou být hodnoty, stejně jako podmínky pro porovnání dvou uzlů naprostě libovolné. Namísto celých čísel mohou být použity například objekty, vektory, nebo klidně i vnořené ADT seznamy či ADT stromy – pokud tedy jsou známy metody pro vzájemné porovnání takových struktur.



Obr. 60: Binární vyhledávací strom

5.3.4 Vložení prvku

Důvodem proč jsme se nebaobili o metodách vkládání a mazání prvků z obecných stromů je, že zde neplatily žádná zvláštní pravidla a žádná omezení. Vložení prvku mohlo být libovolné a na libovolné místo. Z toho důvodu může být operace provedena téměř jakkoli, jen nesmí porušit strukturu stromu.

Díky tomu, že již jsou u vyhledávacích stromů jisté nároky na pořadí prvků, je operace vkládání prvku o něco málo složitější. Proces vkládání začíná na kořeni stromu. Pokud se hodnota vkládaného prvku rovná hodnotě aktuálního uzlu, hodnota se přepíše. V jiném případě, pokud je hodnota vkládaného uzlu menší a uzel nemá levého potomka, vloží se levý uzel. Pokud uzel levý uzel má, rekurzivně se aplikuje předchozí pravidlo na levý podstrom aktuálního uzlu. Obdobně, pokud je hodnota vkládaného uzlu větší, provádí se to stejně pro pravou větev stromu.

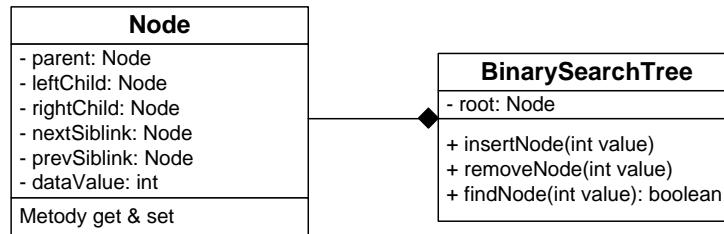
Co je důležité zmínit, že složitost vkládání je maximálně rovna $O(\log_2 h)$, kde h je výška stromu. Pokud se jedná o závislost na počtu prvků ve stromu, je bohužel stejná jako v případě seznamu $O(n)$. Důvod je vysvětlen v kapitole 5.3.6.

5.3.5 Odstranění prvku

Při odstraňování prvku máme tři možnosti: mazaný uzel nemá žádného potomka, má jednoho potomka anebo má levého i pravého potomka. V případě žádného potomka je operace triviální a prvek se jednoduše odstraní. V případě jednoho potomka se mazaný uzel nahradí potomkem. V třetím případě, tedy jestliže máme levého i pravého potomka, jsou dvě varianty – levá a pravá. V případě levé nalezneme nejpravější prvek levého podstromu (musí se tedy jednat o list, který má maximálně jednoho potomka) a ten vyjmeme (dle operace uvedené výše) a nahradíme jej za mazaný prvek. Pravá varianta je identická, jen zrcadlově otočená. Vyhledá se nejlevější list, kterým se nahradí mazaný prvek.

5.3.6 Implementace binárních vyhledávacích stromů

Níže je nastíněn zdrojový kód, kterým by bylo možné implementovat uzel binárního stromu a binární vyhledávací strom samotný. Je použita varianta zmíněná v N-árních stromech a bez obousměrně vázané hierarchie.



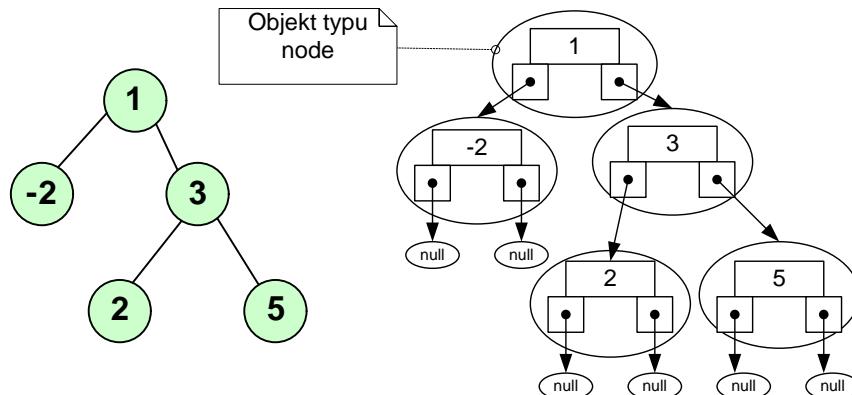
Obr. 61: Příklad implementace stromu v notaci UML [3].

```

public class TreeNode {
    private TreeNode parent;
    private TreeNode leftChild;
    private TreeNode rightChild;
    private TreeNode nextSibling;
    private TreeNode prevSibling;
    private int dataValue;
    //+ metody get* set* pro přístup k těmto atributům
    // (viz cvičení)
}

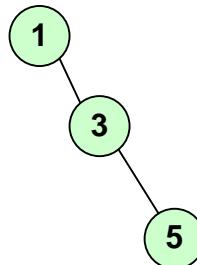
public class BinarySearchTree {
    private TreeNode root = null;
    public void insertNode(TreeNode node) {
    }
    public void removeNode(TreeNode node) {
    }
    public void findNode(TreeNode node) {
    }
    // + další metody například pro průchod in-order, post-order,
    // pre-order, atd.
}

```

Obr. 62: Strom a jeho ekvivalent reprezentovaný objekty typu z Obr. 58a (nejjednodušší stavební blok).
Všimněte si, že jednotlivé objekty neukazují na čísla (int), ale na jiné objekty (typu Node).

5.3.7 Nevyvážené stromy

V extrémním případě může vlivem operací přidávání a mazání prvků dojít k degradaci binárního stromu na lineární seznam (viz Obr. 63). To by mělo za následek vyšší časovou složitost vkládání a při tom stejnou časovou složitost odstraňování prvků. To je také příčinou, proč se standardní binární vyhledávací stromy téměř nepoužívají a AVL či Red-Black stromy jsou lepší variantou.



Obr. 63: Příklad binárního stromu degradovaného na lineární seznam.

5.3.8 Vyhledání prvku

Jelikož jsou prvky v binárním vyhledávacím stromu seřazeny, je možné urychlit vyhledávání. To je v případě stromů velice výrazné v porovnání s ADT seznam. Vyhledávání probíhá tak, že vždy začínáme v kořenovém uzlu. Porovnáme, zdali aktuální prvek je hledaný. V případě, že nikoli, vydáme se doleva od aktuálního uzlu (hledaná hodnota je menší než hodnota aktuálního uzlu), popřípadě doprava (hodnota vyhledávaného prvku je vyšší než hodnota aktuální). Stejné porovnání provedeme i na další tohoto následníka a pokračujeme tak dlouho, dokud prvek nenalezneme, či nenarazíme na konec stromu.

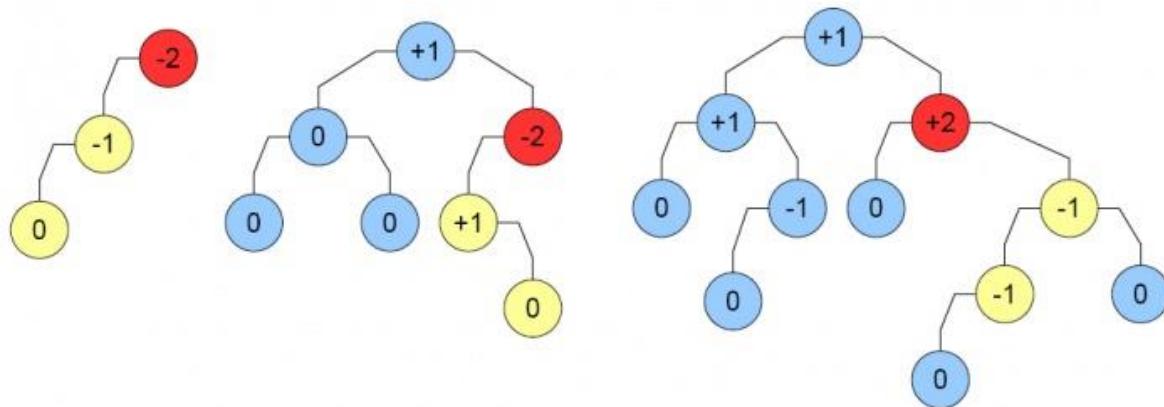
Pro vyhledávání ve stromu je ideální taková konfigurace, kdy je strom vyvážený. V extrémním případě můžeme strom degradovat na lineární seznam (viz Obr. 63).

5.4 AVL stromy

AVL strom je výškově vyvážený binární vyhledávací strom, pro který platí, že pro libovolný vnitřní uzel stromu se hloubka levého a pravého podstromu liší nejvýše o 1. K tomu, aby tyto podmínky byly dodrženy, je nutné upravit funkci vkládání a funkci odstraňování prvků ze stromu.

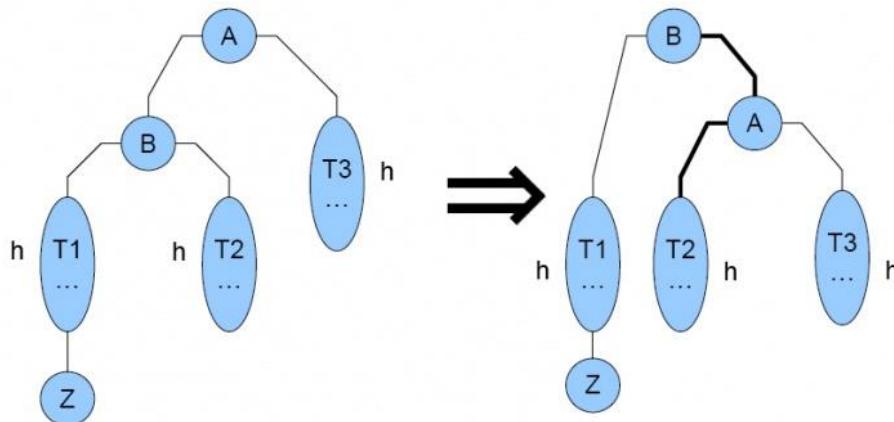
5.4.1 Vkládání prvku

Vkládání u AVL stromů probíhá ve dvou fázích – v první dojde k vložení prvku stejně jako u vyhledávacích stromů. Následuje kontrola vyváženosti – tedy, že rozdíl výšky levého a pravého podstromu je v intervalu $<-1,1>$. Pokud tomu tak není, následuje vyvažování pomocí některé z operací SLR, SRR, DLR a DRR [2]. Na Obr. 64 jsou znázorněny 3 nevyvážené stromy, vyznačeno místo nevyváženosti a vyznačena cesta



Obr. 64: Příklady výškově vyvážených stromů. Hodnoty uvnitř nejsou hodnoty dat, ale vyváženosť jednotlivých větví. Červeně jsou označeny nevyvážené uzly. Žlutě je označena trojice prvků, se kterými se příslušná operace musí provést.

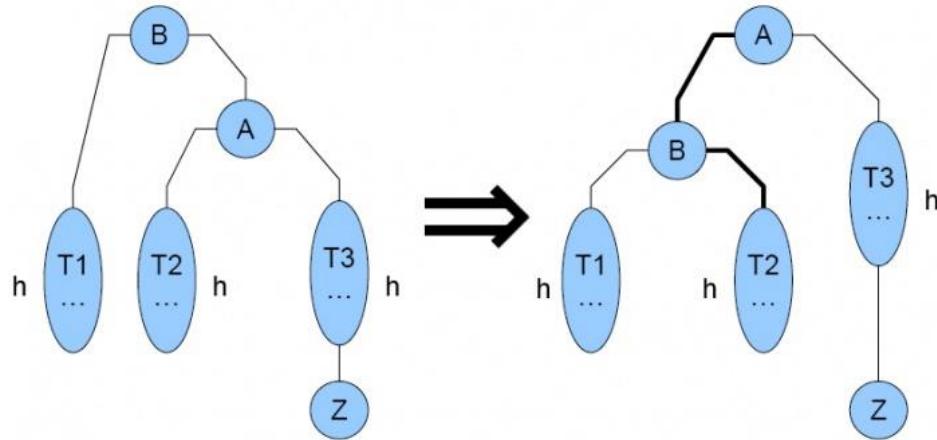
5.4.1.1 SLR (Jednoduchá levá rotace, Single Left Rotation)



Obr. 65: Princip jednoduché levé rotace

Uzly si v tomto případě představte jako zavěšené na šňůrce. Ve stavu 1 je pověšeno vše za uzel A, ve stavu 2 za uzel B. T2 je přepojeno z uzlu A na volné rameno uzlu B. Jednoduchou levou používáme, pokud vyvažujeme přímou větev, tj. jsou-li znaménka stupně vyváženosť stejná (viz Obr. 65 a struktura bodů A,B,T3).

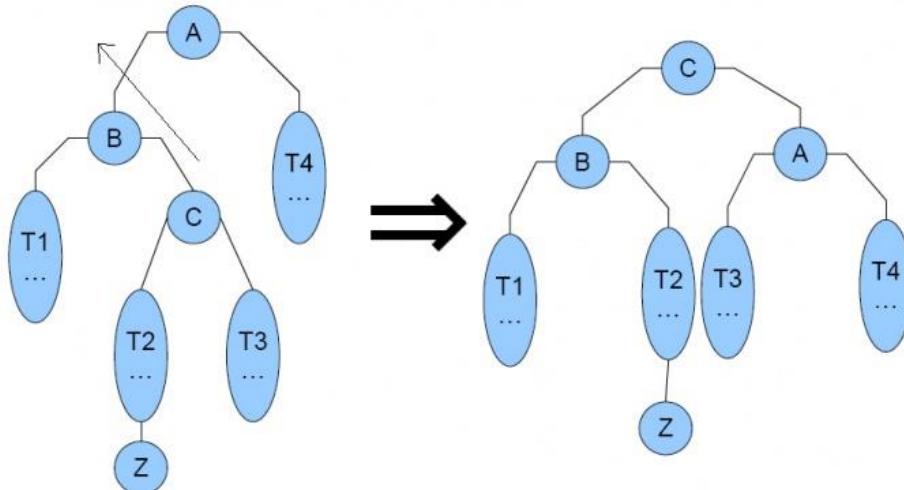
5.4.1.2 SRR (Jednoduchá pravá rotace, Single Right Rotation)



Obr. 66: Princip jednoduché pravé rotace

Jedná se o identickou operaci, jen zrcadlově otočenou. Jednoduchou pravou používáme, pokud vyvažujeme přímou větev, tj. jsou-li znaménka stupně vyváženosti stejná (viz Obr. 68 a struktura bodů A,B,T3).

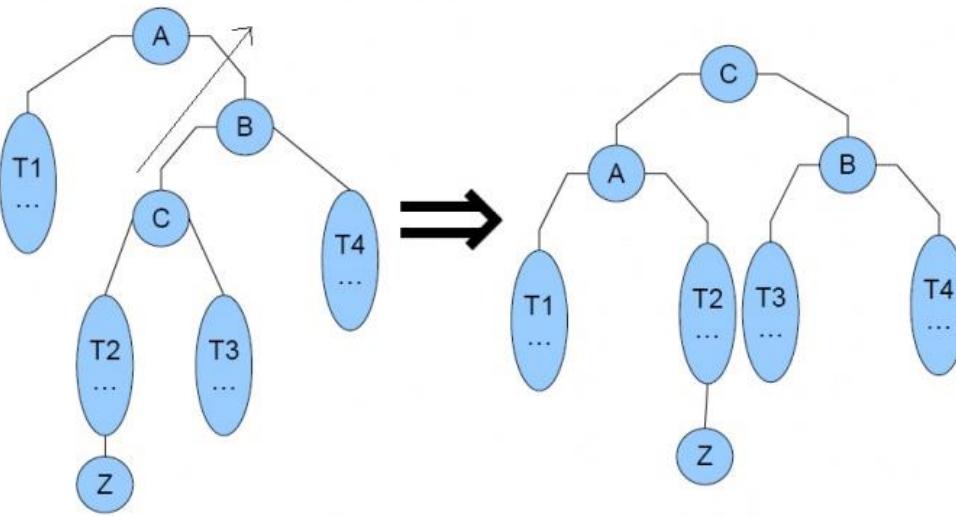
5.4.1.3 DLR (Dvojnásobná levá rotace, Double Left Rotation)



Obr. 67: Dvojnásobná levá rotace

DLR používáme, pokud vyvažujeme „zalomenou“ větev (viz Obr. 67 a struktura bodů A,B,C).

5.4.1.4 DLR (Dvojnásobná pravá rotace, Double Right Rotation)



Obr. 68: Princip dvojnásobné pravé rotace

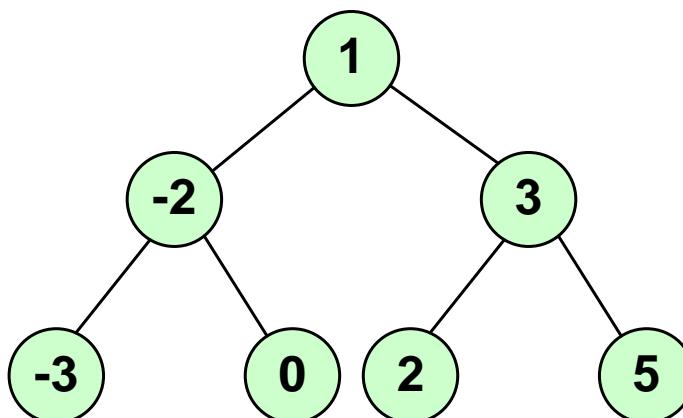
DLR používáme, pokud vyvažujeme „zalomenou“ větev (viz Obr. 68 a struktura bodů A,B,C).

5.4.2 Odstraňování prvku

Mazání prvku je z části identické s operací, která je známa z obecných vyhledávacích stromů. Navíc je ale ještě nutné po smazání prvku, aby pro nadřazené uzly bylo zkontrolováno, že strom je stále vyvážen a popřípadě provést potřebné operace [2].

5.4.3 Příklady

- Do prázdného stromu vložte prvky: 2, 3, 1
- Do prázdného stromu vložte prvky: 1, 2, 3
- Do prázdného stromu vložte prvky: 3, 2, 1
- Do prázdného stromu vložte prvky: 3, 1, 2
- Do prázdného stromu vložte prvky: 1, 3, 2



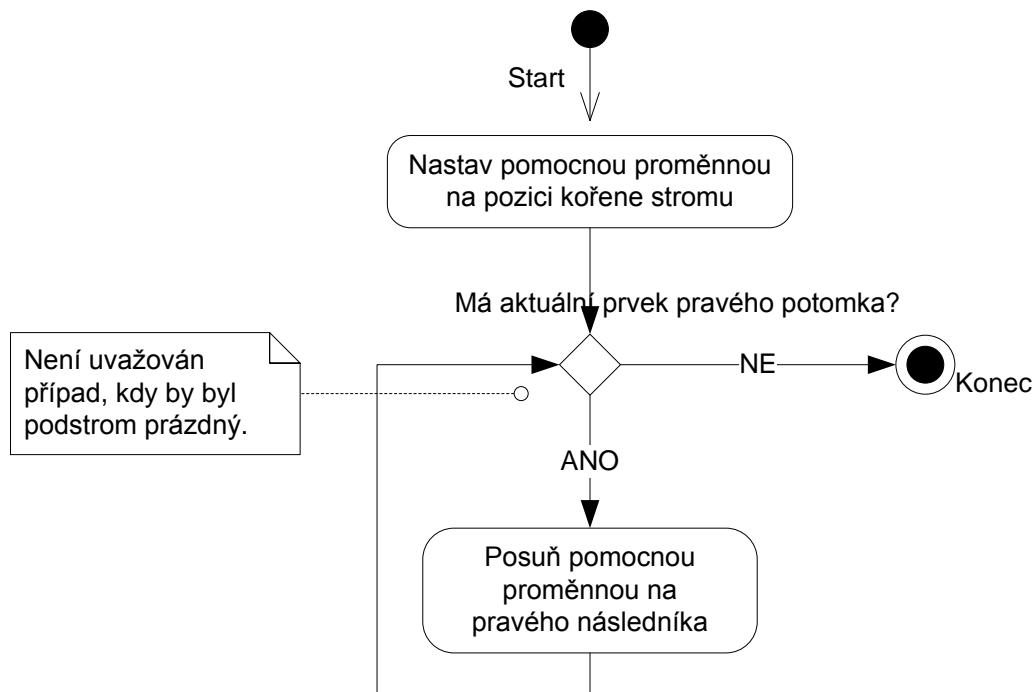
Obr. 69: Příklad stromu

- V Obr. 69 ohodnoťte váhu jednotlivých uzlů
- Odstraňte prvek 1 z Obr. 69
- Odstraňte prvek 3 z Obr. 69

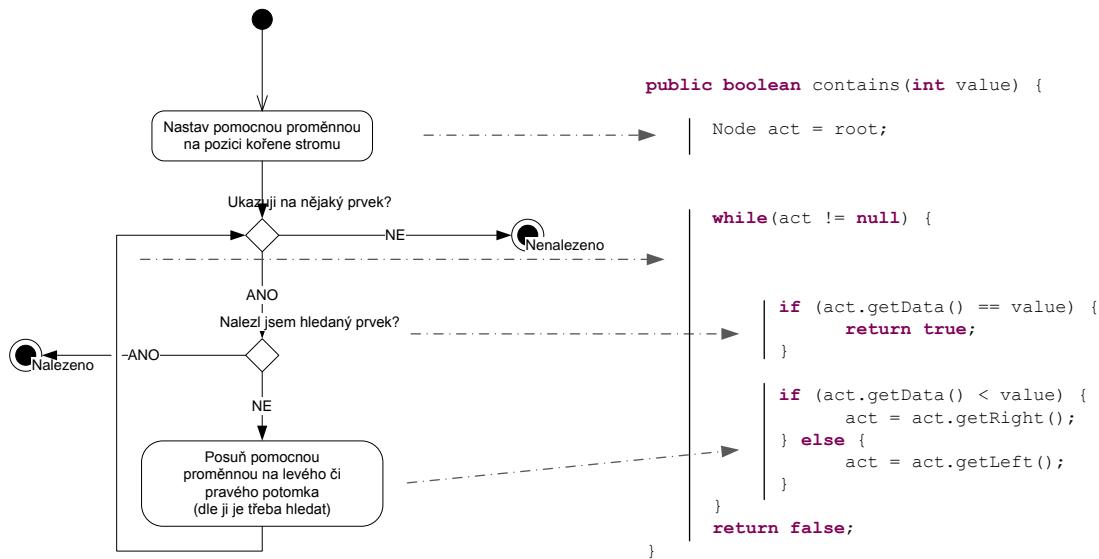
5.4.4 Implementace základních operací ve stromových strukturách

Přestože všechno doporučuji pokusit se nejprve porozumět daným operacím sám vzít si tužku a papír a kreslit (naučíte se tak algoritmizovat nejen tento příklad, ale obecně všechny, kterým porozumíte), níže jsou uvedeny některá vodítka, která mohou napomoci.

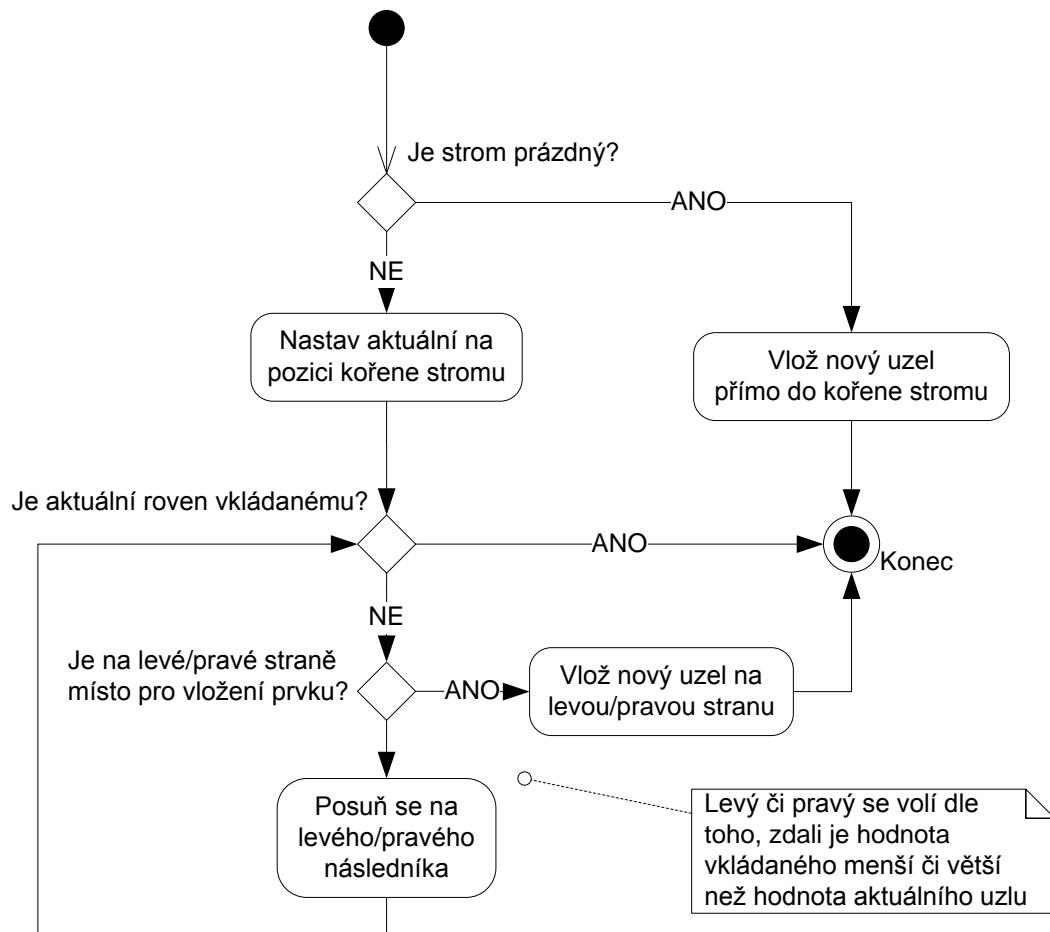
5.4.4.1 Nalezení nejpravějšího prvku



5.4.4.2 Existuje prvek ve stromu?



5.4.4.3 Vložit prvek do stromu



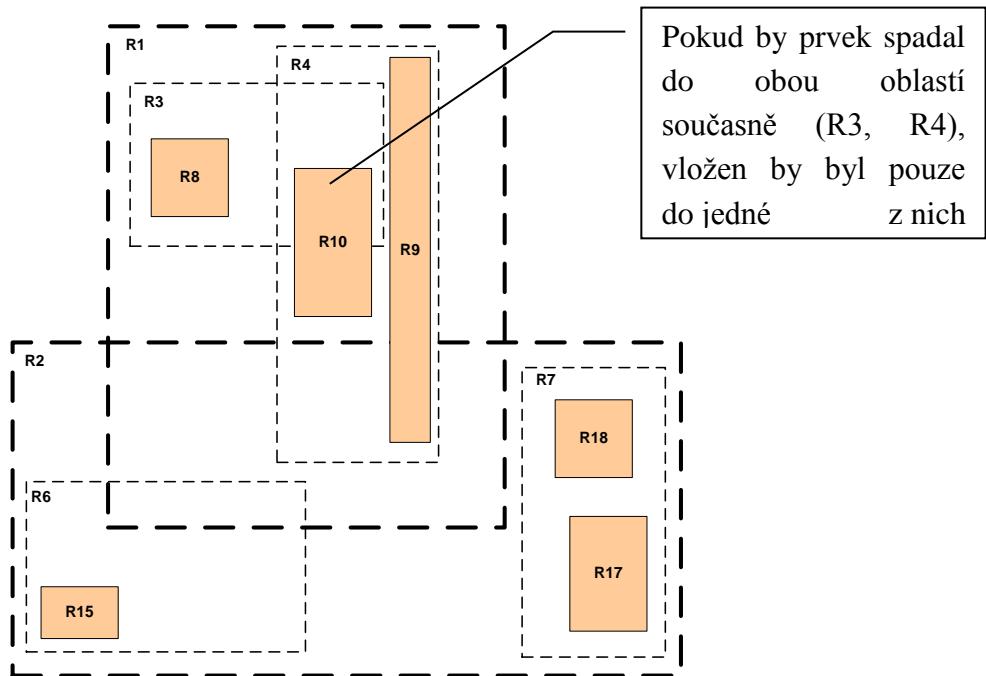
5.4.4.4 Odstranit prvek ze stromu

Nejprve je zapotřebí prvek vyhledat (viz výše). Pokud prvek nebyl nalezen, neděje se nic. Pokud prvek byl nalezen – ověří se, zdali nemá žádného, má jednoho, či dva potomky. V případě žádného či jednoho uzlu je řešení relativně jednoduché (rodič změní vazbu). V případě obou potomků se smaže nejpravější v levém podstromu (nalezení nejpravějšího, viz výše) a původní mazaný se nahradí tímto nejpravějším.

5.5 R stromy

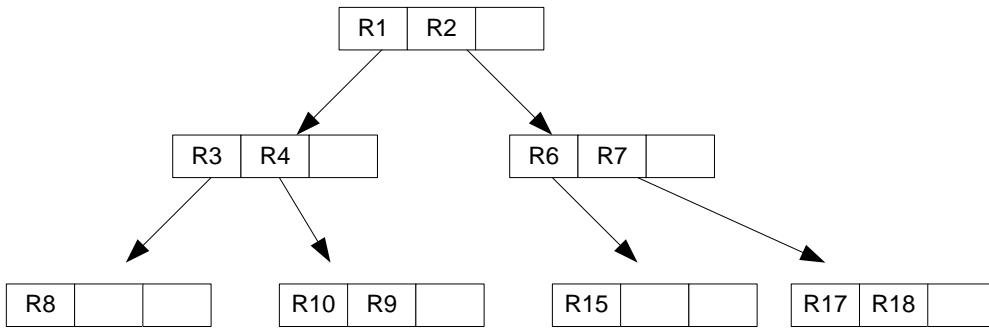
R stromy jsou velmi podobné binárním stromům. Na rozdíl od binárních stromů jsou více vhodné pro prostorově strukturovaná data. Jako příklad může být problém: „Nalezni veškeré bankomaty v blízkosti pozice (X,Y)“.

Data jsou v tomto případě organizována do hierarchicky vnořených obdélníků. Pro různé úrovně je dokonce možné aby se vzájemně překrývaly. Příklad je znázorněn na obrázku Obr. 70.



Obr. 70: Příklad datové struktury R-stromu

Odpovídající stromová struktura je znázorněna na obrázku Obr. 71.



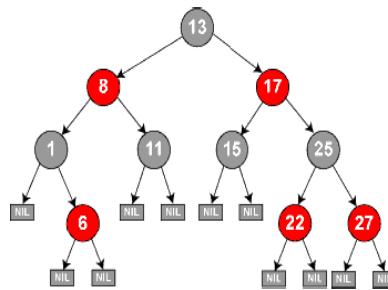
Obr. 71: Reprezentace pomocí R stromu.

Při vkládání a mazání objektů se berou v potaz omezující oblasti. Při vkládání prvku ležícího přímo v oblasti se použije daná oblast (popřípadě libovolná z nich). Pokud nově vkládaný objekt nespadá do žádné dosud existující oblasti, vloží se do takové, která vyžaduje nejmenší zvětšení.

R stromy jsou často používány v databázích (MySQL, MS SQL, Oracle, Posgres, ...).

5.6 Red-Black stromy

Nevýhodou AVL stromů je, že vkládání anebo odstraňování prvku může znamenat i více než jednu operaci rotace. Toto řeší Red-Black stromy, kde maximální počet rotací je $O(1)$.



Obr. 72: Příklad Red-black stromů

Stejně jako AVL stromy i Red-Black stromy jsou samovyvažující se stromové datové struktury. Časová složitost vyvážení je různě časově složitá, ale i přesto pracují v konstantním čase.

Jsou celkem dva rozdíly mezi těmito dvěma strukturami. V případě n prvků u AVL stromů je maximální výška $1.44 * \log_2(n)$, zatímco u Red-Black stromů je maximální výška $2 * \log_2(n)$. Na druhou stranu, u AVL stromů je možné i více rotací nežli jediná. Red-Black stromy garantují maximálně 1 rotaci na vložený prvek.

Red-black stromy používají 4 operace pro vkládání a 6 operací pro mazání z datové struktury. Operace jsou podobné AVL stromům, které zajišťují vyváženosť stromu. V rámci tohoto kurzu nebudou probírány do hloubky. Zájemci se mohou dočíst více na [4].

Red-Black tree jsou v praxi nejčastěji používané stromové struktury. V jazyku JAVA i standardní knihovně C++ pokud je v dokumentaci zmínka o stromových datových strukturách, jsou tím méněny práve Red-Black stromy.

5.7 Časová a paměťová složitost

Operace	Binární vyhledávací strom	AVL strom	Seznam	Red-Black strom
Vkládání	$O(n)$	$O(2 * \log_2 n)$	$O(n)$	$O(\log_2 n + 1)$
Mazání	$O(n)$	$O(2 * \log_2 n)$	$O(n)$	$O(\log_2 n + 1)$
Vyhledávání	$O(n)$	$O(1,44 * \log_2 n)$	$O(n)$	$O(2.0 * \log_2 n)$

I přesto, že v případě notace Omikron se zdají být výsledky seznamu a binárního stromu obdobné, průměrná složitost bude v případě binárního vyhledávacího stromu výrazně lepší. Díky tomu, že použijeme například v databázových systémech stromové indexování, znamená to výraznou úsporu času. V tabulce níže je znázorněn počet prvků a počet operací, které se provedou v případě lineárního seznamu a v případě binárního stromu.

Počet prvků	1	2	4	8	16	32	64	128	256
Seznam	1	2	4	8	16	32	64	128	256
AVL strom	0	2	4	6	8	10	12	14	16
Red-Black strom	0	2	3	5	6	8	9	11	12

Počet prvků	512	1024	2048	4096	8192	16384	32768	65536	131072
Seznam	512	1024	2048	4096	8192	16384	32768	65536	131072
AVL strom	18	20	22	24	26	28	30	32	34
Red-Black strom	13	15	16	18	19	21	22	24	25

Počet prvků	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432	67108864
Seznam	262144	524288	1048576	2097152	4194304	8388608	16777216	33554432	67108864
AVL strom	36	38	40	42	44	46	48	50	52
Red-Black strom	26	28	29	31	32	34	35	36	38

Počet prvků	1000000	1E+26	1E+46	1E+66	1E+86	1E+106	1E+126	1E+146	1E+166
Seznam	1000000	1E+26	1E+46	1E+66	1E+86	1E+106	1E+126	1E+146	1E+166
AVL strom	40	173	306	439	572	705	838	971	1103
Red-Black strom	29	125	221	316	412	508	603	699	795

Tab. 7: Tabulka závislosti počtu kroků při vyhledávání v závislosti na počtu prvků obsažených ve struktuře.

V poslední tabulce vidíme, že pro 67 milionů prvků to pro vyhledávání prvku v případě seznamu znamená 67 milionů operací, v případě Red-Black stromu to bude jen 27 kroků a v případě AVL stromu dokonce jen 6 kroků.

5.8 Souhrnné srovnání

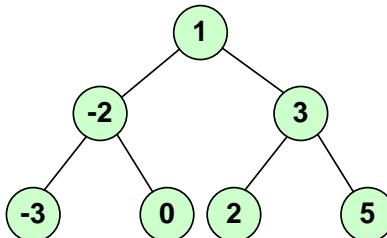
Datová struktura	Výhoda	Nevýhoda
Pole	Rychlé vkládání. Rychlý přístup, pokud je znám index položky.	Pomalé mazání a vyhledávání.
Seřazené pole	Rychlejší vyhledávání	Pomalé vkládání a mazání
Zásobník	Poskytuje last-in first-out přístup.	Pomalý přístup k jiným položkám.
Fronta	Rychlý přístup k první a poslední položce.	Pomalý přístup k ostatním položkám.
Lineární seznam	Efektivní využití paměti	Pomalé vyhledávání. Pomalý přístup k položkám.
Binární strom	Rychlé vkládání, mazání a hledání (pokud zůstává vyvážený)	Algoritmus mazání je časově náročný.
AVL & Red-black stromy	Rychlé operace vyhledávání, hledání a mazání. Stromy jsou vyvážené.	Složité
Hashovací tabulka	Rychlé vkládání. Pokud existuje klíč, velmi rychlý přístup.	Pomalé mazání prvku. Pomalý přístup, pokud klíč není nalezen. Neefektivní využití paměti.
Graf	Modeluje situace reálného světa	Některé algoritmy jsou nad nimi složité a řád časové složitosti je velmi vysoký

5.9 Literatura

- [1] BURGET, R. Grafy, Přednáška MTIN.
- [2] AVL tree applet, Univeristy of Ottawa,
[<http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>](http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html)
- [3] BURGET, R. Unified Modeling Language, Přednáška MTIN.
- [4] Eitan Gurari, „Red-black Trees“, Department of Computer Science and Engineering at the Ohio State University, [<http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch11.html>](http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch11.html)
- [5] Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft. „Data Structures and Algorithms“, Addison-Wesley Series in Computer Science and Information
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, „Introduction to Algorithms“, The MIT Press; 2nd edition, ISBN-10: 0262032937

5.10 Otázky k opakování

-  Jaké jsou výhody/nevýhody ADT binární strom oproti ADT lineární seznam?
-  Je zadán následující binární strom. Vypište prvky v pořadí průchodu pre-order, in-order a post-order.



- 3. Mějme binární vyhledávací strom (bez vyvažování). Vložte do něj prvky 5, 3, 7, 10.
-  Je strom z předchozího příkladu vyvážený? Byl strom vyvážený i v průběhu přidávání prvků?
-  Jaký extrémní případ binárních stromů existuje a jaké má vlastnosti v porovnání
- 6. Mějme binární vyhledávací strom s vyvažováním pomocí algoritmu AVL. Vložte do prázdného stromu prvky v následujícím pořadí:
 - a. 2, 3, 1
 - b. 1, 2, 3
 - c. 3,2,1
 - d. 1,3,2
 - e. 3, 1, 2
 - f. 1, 2, 3, 4, 5
 - g. 1, 2, 3, 5, 4
 - h. 1, 2, 3, -2, -1
 - i. 1, 2, 3, -2, -3
-  Jaký je rozdíl mezi AVL stromy a Red-black stromy?
- 8. Odstranění prvku s žádným potomkem, s jedním potomkem, s oběma potomky (levá varianta)
-  Pomocí kterého ADT lze implementovat ADT zásobník?
-  Pomocí kterého ADT lze implementovat ADT fronta?

6 Tabulkové datové struktury

6.1 Datový typ tabulka

ADT tabulka (v české terminologii také „tabulka s rozptýlenými položkami“). Implementuje tzv. asociativní pole, tj. pole, které nemusí být nutně indexováno pouze ordinálním datovým typem², ale libovolným typem (např. řetězcem). Nejčastější implementace jsou založeny na použití statických polí ve spojení s hashovací funkcí, nebo používají AVL stromy. Přitom **operace vkládání je v případě hashovací funkce výrazně rychlejší** a do jistého rozsahu dává dokonce lepší výsledky než AVL strom.

Příklad použití **hashovaných funkcí**:

- Souborové systémy, komparace souborů, ukládání hesel, ověření integrity, autentizace
- Tabulky symbolů v překladačích, databázové systémy, kontrolní součty
- Algoritmy pro cache, kryptografické protokoly, pseudonáhodné generátory
- A mnoho dalších...

Aby bylo jasnější, jak hashovaní funkce pracují, lze nalézt analogii s matematicko-fyzikálními tabulkami. Ještě před příchodem doby výpočetních systémů se matematické tabulky používaly pro výpočty velice často. V nich byly uvedeny pro jednotlivá čísla např. hodnoty jejich logaritmů, převrácených hodnot, mocnin ap. Odpovídající abstrakcí je organizace dat, kde podle jednoznačné hodnoty klíče prvku množiny zpřístupňujeme další hodnoty uložené v prvku množiny. Připomeňme, že obdobně to funguje i v případě AVL stromů – zde sice, pro jednoduchost, jsou pro data a klíč použita celá čísla, nicméně klíč a hodnota se může lišit, stejně jako v ADT tabulka.

6.1.1 Tabulka s přímým adresováním

Tabulku s přímým přístupem lze použít tehdy, je-li znám celkový počet klíčů $K=[k_1, \dots, k_N]$, které se budou v tabulce používat, a je-li možné najít jednoznačnou mapovací funkci $f(k_i)=i$ (pro $i=1,2,\dots,N$) pro všechny prvky množiny K . V tomto případě je možné vytvořit tabulku s přímým přístupem.

Příklad:

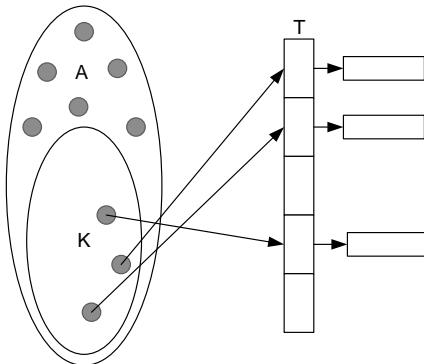
Množina všech klíčů $K=['Petr', 'Pavel', 'Karel', 'Jan']$,
Mapovací funkce $f(K)=[\text{'Petr'} \rightarrow 1, \text{'Pavel'} \rightarrow 2, \text{'Karel'} \rightarrow 3, \text{'Jan'} \rightarrow 4]$

V případě matematických tabulek je klíčem číslo a zjišťujeme jeho logaritmus, převrácenou hodnotu, atd. V tabulce startujících sportovců je klíčem startovní číslo a z tabulky zjistíme jméno sportovce a případně další uchovávané informace jako věk, nejlepší dosavadní výkon

² Ordinální datové typy jsou všechny typy celočíselné, typ *Boolean* a *Char*. Ordinální jsou typy, pro něž je definováno vzájemně jednoznačné zobrazení množiny jejich hodnot na množinu *ordinálních čísel* těchto hodnot. Každé hodnotě ordinálního typu je tedy tímto zobrazením přiřazeno její ordinální číslo. Každá hodnota má předchůdce a následníka.

ap. Pokud vytvoříme tabulku převrácených hodnot pro čísla od 1 do 1000, potom bude obsahovat tuto hodnotu pro každou hodnotu klíče. Efektivní implementací takové tabulky je pole, kde klíč je přímo indexem prvku, ve kterém je uchována hodnota.

Časová složitost vyhledání prvku bude v tomto případě velice rychlá – konstantní O(1).



Obr. 73: Množina všech hodnot A a množina použitých hodnot K mapovaná do tabulky T

6.1.2 Hashovací tabulky

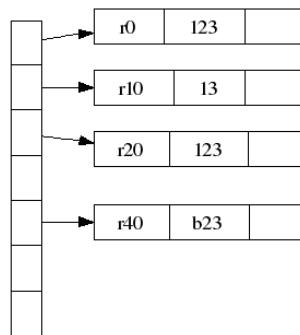
Problém tabulky s přímým adresováním je zřejmý – jestliže je počet všech hodnot $|A|$ velký, je nutné, aby byly udržovány všechny hodnoty z množiny A , které budou obsaženy v tabulce T . Jestliže velikost množiny $|K|$ je dostatečně menší než velikost množiny všech hodnot $|A|$, paměťové nároky mohou být redukovány na průměrnou složitost $\Theta(1)$ a přitom složitost vyhledání prvku bude zachována $O(1)$.

Uvažujme množinu A , množinu veškerých prvků, a množinu K , množinu všech hodnot klíče. V případě hashovacích tabulek platí, že velikost množiny prvků je větší než množina všech hodnot klíčů $|A| >> |K|$. Pokud prvky množiny resp. ukazatele či reference na ně budeme kvůli rychlosti přístupu chtít uchovávat v poli, abychom nemrhali pamětí, toto pole by mělo mít rozměr úměrný $|A|$. V tabulce budou nyní prvky množiny identifikované svým klíčem zpřístupňovány pomocí indexu s hodnotami 0 až $n-1$. Potřebujeme tedy hodnotu klíče prvku transformovat na hodnotu indexu, jinými slovy musíme definovat funkci

$$h: K \rightarrow 0, 1, \dots, m-1, \quad (1)$$

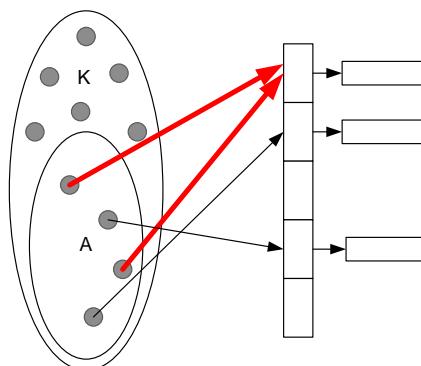
kterou budeme nazývat rozptylovou/hashovací fcí. Prvek s klíčem $k \in K$ bude mít v tabulce index $h(k)$.

Vzhledem k tomu, že $m << |K|$, tedy počet všech klíčů je daleko větší, než počet hodnot indexů, na které je zobrazujeme, nevyhnutně musí rozptylová funkce zobrazit obecně dva a více různých klíčů na stejný index, tj. pro $k_1 \neq k_2$ a $k_1, k_2 \in K$ bude $h(k_1) = h(k_2)$, což je také znázorněno na obrázku Obr. 74. Tento jev nazýváme kolizí. V takovém případě by to znamenalo přepsání stávající hodnoty a nesprávnou funkčnost tabulky. Z toho důvodu je nezbytné problém kolizí řešit.



Obr. 74: Příklad hashovací tabulky

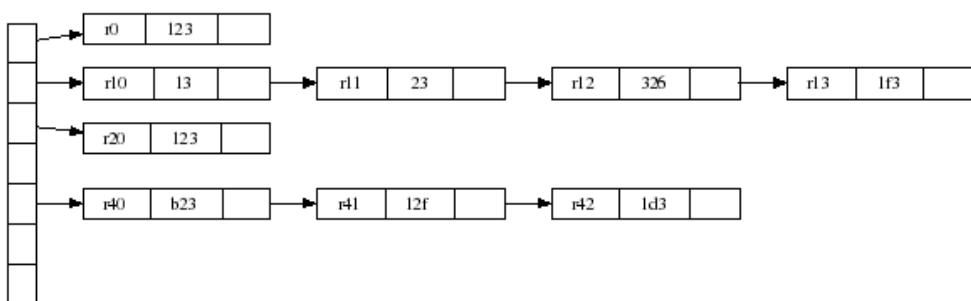
6.1.3 Řešení kolizí



Obr. 75: Příklad kolize. Dva prvky z množiny A mají stejnou hodnotu hashovací funkce.

6.1.3.1 Hashovací tabulky se zřetězením

V případě, že dojde ke kolizi a dva či více prvků ukazují na místo v tabulce se stejnou adresou a je použito zřetězení, vytvoří se na této adrese seznam kolidujících klíčů s pomocí lineárního seznamu. Poslední přidaný klíč je vždy umístěn na začátek seznamu, aby se zachoval konstantní čas vkládání. Tím získáme kompromis mezi rychlostí a množstvím paměti, které je nutné uchovávat (Obr. 76).



Obr. 76: Příklad řetězené hashovací tabulky

6.1.3.2 Otevřené adresování – lineární vkládání

Pokud nastane případ kolize, tedy $h(k_1) = h(k_2)$, kde $k_1 \neq k_2$, to znamená, že výsledek hashování funkce ukazuje na již obsazené místo v hashovaní tabulce, vložíme prvek na následující místo, tedy $h(k)+1$. Pokud i toto je obsazeno, tak se pokusí o $h(k)+2$. Při této metodě je třeba dodržovat, aby se tabulka nezaplnila zcela, protože by se vyhledávání v ní

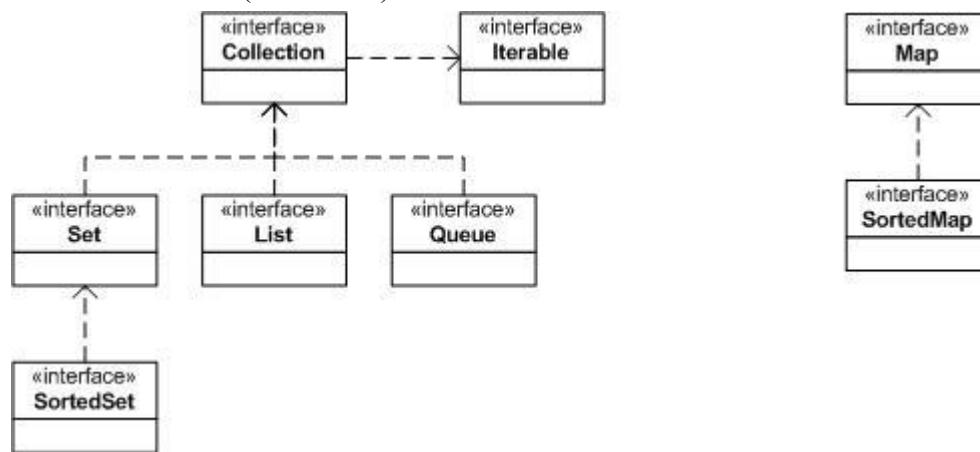
mohlo stát nekonečnou smyčkou. Čím více kolizí během vkládání vznikne, tím se budou tvořit delší souvislé skupiny obsazených buněk, kterým říkáme shluky nebo také klastry. Naším cílem ovšem je, aby klastrů bylo co nejméně a byly co nejkratší.

6.1.3.3 Otevřené adresování – dvojité hashování

Metoda s dvojitým hashováním odstraňuje nedostatky metody lineárního vkládání. Principem je, že pokud dojde při vkládání prvku ke kolizi, není tvořen shluk hodnot na dané pozici jako v předešlém případě, ale je spočítána sekundární hashovací funkce. Tato druhá hashovací funkce musí být volena nanejvýš opatrně, protože pokud by stále docházelo ke kolizím, program by vůbec nemusel pracovat.

6.2 Java Collections Framework

Jazyk JAVA v jeho základní knihovně [2] disponuje bohatou výbavou datových typů a z toho důvodu není ve většině případů nutné implementovat vlastní ADT, lze využít již některý z existujících. Kolekce je jednoduše řečeno objekt, který umí seskupovat více prvků do jedné množiny. Kolekcemi jsou obvykle reprezentovány datové položky, které spolu logicky souvisejí, např. telefonní seznam. V prvních verzích jazyka JAVA (až do verze 1.2) byly implementovány pouze kolekce typu Vector, Hashtable a pole. V nových verzích je již zakomponovaná jednotná architektura Java Collection Framework (viz Obr. 5 a Obr. 6) a stávající kolekce musely být upraveny, aby splňovaly podmínky implementace nových rozhraní a zároveň si zachovaly zpětnou kompatibilitu. **Mezi klíčová rozhraní patří Collection, Set (a SortedSet), List, Queue, Map (a SortedMap)**. Tato rozhraní jsou generická a vzájemně konvretovatelná (viz Obr. 5).



Obr. 77: Rozhraní základních datových typů v jazyce JAVA

6.2.1 Collection

Jde o nejobecnější rozhraní, neexistuje jeho přímá implementace. Jeho použití se doporučuje v případech, kdy je třeba zvolit maximálně obecný přístup k prvkům. Neříká nic o tom, zda se prvky v kolekci mohou vyskytovat duplicitně nebo jsou nějakým způsobem uspořádány.

```

public interface Collection<E> extends Iterable<E> {
    int size();                                //počet prvků
    boolean isEmpty();                         //je kolekce prázdná?
    boolean contains(Object element);          //obsahuje kolekce element?
    boolean add(E element);                   //přidej element do kolekce
    boolean remove(Object element);            //vymaž z kolekce element
    Iterator<E> iterator();                  //iterátor kolekce

    boolean containsAll(Collection<?> c);    //obsahuje kolekce kolekci c?
    boolean addAll(Collection<? Extends E> c); //přidej kolekci c
    boolean removeAll(Collection<?> c);        //vymaž kolekci c
    boolean retainAll(Collection<?> c);         //vymaž vše co není v kolekci c
    void clear();                            //vymaž všechny prvky

    Object[] toArray();                      //převed' na pole objektů
    <T> T[] toArray(T[] a);                //převed' na pole typu T
}

```

Jak již bylo poznamenáno, je rozhraní typu **Collection** navrženo dostatečně obecně, aby bylo možné použít jeho operace, za předpokladu kdy chceme v kolekci uchovávat duplicitní i jedinečné prvky. Tomuto způsobu museli být upraveny i metody jako add, remove, atd.

Procházení kolekcí lze realizovat dvěma způsoby, **konstrukcí for-each nebo iterátory**:

- For-each (**speciální případ cyklu for**) umožňuje **efektivně procházet kolekcemi prvek po prvku**

```

for (Object o : collection) {
    System.out.println(o);
}

```

- Iterátory umožňují procházet kolekcí a v případě potřeby z ní prvky vybírat, metoda hasNext() vrací true pokud existuje další položka, metoda next() vrací další položku a remove...:

```

for (Iterator<?> it = c.iterator() : it.hasNext(); ) {
    System.out.println(it.next());
    it.remove();
}

```

6.2.2 Množina - Set

Kolekce typu **Set** **nemohou obsahovat duplicitní prvky**, jedná se o analogii s matematickou množinou. Speciálním případem tohoto rozhraní je rozhraní **SortedSet**, které udržuje prvky **seřazené ve vzestupném pořadí**. Rozhraní Set obsahuje pouze prvky zděděné z Collection a přidává omezení, aby nebylo možné používat duplicitní prvky.

Jelikož rozhraní typu Set představuje analogii s matematickou množinou, je vhodné ukázat i analogii množinových operací s metodami rozhraní. Metoda `c1.containsAll(c2)` vrací true, pokud je množina `c2` podmnožinou množiny `c1`. Metoda `c1.addAll(c2)` představuje sjednocení množin, podobně jako metoda `c1.retainAll(c2)` představuje průnik obou množin. Nakonec zbývá metoda `c1.removeAll(c2)`, která převede množinu `c1` na asymetrickou množinovou differenci obou množin.

Obecné implementace

Obecné implementacemi rozhraní Set jsou **HashSet** (ukládá prvky do hashovací tabulky, nezaručuje však jejich pořadí), **TreeSet** (ukládá prvky do černobílého stromu, řadí tedy prvky podle jejich hodnot – je mnohem pomalejší než HashSet) a **LinkedHashSet** (ukládá prvky do hashovací tabulky, kterou prochází propojený seznam (LinkedList), prvky jsou uloženy podle pořadí vložení).

Speciální implementace

Dvěma speciálními implementacemi rozhraní Set jsou **EnumSet** a **CopyOnWriteArrayList**. První z nich je velmi výkonná implementace pro výčtové typy, jejíž vnitřní reprezentaci zajišťuje bitový vektor. Druhá implementace je založena na poli s kopírováním při zápisu. Všechny proměnné operace (`add()`, `set()` a `remove()`) jsou implementovány vytvořením nové kopie pole. V žádném případě není nezbytné aplikovat uzamčení, každý průchod polem může souběžně aplikovat vkládání a mazání prvků. Implementace je vhodná pro kolekce, které jsou málo kdy upravovány a často dochází k procházení.

6.2.3 Seznam - List

Jde o uspořádanou kolekci (sekvenci) prvků, která může obsahovat duplicitní prvky. Rozhraní typu List krom zděděných operací zahrnuje také operace, které je možné rozdělit do tří skupin. Za prvé jsou to operace pozičního přístupu, které operují s prvky seznamu na základě číselného indexu jejich pozice v tomto seznamu. Za druhé jsou to operace hledání, které vyhledají prvek a vrátí jeho index v seznamu. A za třetí jsou to iterace, rozšiřující objekt Iterátor.

Metody pozičního přístupu – hlavními metodami jsou `get` a `set` pro čtení a zápis prvku na/z určité pozice, dále jsou to metody `add()`, `addAll()` a `remove()`.

Metody hledání – `indexOf()` a `lastIndexOf()` vrací index pořadí požadovaného prvku

Metody iterací – `listIterator()`, která je dvakrát přetížena, více viz podkapitola [Iterátory](#)

Další algoritmy rozhraní List

- `subList(int fromIndex, int toIndex)` – vrací podmnožinu z kolekce, jde pouze o zobrazení, tzn., že metody provedené v podmnožině se promítnou i do původní kolekce (netvoří se kopie)
- `sort()` – seřadí kolekci algoritmem „merge sort“
- `shuffle()` – náhodně přeskopí prvky
- `reverse()` – obrátí pořadí prvků
- `rotate()` – rotuje prvky o určitý počet kroků
- `swap()` – zamění prvky na určitých pozicích

- `replaceAll()` – nahradí všechny výskyty nějakým prvkem
- `fill()` – přepíše všechny prvky v kolekci konkrétní hodnotou
- `copy()`, `binarySearch()`, `indexOfSubList()`, `lastIndexOfSubList()`

Obecné implementace

Dvěma obecnějšími implementacemi tohoto rozhraní jsou implementace typu **ArrayList**, která je obvykle rychlejší (nabízí poziční přístup v konstantním čase) než druhá implementace typu **LinkedList**. Také je vhodné se na tomto místě zmínit o implementaci **Vektor**, která byla zpětně přizpůsobena tak, aby implementovala v novém Collection Frameworku rozhraní List. Příbuznost mezi ArrayListem a Vectorem je v tom, že Vektor je ve své podstatě ArrayListem, který navíc obsahuje režii spojenou se synchronizací kolekce.

Speciální implementace

Implementace **CopyOnWriteArrayList** je založena podobně jako CopyOnWriteArrayList na poli s kopírováním při zápisu. Není nutná žádná synchronizace a u iterátorů je zaručeno, že nikdy nezpůsobí výjimku, dokonce ani při zápisu prvku do kolekce. Pozn. V jiných případech by šlo o výjimku ConcurrentModificationException. Kolekce je vhodná k tvorbě seznamů zřídka se měnících.

6.2.4 **Fronta - Queue**

Kromě základních operací rozhraní Collection obsahuje ještě speciální metody, speciálně upraveny pro frontu (viz Obr. 78). Fronta nabízí dvě možnosti jak operovat s jejími prvky. Bud' je možné použít metody, které v případě neúspěchu vyvolají výjimku, nebo je možné použít operace, které i v případě neúspěchu vrátí jistou hodnotu a to null nebo false.

	Vrácení hodnoty	Způsobení výjimky
Vložení prvku	<code>offer(prvek)</code>	<code>add(prvek)</code>
Odebrání prvku	<code>poll()</code>	<code>remove()</code>
Vybrání prvku	<code>peek()</code>	<code>element()</code>

Obr. 78: Možnosti práce s prvky v rozhraní typu Queue

Implementace fronty je zpravidla provedena jako **FIFO** (first in, first out). Dále může být použita **prioritní fronta**, která řadí prvky podle jejich hodnot, ale v tomto případě musí být specifikováno jak prvky řadit. Dále mohou být implementace provedeny také jako limitované (bounded) fronty, které umožní uložit jen omezený počet prvků. Zajímavostí je, že rozhraní **LinkedList** bylo zpětně upraveno tak, že neimplementuje pouze rozhraní List, ale i rozhraní Queue. Jedním z důvodů je to, že implementace typu Queue nemohou ukládat prvky s hodnotou null, a v tomto případě právě přichází na řadu implementace LinkedList, která toto dovoluje. Tato možnost je však přístupná pouze z historických důvodů a neměla by být využívána, neboť hodnota null slouží jako návratová hodnota metod `poll()` a `peek()` a nikoli jako hodnota, která by měla být uchovávána v kolekci.

Obecná implementace

Obecnou implementací, která mimo jiné implementuje rozhraní Queue je **LinkedList**. **LinkedList** se v tomto kontextu chová jako fronta FIFO. Dalším zástupcem je třída **PriorityQueue**, která představuje, jak název napovídá, prioritní frontu založenou na datové struktuře halda (heap). Tato fronta řadí prvky podle pořadí určeného během vytváření kolekce.

Speciální implementace

Mezi speciální implementace fronty patří **LinkedBlockingQueue** – limitovaná fronta FIFO s blokováním, založena na propojených uzlech. **ArrayBlockingQueue** – limitovaná fronta FIFO s blokováním, založena na poli. **PriorityBlockingQueue** – nelimitovaná fronta s blokováním, založena na hladě, **DelayQueue** – fronta s časově založeným plánováním, založena na hladě a **SynchronousQueue**. Tyto fronty jsou založeny na obecnějším rozhraní BlockingQueue, které rozšiřuje rozhraní Queue o operace, které před načtením prvku čekají, než je do fronty umístěn alespoň jeden prvek a před uložením prvku čekají, až se ve frontě uvolní místo.

6.2.5 Mapování

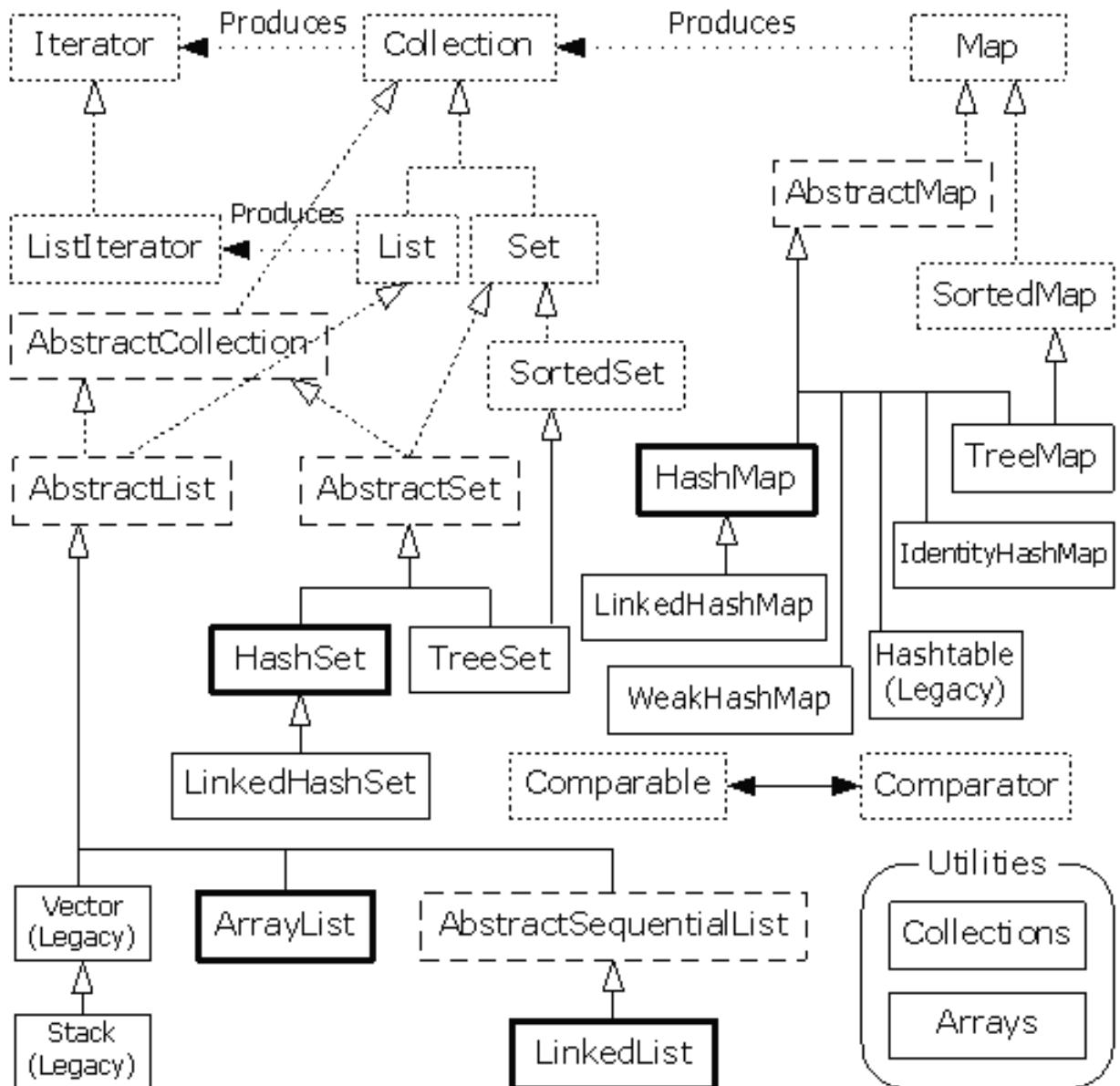
Kolekce typu Map je taková kolekce, jež mapuje klíče na hodnoty. **Mapa nemůže obsahovat duplicitní klíče a každý klíč je možné mapovat pouze na jednu hodnotu**. Speciálním případem tohoto rozhraní je rozhraní typu **SortedMap**, které udržuje vzestupné pořadí klíčů. Uspořádaná verze se hodí pro kolekce, **jako jsou slovníky, adresáše atd.**

Obecné implementace

Java Collection Framework obsahuje tři implementace tohoto rozhraní: **HashMap**, **TreeMap** a **LinkedHashMap**. Starší implementace **Hashtable** musela být upravena, a je právě jednou z implementací Mapy. Základní metody rozhraní Map jsou put, get, containsKey, containsValue, size a isEmpty a chovají se zcela stejně jako ve starším rozhraní Hashtable.

Speciální implementace

Rozhraní Map implementují tři speciální třídy. První z nich je **EnumMap**, která je založena na struktuře pole. Představuje vynikající implementaci použitelnou s výčtovými typy podobně jako EnumSet. Tato implementace je kombinací rychlosti polí a všeobecnosti rozhraní Map. Pokud máme v úmyslu mapovat výčtové typy na hodnoty, měla by být zvolena tato implementace. Další třídou je **WeakHashMap**, která ukládá pouze slabé odkazy na své klíče. Tato třída je vhodnou implementací datových struktur, kde klíč je bezcenný, pokud není nadále dostupný pro žádny podproces. Poslední implementací je třída **IdentityHashMap**, která je založena na hashovací tabulce. Tato třída je vhodná pro uložení objektů při serializaci či vytváření hlubokých kopií.



Obr. 6: Nástin implementace Java Collection Frameworku, tečkovaně jsou zobrazena rozhraní, čárkovaně abstraktní třídy, plnou čarou jsou zobrazeny implementační třídy, tučnou plnou čarou jsou zobrazeny čtyři nejběžnější a nejdůležitější implementační třídy a jako Utilities jsou představeny třídy Collections a Arrays, které jsou statické a poskytují hromadné a konverzní operace nad poli a kolekcemi, převzato z [5]

6.2.6 Iterátory

Iterátor je speciální typ objektu, který umožňuje procházení kolejek. Oproti procházení pomocí cyklu `for-each` je možné pomocí iterátoru během procházení kolejce prvky i odebírat. Iterátory mají v podstatě podobnou funkci, jako měla proměnná typu ACT, jenž jsme probírali v minulých přednáškách.

Speciálním případem iterátoru je **ListIterator**, který rozšiřuje rozhraní Iterátor a je použit v kolejích typu List. Tento speciální iterátor umožňuje v rozhraní List obousměrné procházení, tzn., že kromě metod `next()`, jsou obdobně implementovány i metody `previous()`.

Častou otázkou je, jak iterátory vlastně fungují. Stručně řečeno, kurzor iterátoru se nachází vždy mezi dvěma sousedními prvky (prvkem, který by vrátilo volání metody next() a prvkem který by vrátilo volání metody previous()). Stejně tak platí, že po volání metod previous() a next() resp. next() a previous() je vždy vrácen stejný prvek. Dále je vhodné se na tomto místě zmínit o metodách nextIndex() a previousIndex(), které vracejí hodnoty indexů prvků. Pokud bude kurzor umístěn na začátku kolekce a bude volána metoda previousIndex() bude vrácena hodnota -1. Pokud bude kurzor umístěn na konci kolekce a bude volána metoda nextIndex() bude vrácena hodnota kolekce.size().

6.2.7 Řazení objektů – Rozhraní Comparable

Implementací rozhraní Comparable můžeme jednoduše docílit, že námi implementované objekty budou porovnatelné resp. seřaditelné. Toto rozhraní je automaticky implementováno již v některých základních třídách Java Collection Frameworku. Mezi nejvýznamnější třídy, které toto rozhraní implementují, patří třídy představující číselné typy, ať již celočíselné nebo reálné, třída String, Boolean (FALSE < TRUE), Date a další. Pokud se pokusíme seřadit seznam prvků (např. pomocí Collections.sort(...)) a daná třída nebude rozhraní implementovat, dojde k výjimce ClassCastException.

Příklad [3]:

```
public class Employee implements Comparable {

    int employeeID;
    String employeeName;
    double salary;
    static int i;

    public Employee() {
        employeeID = i++;
        employeeName = "dont know";
        salary = 0.0;
    }

    public Employee(String employeeName, double salary) {
        this.employeeID = i++;
        this.employeeName = employeeName;
        this.salary = salary;
    }

    public String toString() {
        return "ID " + this.employeeID + "\n" + "Salary " + this.salary;
    }

    public int compareTo(Object o1) {
        if (this.salary == ((Employee) o1).salary)
            return 0;
        else if ((this.salary) > ((Employee) o1).salary)
            return 1;
        else
            return -1;
    }
}
```

```
        return -1;
    }
}
```

```
import java.util.*;

public class ComparableDemo{

    public static void main(String[] args) {

        List ts1 = new ArrayList();

        ts1.add(new Employee ("Tom",40000.00));
        ts1.add(new Employee ("Harry",20000.00));
        ts1.add(new Employee ("Maggie",50000.00));
        ts1.add(new Employee ("Chris",70000.00));

        Collections.sort(ts1);
        Iterator itr = ts1.iterator();

        while(itr.hasNext()){
            Object element = itr.next();
            System.out.println(element + "\n");

        }

    }
}
```

Výstup programu:

```
EmpID 1
Sal20000.0
```

```
EmpID 0
Sal40000.0
```

```
EmpID 2
Sal50000.0
```

```
EmpID 3
Sal70000.0
```

Z příkladu je patrné, že třída implementuje metodu compareTo() předepsanou rozhraním Comparable. Metoda má návratový typ integer, vracející záporné celé číslo, nulu nebo kladné celé číslo. Návratová hodnota závisí na tom, zda je porovnávaný objekt menší, roven nebo větší objektu porovnávanému. Pozn. Porovnávaný objekt je ten, který je do metody poslán jako její parametr.

6.2.8 Řazení objektů – Komparátory

Komparátory použijeme, pokud chceme porovnávat objekty podle jiného řazení, než je řazení implementované pomocí rozhraní Comparable. Rozhraní Comparator pracuje podobně jako rozhraní Comparable, obsahuje jedinou metodu compare(o1, o2), jejíž návratová hodnota je vrácena na přesně stejném principu jako u rozhraní Comparable a pokud vznikne problém, je vrácena i stejná výjimka. Implementaci komparátoru bude nejlepší vysvětlit na následujícím příkladě.

```
public class Employee {

    /*
        stejná implementace jako v předchozím případě,
        krom metody compareTo()
    */

    static final Comparator<Employee> EMP_ORDER =
        new Comparator<Employee> {

            public int compare(Employee e1, Employee e2) {

                return e2.salary.compareTo(e1.salary());
            }
        }
}
```

```
import java.util.*;

public class ComparableDemo{

    public static void main(String[] args) {

        ...
        Collections.sort(tsl,
                        EMP_ORDER);
        ...
    }
}
```

Jelikož se změnilo to, jak budou objekty porovnávány, musí být změněna i metoda Main(), která toto porovnávání zajišťuje voláním metody sort(). Výstup pak bude záviset na velikosti platu zaměstnance, protože přesně takto je definováno porovnávání v komparátoru.

6.3 Ukázky implementace

Příklad 1: Převod kolekce na pole

Mějme c a předpokládejme, že jde o typ Collection, převedeme je na pole Objektů:

```
Object[] a = c.toArray()
```

Předpokládejme, že c obsahuje objekty typu String (délka pole bude stejná jako počet elementů c):

```
String[] a = c.toArray(new String[0])
```

Příklad 2: Odstranění duplicit v kolekci

Mějme kolekci c typu Collection, a chceme odstranit duplicitní prvky v ní obsažené:

```
Collection<Type> noDups = new HashSet<Type>(c)
```

Nebo chceme odstranit duplicitní prvky a zároveň prvky seřadit:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c)
```

Příklad 3: Implementace Mapy

```
import java.util.*;
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

Mějme výše uvedený program, který zjišťuje počet výskytů jednotlivých slov, přivedených na vstup tomuto programu. Pokusme se nyní implementovat tento program za pomocí všech tří základních implementací rozhraní Map, tedy HashMap, TreeMap a LinkedHashMap.

Aplikaci spustíme příkazem:

```
java Freq if it is to be it is up to me to delegate
```

Výstup aplikace, při použití HashMap:

```
8 distinct words (neseřazeno):
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Výstup aplikace, při použití TreeMap (abecedně seřazeno):

```
8 distinct words:  
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Výstup aplikace, při použití LinkedHashMap (seřazeno podle prvního výskytu slova):

```
8 distinct words:  
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

6.4 Literatura

- [1] The Performance of Java's Lists, O'Reilly ONJava.com
[<http://www.onjava.com/pub/a/onjava/2001/05/30/optimization.html>](http://www.onjava.com/pub/a/onjava/2001/05/30/optimization.html)
- [2] Java™ Platform Standard Ed. 6, <http://java.sun.com/javase/6/docs/api/>
- [3] JAVA TIPS, <http://www.java-tips.org/java-se-tips/java.lang/how-to-use-comparable-interface.html>
- [4] ZAKHOUR, Sharon. *Java : Výukový kurz*. Vydání první. Brno : Computer Press, a.s., 2007. 534 s. ISBN 9788025115756.
- [5] UMD, <http://www.isr.umd.edu/~austin/ence200.d/notes09b.html>

7 Teorie grafů

7.1 Grafy

Teorie grafů představuje dnes již samostatně rozvinutou matematickou disciplínu, jejíž aplikace nacházíme v řadě oblastí. Existuje k ní dostatek literatury a to i v českém jazyce [1], [2]. Graf v matematice je nejčastěji používané **znázornění funkční závislosti**. Na druhou stranu, **grafem** v teorii grafů se rozumí objekty (obecně, nikoli jen z pohledu objektově orientovaného programování)

Grafy jsou struktury, které jsou tvořeny vrcholy a hranami, které tyto vrcholy vzájemně spojují. Graf se obvykle znázorňuje jako množina bodů spojených čarami.

Definice: Graf G je uspořádaná dvojice $G = (V, E)$, kde V je konečná množina vrcholů a množiny E je konečná množina hran:

$$G = (V, E), \text{ kde } V \cap E = \emptyset \quad (1)$$

$$E \subseteq V \times V \quad (2)$$

Prvky množiny nazýváme vrcholy (vertex) nebo často také uzly.

Definice: Mějme vrcholy $x, y \in V$. Tyto vrcholy nazýváme sousedními vrcholy, pokud $(x, y) \in E$.

Všimněte si, že na základě definice nelze, ab mezi dvěma vrcholy existovala více než jedna hrana (tzv. násobné hrany).

Rozlišujeme dva typy grafů. **Orientované a neorientované**. Pojem neorientovaného grafu je velice blízko pojmu symetrická relace. V případě neorientovaného grafu musí pro každé dva prvky $x, y \in V$ platit, že pro $\forall (x, y) \in E: (y, x) \in E$.

Grafy jsou nejobecnější strukturou a do jejich podmnožiny spadají všechny ostatní datové struktury. Dají se s jejich pomocí reprezentovat pole, lineární seznamy, stromy i tabulky. Grafy mají bohaté zastoupení v rozmanitých oborech jak počítačových tak i nepočítačových. Dají se pomocí nich reprezentovat znalosti pro umělou inteligenci, počítačové sítě nebo kupříkladu vzájemné reference odborných článků. **Grafy mají obecně největší vyjadřovací sílu pro reprezentaci informace v paměti.** Jejich nevýhodou je oproti lineárním či binárních strukturám výrazně vyšší nároky na paměťovou expanzi a časovou expanzi.

Struktura grafu může být **rozšířena o ohodnocení hran** (označováno jako váha) a její význam může být různý, např. vzdálenost mezi městy, počet „hopů“ v počítačových sítích, propustnost atd.). **Výsledkem je model reálné sítě.** Takové modely se používají pro analýzu dopravy nebo počítačových sítí (jako např. internetu).

$$G = (V, E, W) \quad (3)$$

$$W: E \rightarrow \mathbf{Z} \quad (4)$$

Další variantou je ohodnocení vrcholů.

$$G = (V, E, M) \quad (5)$$

$$M: V \rightarrow \mathbf{Z} \quad (6)$$

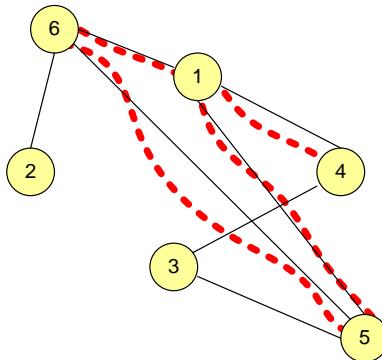
Popřípadě se může jednat o kombinaci předchozích dvou variant, kde jsou ohodnoceny jak hrany grafu, tak vrcholy grafu:

$$G = (V, E, M, W) \quad (7)$$

7.1.1 Základní pojmy

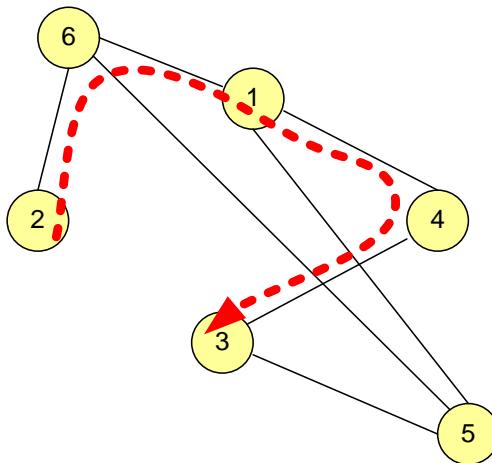
Stupněm vrcholu v v grafu G rozumíme počet hran incidentních (tj. vycházejících) z v . Stupeň vrcholu v v grafu G značíme $d_G(v)$. U orientovaných grafů je stupeň definován pomocí uspořádané dvojice, kde první hodnotou je počet vrcholů do vrcholu vstupujících a druhou počet vrcholů z uzlu vystupujících.

Sled je posloupnost vrcholů a hran, začínající a končící v daném vrcholu. Tyto vrcholy musí být navzájem spojeny hranami a mohou se opakovat. **Délka sledu** je rovna počtu hran ve sledu (v případě ohodnoceného grafu je rovna součtu ohodnocení hran obsažených ve sledu). Příklad sledu je na Obr. 79.



Obr. 79: Příklad sledu, kde posloupnost vrcholů je 1, 6, 5, 1, 4. Vrchol 1 se v posloupnosti vyskytuje 2x.

Tah - orientovaný (neorientovaný) sled, v němž se žádná hrana neopakuje, se nazývá orientovaný (neorientovaný) tah. V případě sledu je možné, aby se vrcholy resp. i hrany opakovaly.



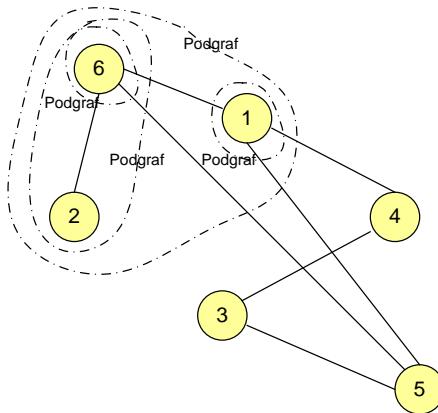
Obr. 80: Příklad tahu, kde posloupnost vrcholů je 2, 6, 1, 4, 3. Žádný z vrcholů se neopakuje.

Cesta P_n - orientovaný (resp. neorientovaný) sled, v němž se neopakuje žádný vrchol, se nazývá orientovaná (resp. neorientovaná) cesta. **Cesta P_n** na n vrcholech je definována jako graf: $G = (V, E)$, kde $V = \{v_1, v_2, \dots, v_n\}$ a $E = \{(v_i, v_{i+1}): 1 \leq i < n\}$.

Nejvyšší stupeň v grafu G značíme $\Delta(G)$ a naopak **nejnižší stupeň** grafu značíme $\delta(G)$.

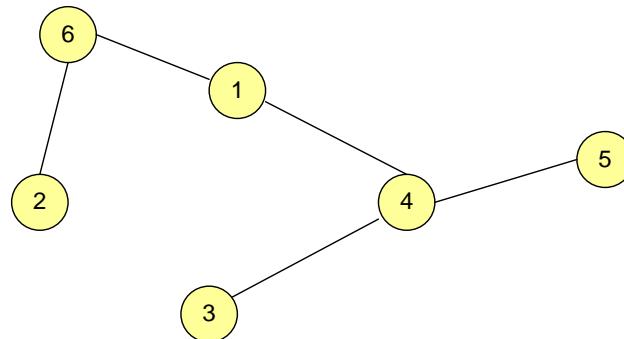
Vzdálenost $d_G(u, v)$ dvou vrcholů u, v v grafu G je dána délkou **nejkratšího počtu hran** mezi u a v v G . Pokud sled mezi u, v neexistuje, je vzdálenost $d_G(u, v) = \infty$.

Podgraf je graf tvořený některými nebo i všemi vrcholy původního grafu a některými (nebo i všemi) z hran původního grafu, které jsou definovány mezi vrcholy podgrafu (viz Obr. 81 pro příklady grafu a podgrafů).



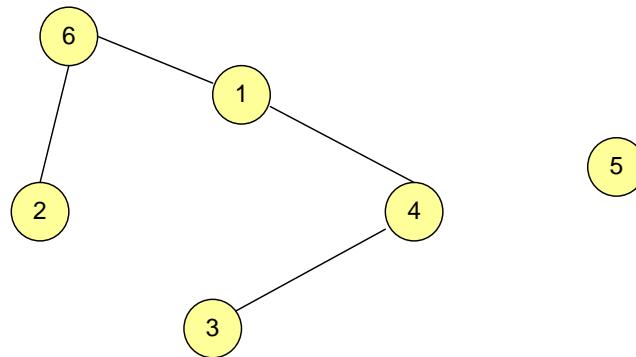
Obr. 81: Graf a příklad podgrafů.

Souvislý graf je takový graf, kde mezi každými dvěma navzájem různými vrcholy x a y existuje cesta, která má x za počáteční a y za koncový vrchol. Jinými slovy, kde z každého vrcholu existuje cesta do každého ze zbývajících vrcholů grafu.



Obr. 82: Souvislý graf

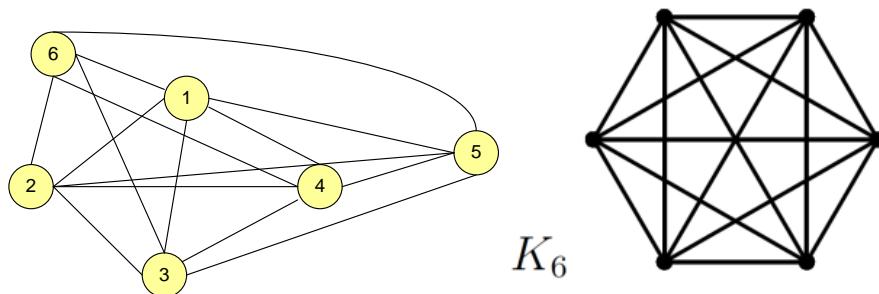
Nesouvislý graf je opakem souvislého grafu. V grafu existují nejméně dva vrcholy, pro které neexistuje cesta.



Obr. 83: Nesouvislý graf

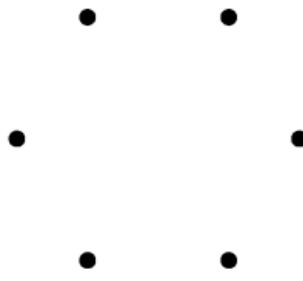
Úplný graf je takový graf, ve kterém existuje hrana mezi každou dvojicí vrcholů grafu. Každý úplný graf je současně souvislý.

Definice: Úplný graf na n vrcholech (značí se K_n) obsahuje množina hran všechny neuspořádané dvojice prvků: $E = V \times V$.



Obr. 84: Příklad úplného grafu.

Definice: Diskrétní graf D_n na n vrcholech, je takový graf, jehož množina hran $E = \emptyset$.

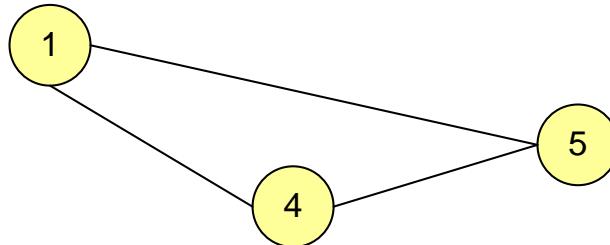


Obr. 85: Diskrétní graf

Kružnice C_n na $n \geq 3$ vrcholech vznikne přidáním jedné hrany do cesty:

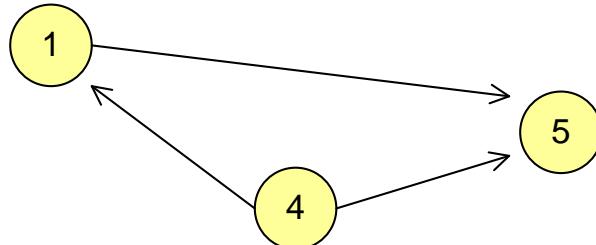
Cyklus je obdobou kružnice u orientovaných grafů. Je to silně souvislý graf, kde z každého vrcholu vychází právě jedna hrana a právě jedna hrana do něj vchází.

Cyklický graf - je orientovaný graf obsahující alespoň jeden cyklus. Příkladem může být dálniční síť nebo letecká dopravní síť.



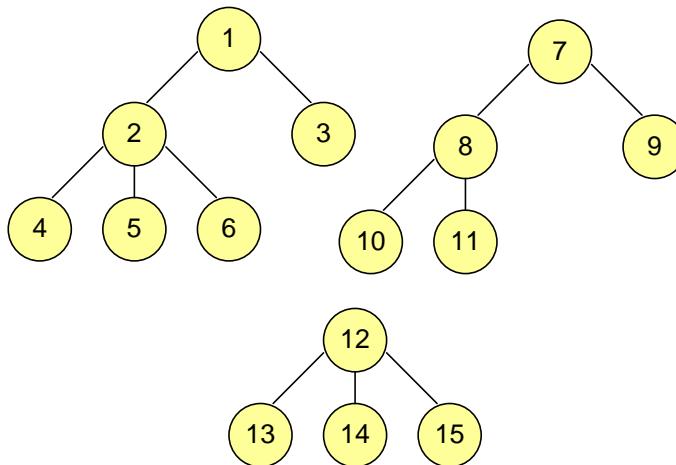
Obr. 86: Cyklický graf

Acyklické grafy - orientovaný graf, kde se nevyskytuje cyklus. Příkladem může být kanalizační síť.



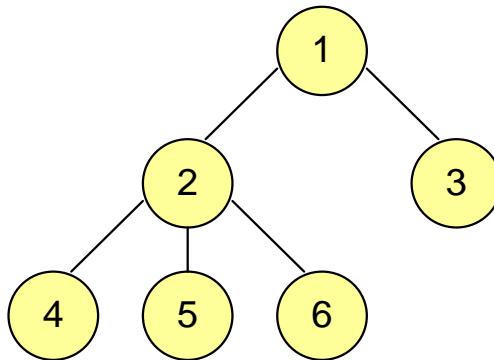
Obr. 87: Příklad acyklického grafu.

Les je jednoduchý graf bez kružnic. Je to nesouvislý neorientovaný graf, jehož komponenty jsou pouze stromy (viz Obr. 88).



Obr. 88: Příklad lesu.

Strom je jednoduchý souvislý graf T bez kružnic. Je to souvislý neorientovaný graf, ve kterém mezi každými dvěma vrcholy existuje právě jedna cesta. Strom tedy neobsahuje jako podgraf kružnici (viz Obr. 89).



Obr. 89: Příklad stromu.

Bipartitní graf - označuje takový graf, jehož množinu vrcholů je možné rozdělit na dvě disjunktní množiny V_1, V_2 ; $V_1 \cap V_2 = \emptyset$. Množina hran je potom $E \subseteq V_1 \times V_2 \cup V_2 \times V_1$, tedy neexistuje hrana, která by spojovala dva prvky ze stejné množiny.

Izomorfní grafy - Grafy, které se liší pouze označením uzlů a hran a způsobem zakreslení (diagramem) jsou označovány za vzájemně izomorfní. Kardinalita vztahu mezi uzly je 1:1 a stejně tak pro hrany.

7.1.2 Vlastnosti ADT graf

Abstraktní datový typ graf je nejkomplexnější datovou strukturou pro reprezentaci informace. Informace může být libovolně vzájemně provázána a je pomocí nich možné modelovat téměř každý model.

Jejich velkou nevýhodou časová i paměťová expanze a z tohoto pohledu i ne příliš dobré výsledky. Tyto vlastnosti se projeví zejména při větších počtech prvků.

7.1.3 Vlastnosti množiny hran

Jak bylo popsáno rovnicí (2), množina hran je relací nad množinou vrcholů. Díky tomu lze popisovat vlastnosti grafů i z pohledu relací. Níže je výčet základních vlastností relací:

- a) Relace R je **reflexivní**: Každý prvek je v relaci sám se sebou.

$$\forall x \in V : R(x, x) \in E$$

- b) Relace R je **symetrická**: Je-li první v relaci s druhým, pak druhý je v relaci s prvním.

$$\forall x, y \in V : R(x, y) \in E \wedge R(y, x) \in E$$

- c) Relace R je **anti-symetrická**: Je-li první v relaci s druhým a druhý je v relaci s prvním, pak první je identický s druhým.

$$\forall x, y \in V : (R(x, y) \in E \wedge R(y, x) \in E) \Rightarrow x = y$$

- d) Relace R je **asymetrická**: Je-li první v relaci s druhým, pak druhý není v relaci s prvním.

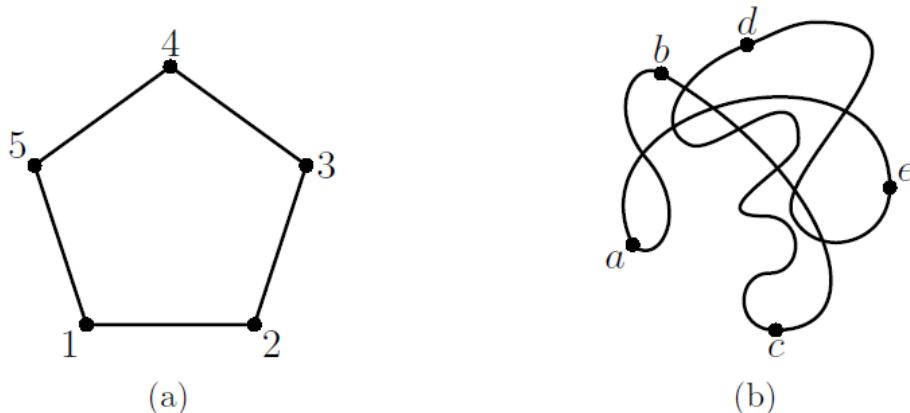
$$\forall x, y \in V : R(x, y) \in E \wedge R(y, x) \notin E$$

- e) Relace R je **transitivní**: Je-li první v relaci s druhým a druhý v relaci s třetím, pak první je v relaci s třetím.

$$\forall x, y, z \in V : (R(x, y) \in E \wedge R(y, z) \in E) \Rightarrow R(x, z)$$

7.1.4 Isomorfismus a podgrafy

Podívejme se na dvojici grafů G, H znázorněných na následujícím obrázku. Jsou to různé grafy. Nejde ani tak o to, že se liší způsob nakreslení | každý graf lze nakreslit mnoha způsoby, a nezáleží na tom, zda jsou čáry představující hrany rovné či zda se kříží.



Obr. 90: Různé ale izomorfní grafy

Grafy se liší už jen tím, že označení vrcholů je jiné:

$$V(G) = \{1, 2, 3, 4, 5\}$$

a

$$V(H) = \{a, b, c, d, e\}$$

Nemůže tedy jít o totožné grafy. Otázkou nyní je, zdali je možné tyto vrcholy přeznačit tak, aby korespondoval s grafem, tj. zdali jsou izomorfní.

Definice: Isomorfismus grafů G a H je zobrazení: $f: V(G) \rightarrow V(H)$, pro kterou platí, že dvojice $\{x, y\}$ je hranou grafu G právě když dvojice $\{f(x), f(y)\}$ je hranou grafu H . Grafy G, H , mezi kterými existuje isomorfismus nazýváme izomorfní (psáno $G \simeq H$).

Grafy na Obr. 90 jsou izomorfní. Stačí uvážit bijekci, která zobrazi prvky 1,2,3,4,5, postupně zobrazi na prvky a, b, c, d, e .

Definice: Graf H je podgrafem grafu G (psáno $H \subset G$), pokud $V(H) \subset V(G)$ a $E(H) \subset E(G)$.

Silnější variantou pojmu podgrafu je pojem indukovaného podgrafa, u kterého vyžadujeme, aby obsahoval všechny hrany, které ve vnitřním grafu na dané množině vrcholů existují.

Definice: Graf H je indukovaným podgrafem grafu G , pokud $V(H) \subset V(G)$ a současně $E(H) = E(G) \cap 2^{V(H)}$. Každá množina $X \subset V(G)$ tedy určuje právě jeden indukovaný podgraf H grafu G takový, že $V(H) = X$. Tomuto podgrafu říkáme indukovaný podgraf na množině X .

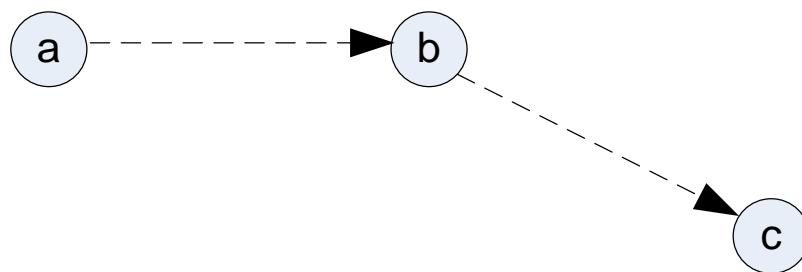
7.1.5 Tranzitivní uzávěr

Tranzitivní uzávěr zahrnuje veškeré relace, které vzniknou „transitivně“, tj. transitivní uzávěr množiny M^* je množina sjednocená s jinou množinou, která obsahuje relace vzniklé přes více relací z množiny.

Příklad:

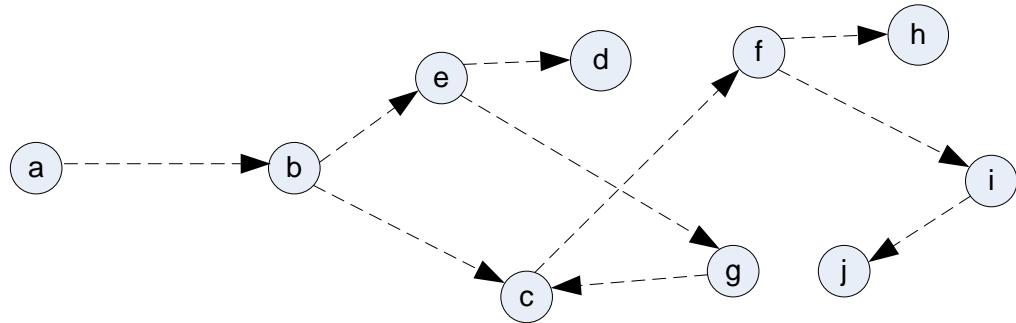
$$R = \{(a, b), (b, c)\}$$

$$\text{Tranzitivní uzávěr množiny } R: R_t = R \cup \{(a, c)\}$$



U složitějších grafů již nemusí být triviální zjistit, zdali existuje cesta z nějakého uzlu. Pokud je relace tranzitivním uzávěrem, jsme schopni v efektivnějším³ čase rozhodnout, zdali je uzel dostupný či nikoli.

³ Závisí na konkrétní implementaci – lineární či logaritmická časová složitost. Odpadne proces vyhledávání uzlu v grafu, který je výrazně náročnější.



Floyd-Warshallův algoritmus

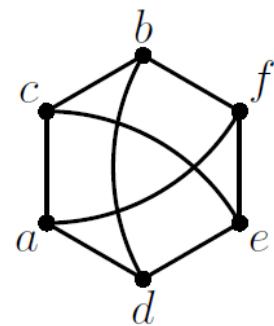
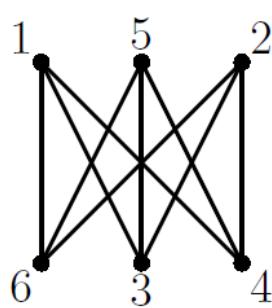
Jedním z **algoritmů pro hledání nejkratší cesty** je Floyd-Warshallův algoritmus. Jeho výhodou je velmi snadné použití. Princip algoritmu spočívá v tom, že se postupně projdou všechny možnosti cest mezi každými dvěma městy a pokud se zjistí, že cesta přes třetí město byla kratší, použije se.

```

1. procedure nejkratsiCesta(var matice:typ_matice);
2. var
3.   i, j, k: integer;
4. begin
5.   for i:=1 to velikost_matice do
6.     begin
7.       for j:=1 to velikost_matice do
8.         begin
9.           for k:=1 to velikost_matice do
10.             begin
11.               matice[i,j]:= minimum(matice[i,j], (matice[i,k] + matice[k,j]));
12.             end; { k }
13.           end; { j }
14.         end; { i }
15.     end;
  
```

7.2 Příklady

- Dokažte, že konečné grafy s různým počtem vrcholů nemohou být izomorfní.
- Dokažte, že je-li graf G izomorfní s grafem H , je současně izomorfní i graf H s grafem G .
- Zapište formálně reprezentaci grafu na obrázku Obr. 90.
- Rozhodněte, zdali jsou grafy izomorfní. Tvrzení dokažte.

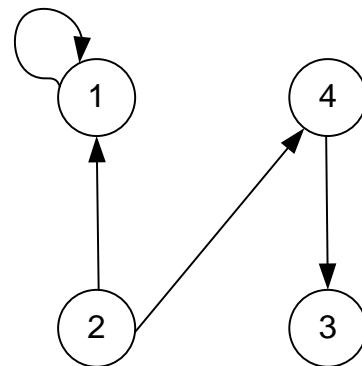


7.3 Způsoby reprezentace ADT graf

7.3.1 Maticí sousednosti

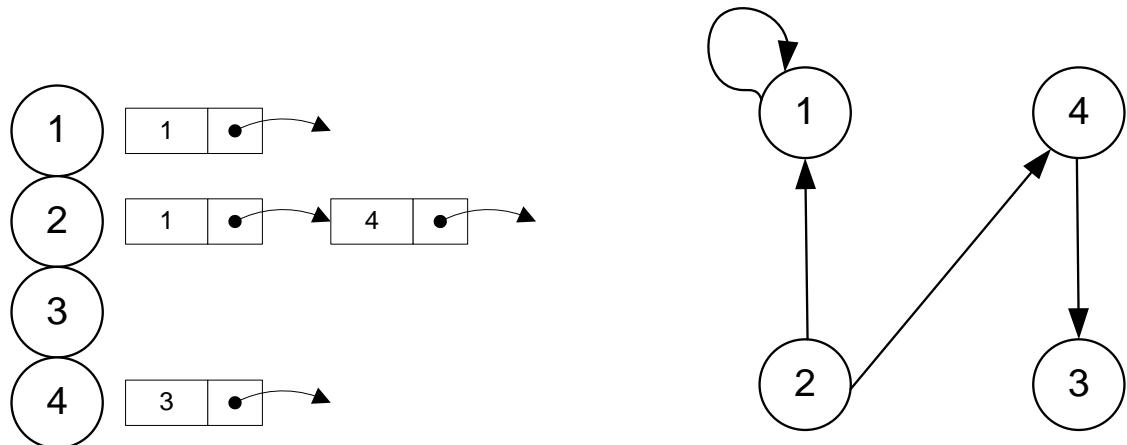
- Vysoké nároky na paměť ($O(n^2)$)
- Rychlý přístup k jednotlivým vazbám
- Není možné dynamicky za běhu programu měnit tyto vazby

	1	2	3	4
1	1	0	0	0
2	1	0	0	1
3	0	0	0	0
4	0	0	1	0



7.3.2 Vektory sousednosti

- Efektivní využití paměti (v paměti existují pouze relace, které existují, neexistující relace nezabírají paměť). Složitost $O(n^2)$, nicméně průměrná složitost Θ je v reálu daleko nižší. Pouze plně propojený graf by využíval plnou celých n^2 paměti.

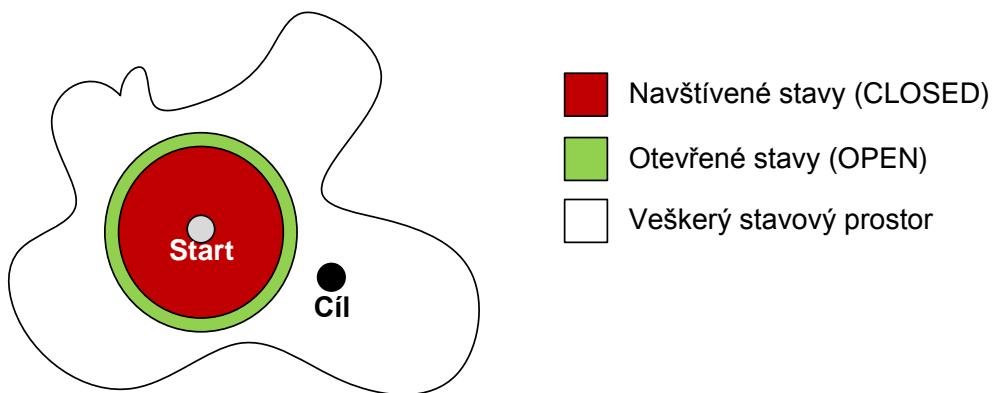


- Pro ověření, zda-li existuje hrana mezi dvěma vrcholy je složitost lineární $O(n)$, protože je nezbytné projí všechny hrany a zkontolovat, zdali existují, či nikoli.

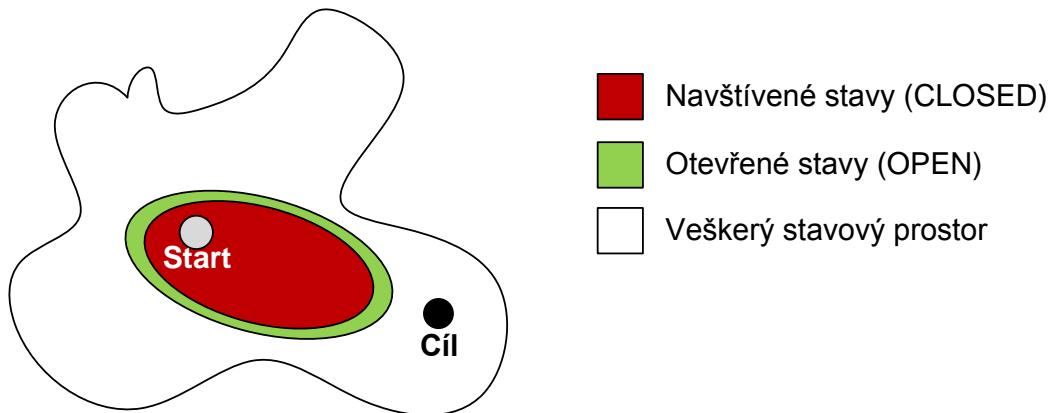
7.4 Průchod a vyhledávání v ADT graf

7.4.1 Úvod

Pro průchody grafy existují 2 skupiny algoritmů. První skupina algoritmů prochází stavový prostor beze snahy odhadnout optimální cestu k nalezení cíle a prochází prostor náhodně. Druhým extrémem je skupina algoritmů využívající informované metody průchodu grafy. V takovém případě naopak dochází k odhadu cesty, která (v ideálním případě) nejoptimálněji povede k cíli. To bude mít za následek menší počet procházených stavů. Na Obr. 91 a Obr. 92 jsou znázorněny trendy, jak je v případě jednotlivých algoritmů prohledáván stavový prostor.



Obr. 91: Způsob prohledávání stavového prostoru neinformovanou metodou, kde se všechny směry expandují stejně rychle.



Obr. 92: Způsob prohledávání stavového prostoru v případě informované metody. Metoda upřednostňuje bližší stavy dříve.

7.4.2 Slepé prohledávání

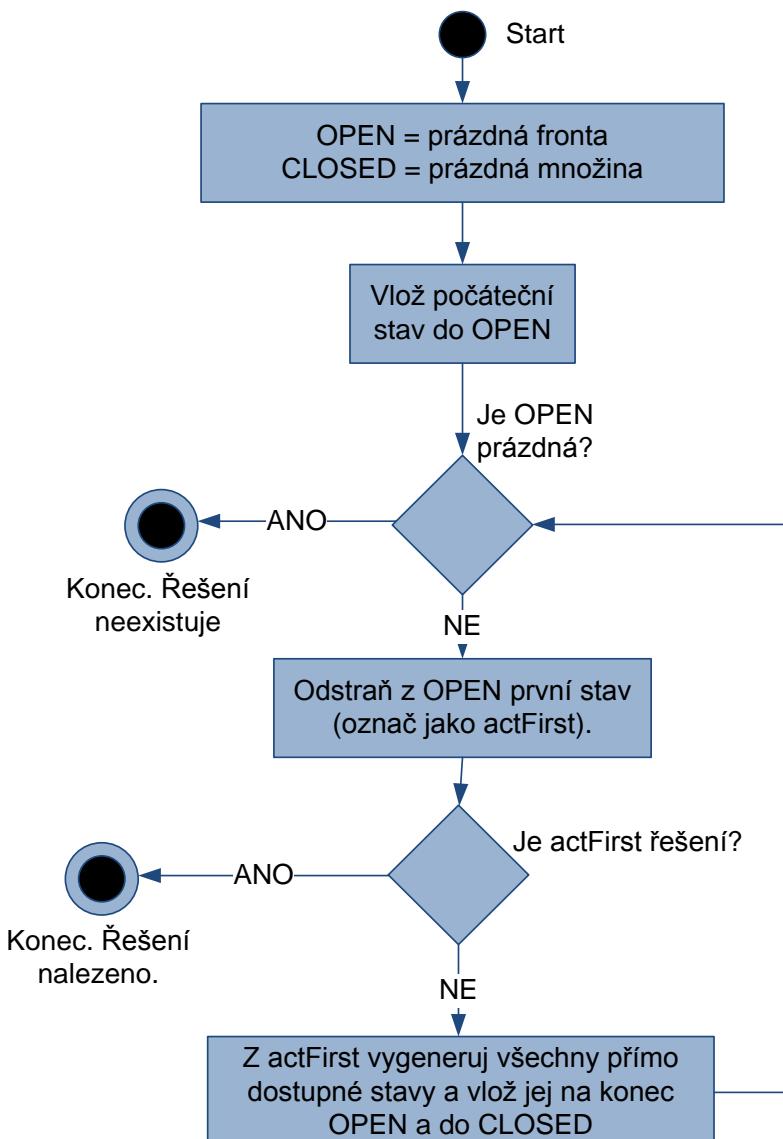
Slepé metody prohledávání fungují na principu postupného procházení stavového prostoru. Neřídí se žádnou prioritou pro pořadí procházení stavů. Výhodou tohoto přístupu je, že nemusíme implementovat žádnou tzv. fitness⁴ funkci.

7.4.2.1 Prohledávání do šířky, BFS

Prohledávání do šířky (Breadth First Search) probíhá, jak už název napovídá, způsobem, kde jsou nejdříve prohledávány stavy blízké počátku a pokud algoritmus nedospěje k řešení, postupně prochází vzdálenější a vzdálenější uzly.

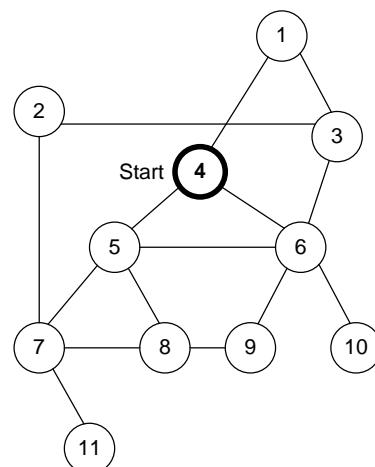
Pseudokód reprezentující BFS algoritmus odpovídá následujícímu diagramu. ADT OPEN představuje posloupnost dosud nenavštívených stavů. Na druhou stranu množina CLOSED představuje množinu těch stavů, které byly prohledány.

⁴ Fitness funkce je funkce, která ohodnotí každý stav průchodu algoritmu nějakou hodnotou. Pokud tuto funkci používáme, je zpravidla snaha zvolit tuto funkci tak, aby vypovídala o tom, jak blízko jsme u řešení.



Obr. 93: Diagram toku v notaci UML.

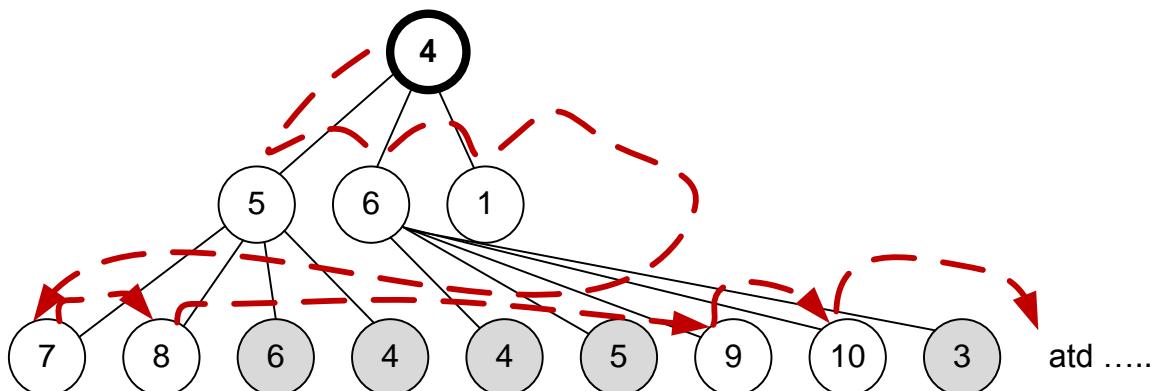
Vezměme v úvahu následující graf, kde je tučně vyznačen počáteční uzel:



Stavový prostor tohoto algoritmu by se poté dal reprezentovat následujícím N-árním stromem. Postup procházení je v protisměru hodinových ručiček, startující z celé hodiny.



Šedé uzly jsou uzly, které již byly projity. V příkladu je použita varianta s kontrolou projitých stavů:

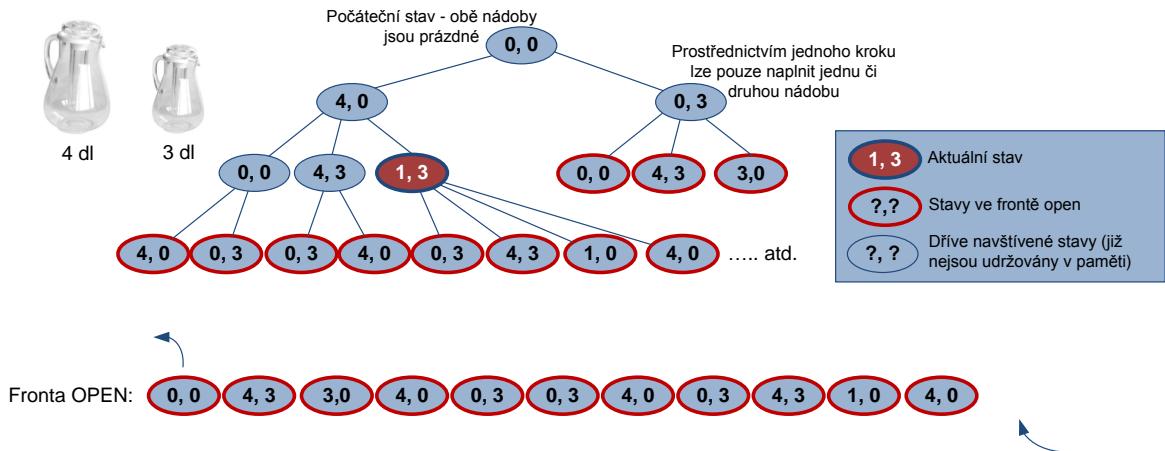


DEMO: [3]

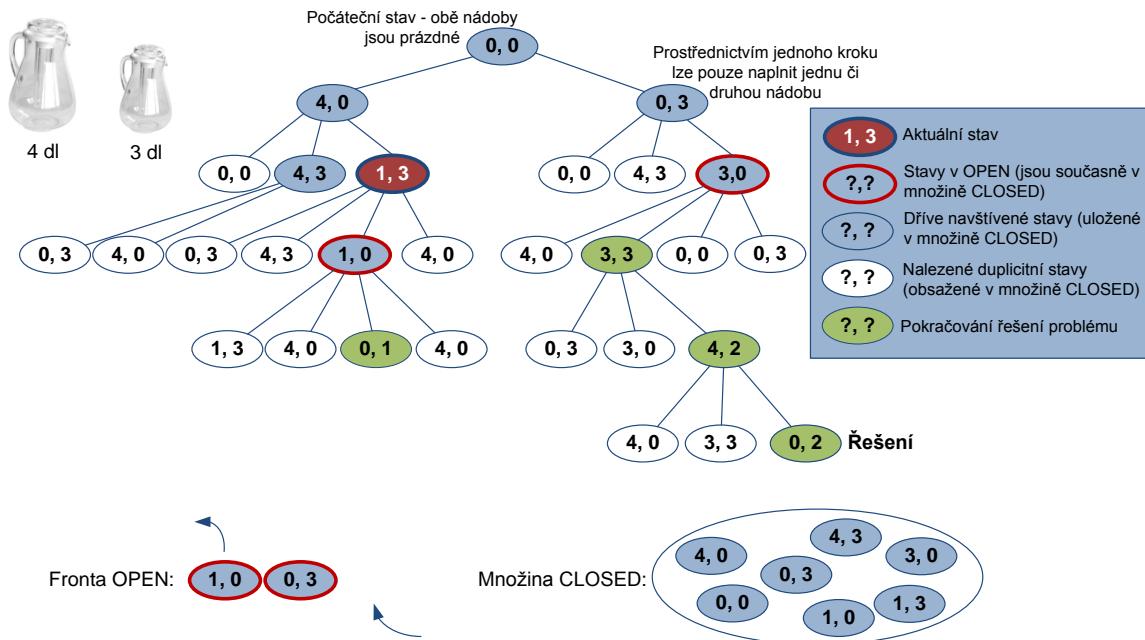
Variantou algoritmu BFS je i varianta bez kontroly opakování průchodu stavů. Taková varianta se hodí jen v případě, že je nemožné popřípadě jen velmi málo pravděpodobné, že nastane cyklus. V takovém případě by algoritmus nemusel konvergovat k řešení vůbec. Pokud si můžeme dovolit odstranit kontrolu navštívených stavů, získáváme časovou úsporu a paměťovou úsporu, která nemusí být bezvýznamná.

Nevýhodou algoritmu je poměrně velký nárok na paměťové prostředky.

Příklad: Mějme dva džbány o objemu 4dl a 3dl. Operace, které jsou nad nimi definovány, jsou nalití po okraj, úplné vylití a přelití. Nalezněte způsob jak odměřit 2dl.



Obr. 95: Algoritmus BFS bez využití množiny CLOSED. Stavy jsou vícekrát navštěvované není zapotřebí udržovat v paměti množinu CLOSED.



Obr. 96: Algoritmus BFS s využitím množiny CLOSED. Opakující se stavy jsou eliminovány, je ale nutné v paměti udržovat veškeré navštěvené stavy.

7.4.2.2 Vlastnosti Algoritmu BFS

Úplnost – algoritmus je úplný, tj. jestliže existuje řešení, BFS jej nalezne. Jedná-li se o nekonečný graf, algoritmus bude konvergovat k řešení (v praxi ale dříve nebo později dojde k vyčerpání paměťových prostředků, které jsou vždy konečné).

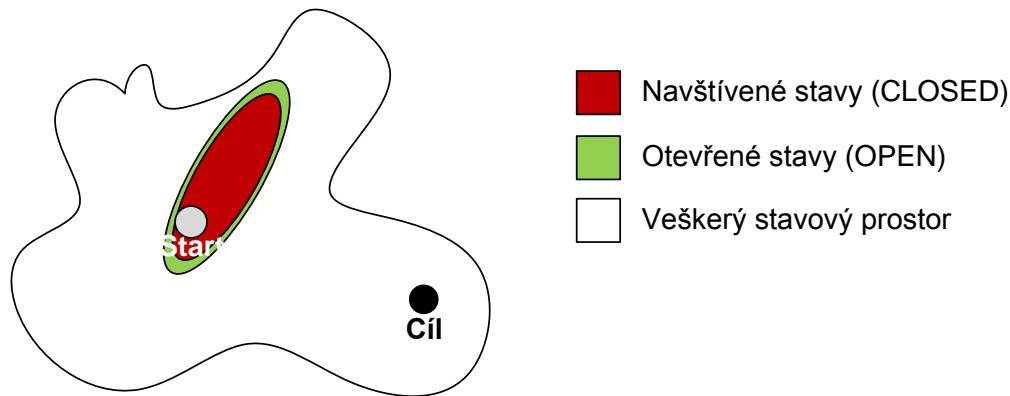
Optimálnost – Algoritmus je optimální, tj. vybere cestu s nejmenším počtem kroků.

Prostorová složitost – B^M , kde B je max. počet větvění, M je maximální hloubka v grafu od počátečního uzlu.

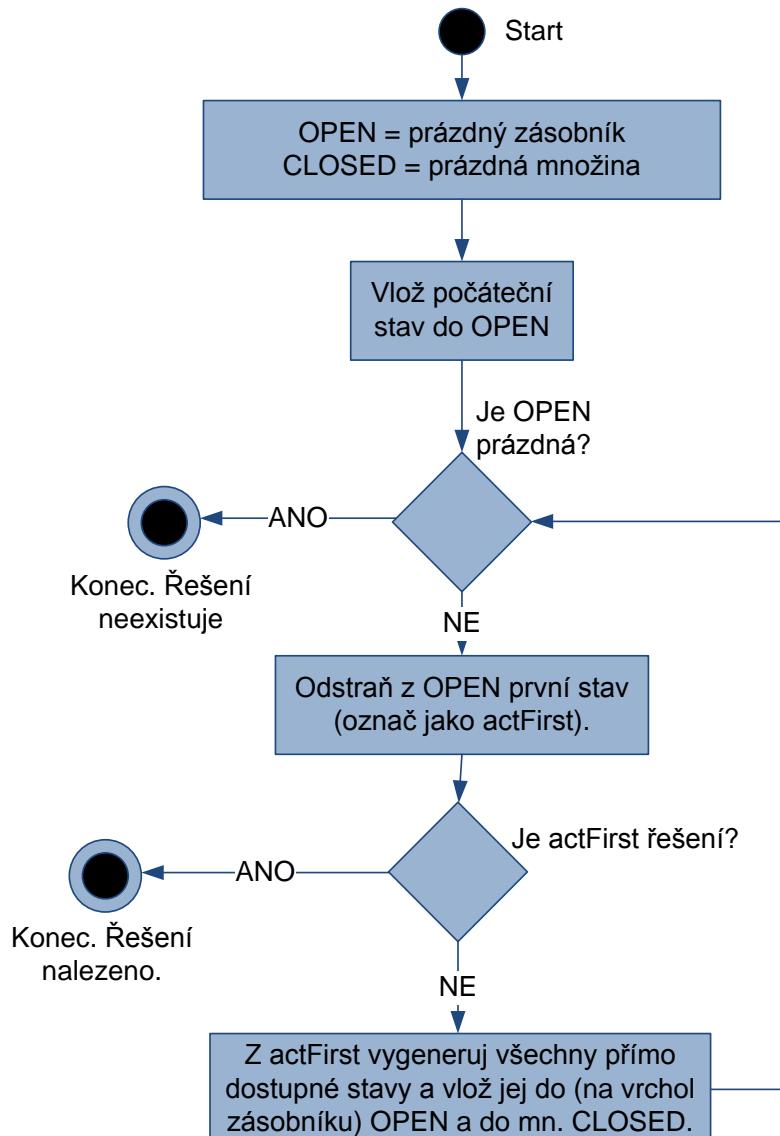
Časová složitost – $O(|V| + |E|)$, kde |V| je počet vrcholů, |E| je počet hran

7.4.2.3 Prohledávání do hloubky, DFS

Prohledávání do hloubky (Depth First Search) probíhá v porovnání s BFS odlišně – jsou naopak upřednostněny uzly, které jsou nejvzdálenější počátku. Pseudokód chování algoritmu je velice podobný předchozímu s drobným rozdílem: **prvky při regenerování prohledávaného stavu nejsou odebírány z počátku, ale z konce seznamu.** Tato drobná změna má i poměrně významný rozdíl v chování a časových charakteristikách:

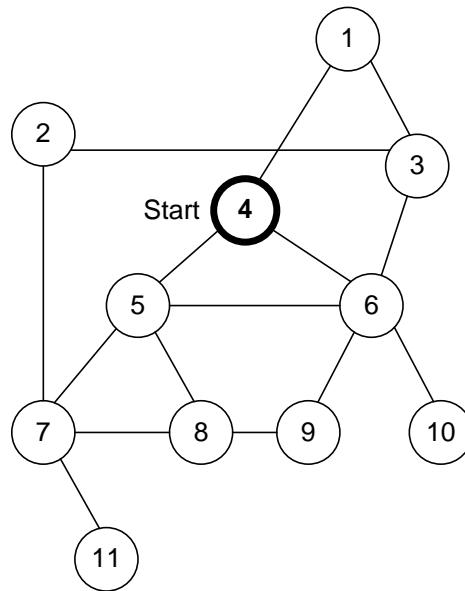


Obr. 97: V případě DFS algoritmu je náhodně zvolen směr, který je prohledáván.

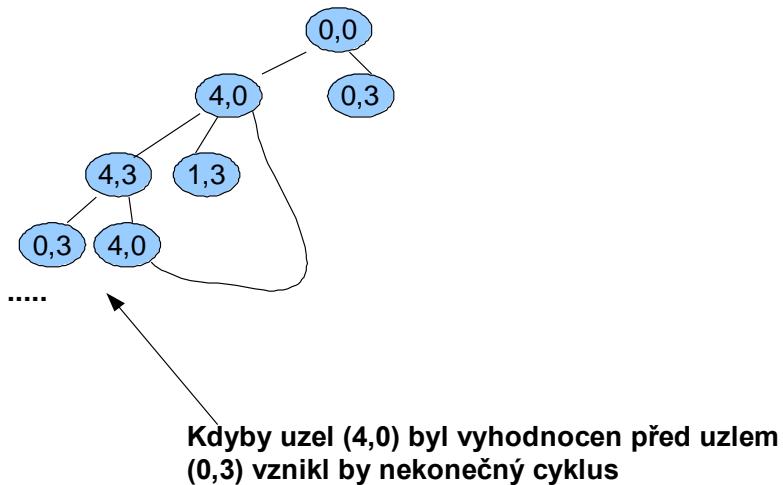


Obr. 98: Diagram procházení toku pro algoritmus procházení do hloubky.

Uvažujme nyní stejný případ jako v předchozím příkladu s BFS a se stejným pořadím procházení uzlů:

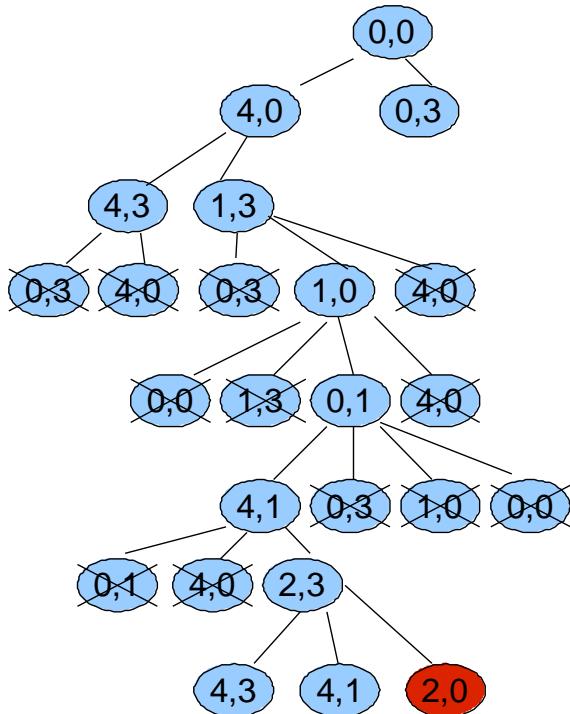


Příklad: Úloha dvou džbánů (viz příklady výše) pomocí algoritmu DFS bez využití množiny closed.



Obr. 99: Algoritmus BFS bez využití množiny closed.

Příklad: Úloha dvou džbánů (viz příklady výše) pomocí algoritmu DFS bez využití množiny closed.



Obr. 100: Algoritmus BFS s využitím množiny closed.

DEMO: [4]

7.4.2.4 Vlastnosti algoritmu DFS

Úplnost – algoritmus je úplný (tj. vždy nalezne řešení, pokud existuje).**Optimálnost** – algoritmus DLS není optimální, upřednostňuje jednu větev oproti druhé. Z toho důvodu je velice pravděpodobné, že bude nalezeno řešení, ke kterému je velký počet kroků i když existuje řešení bližší.**Prostorová složitost** – o mnoho lepší než je BFS – $O(d^*b)$, kde d je hloubka a b je stupeň stromu.**Časová složitost** – obdobně jako DFS. $O(|V| + |E|)$, kde $|V|$ je počet vrcholů, $|E|$ je počet hran

1.1.1. Prohledávání do hloubky s omezenou hloubkou, DLS

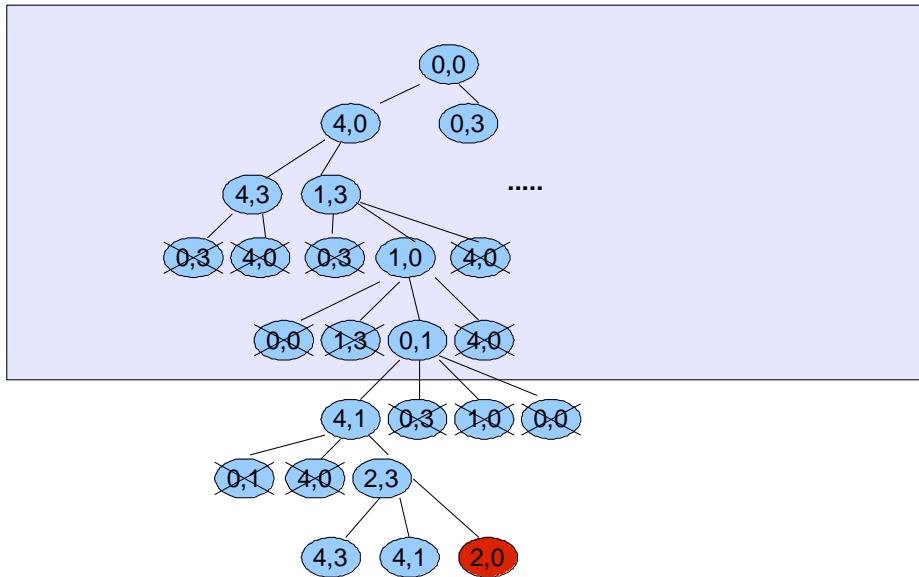
Algoritmus DLS (Depth Limited Search) je speciální variantou DFS, kde je kontrolovaná hloubka zanoření. Varianta je obzvláště výhodná, pokud známe, v jaké hloubce se nachází řešení: (příklad: nalezněte v třetím tahu šach mat).

Dosáhneme tím nízkou spotřebu paměti (množství otevřených stavů je nižší).

```

procedure dfs(v, limit)
  q := ADT_FRONT()
  q.insertFirst(v);
  označ v jako navštívený
  while NOT q.isEmpty()
    v = q.getFirstAndRemove();
    vyhodnot v
    for všechny nenavštívené vrcholy v' přilehlé k v
  
```

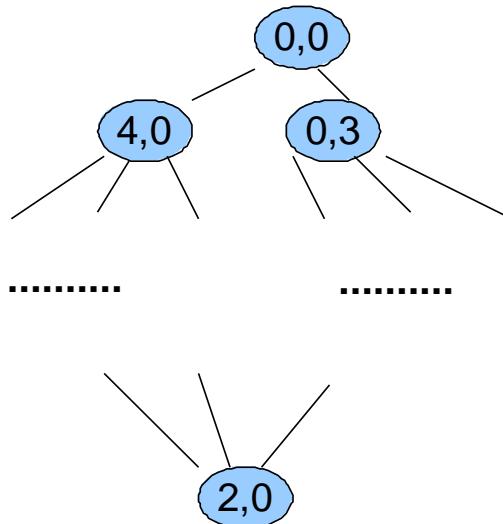
označ v' jako navštívený
 if (hloubka zanoření je < limit) then q.InsertFirst(v');



Obr. 101: Algoritmus DLS, kde vyhledávání probíhá jen v omezeném počtu kroků.

7.4.2.5 Obousměrné vyhledávání (Bidirectional Search, BS)

- Prochází stavový prostor od počátku a současně i od požadovaného stavu a hledá se cesta mezi těmito dvěma stavami.
- Co algoritmus velice zatěžuje, je nutnost kontroly ekvivalence stavů v jednotlivých krocích. Díky tomu je časová složitost algoritmu ne příliš příznivá.
- A* metoda je často lepší volbou.



Obr. 102: Obousměrné vyhledávání, kde známe počáteční stav a koncový stav.

7.4.2.6 Vlastnosti algoritmu BS

Úplnost – algoritmus je úplný.

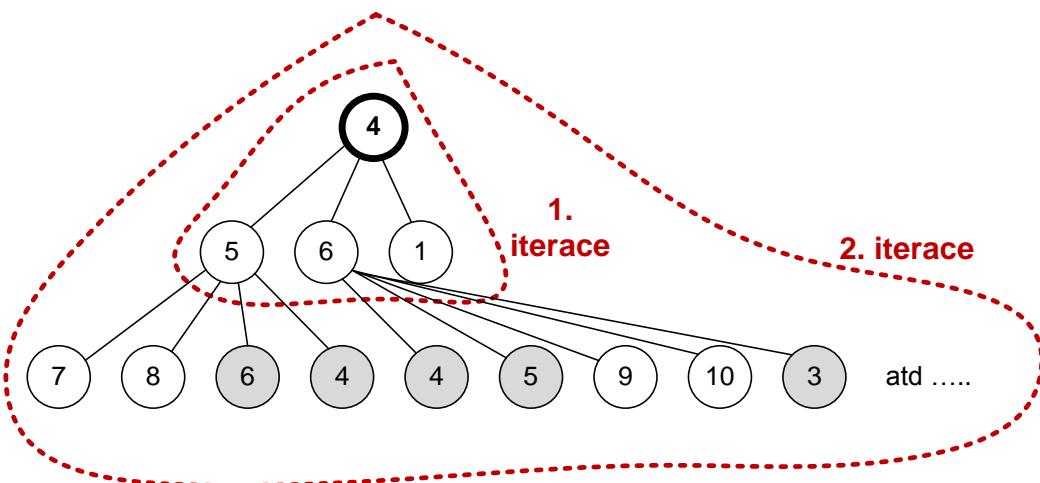
Optimálnost – algoritmus je optimální, je nalezeno takové řešení, ke kterému vede nejmenší počet kroků.

Prostorová složitost – BFS/2.

Časová složitost – Velký problém je porovnávání množin OPEN z obou směrů. V takovém případě se dostaváme na $O(m^*2^n)$, kde m je počet prvků z množiny 1 a n je počet prvků z množiny 2.

7.4.2.7 Iterativní prohledávání do hloubky s omezenou hloubkou, IDLS

Algoritmus IDLS je rozšířenou variantou DLS. Začíná se na nízké hodnotě hloubky. Pokud v ní je nalezeno řešení, algoritmus končí. Pokud ne, hloubka se inkrementuje a algoritmus se spouští znova (iteruje).



Obr. 103: Postup procházení stavovým prostorem grafu v případě algoritmu IDLS. V obrázku jsou vyznačeny pouze 2 iterace, ve skutečnosti může být neomezený počet.

7.4.2.8 Vlastnosti algoritmu IDLS

Úplnost – algoritmus je úplný.

Optimálnost – algoritmus je optimální, je nalezeno takové řešení, ke kterému vede nejmenší počet kroků.

Prostorová složitost – Jako DFS.

Časová složitost – suma jednotlivých iterací, kde v každé iteraci je použit algoritmus DLS.

7.4.2.9 Uniform-cost search (UCS, Dijkstrův algoritmus)

UCS algoritmus se opírá o ohodnocení dosud procházené cesty. Z toho důvodu se občas řadí i mezi informované metody.

POZN: (UCS) algoritmu se budeme krátce věnovat v sekci informovaných metod.

7.4.2.10 Algoritmus

```

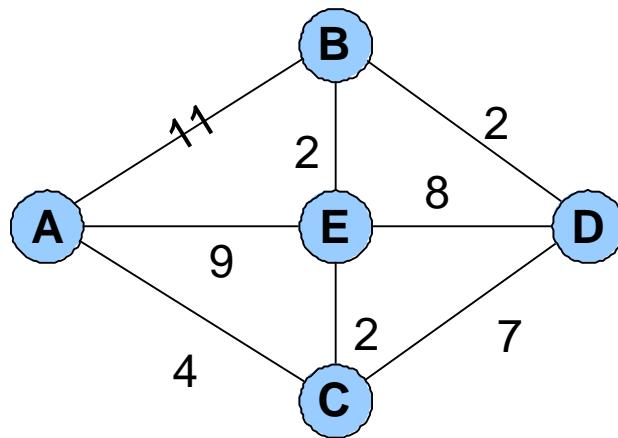
procedure UCS() {
    OPEN := [(s0,0)]; // přiřad' počáteční stav
    while open.isNotEmpty do {
        z množiny OPEN vyber stav s nejmenší cenou, označ ji X;

```

```

if (X je cíl) then {
    return(nalezeno)
} else {
    Generuj potomky X, spočti jejich cenu a přidej je do OPEN;
    Případné mnohonásobné výsky odstraň a ponech jediný s nejnižší
    cenou;
}; // if
} // while
return(NENALEZENO);
} // procedure

```



Obr. 104: Příklad grafu

Příklad: Nalezněte cestu z A do D pomocí algoritmu UCS (Dijkstrova algoritmu).

Tučně jsou označeny stavy s nejmenším ohodnocením a červeně jsou ohodnoceny stavy, které jsou řešením.

- 0) **[((A),0)]**
- 1) **[((B,A),11), (([E,A]),9), ((C,A),4)]**
- 2) **[((B,A),11), (([E,A]),9), ((A,C,A),8), ((E,C,A),6), ((D,C,A),11)]**
- 3) **[((B,A),11), ((A,C,A),8), ((B,E,C,A),8), ((A,E,C,A),15), ((C,E,C,A),8), ((D,E,C,A),14), ((D,C,A),11)]**
- 4) **[((B,AC,A),19); ((E,A,C,A),17); ((C,A,C,A),12); ((B,E,C,A),8); ((C,E,C,A),8); ((DCA),11)]**
- 5) **[((E,A,C,A),17); ((A,B,E,C,A),19); ((E,B,E,C,A),10); ((D,B,E,C,A),10); ((C,E,C,A),8); ((D,C,A),11)]**
- 6) **(((A,B,E,C,A),19); ((E,B,E,C,A),10); **((D,B,E,C,A),10)**; ((A,C,E,C,A),12); ((E,C,E,C,A),10); **((D,C,E,C,A),15)**;]**
- 7) **DOKONČENO**

7.4.3 Informované metody

Informované metody jsou takové metody, které upřednostňují některé stavy oproti jiným k tomu, aby nalezli řešení dříve. Nevýhodou je, že musíme navrhnout nějakou fitness funkci, která provede ohodnocení každého stavu. S tím bohužel souvisí i dodatečný čas, který k tomuto spotřebujeme.

7.4.3.1 Best FS

BestFS je algoritmus, který je optimalizací neinformované metody BFS. Algoritmus stejně jako BFS expanduje aktuální stav, pro pokračování ale bere ten stav, který se zdá být nejvíce slibný pro konvergenci k řešení. Funkce, která se pokouší odhadnout správnou cestu je potom:

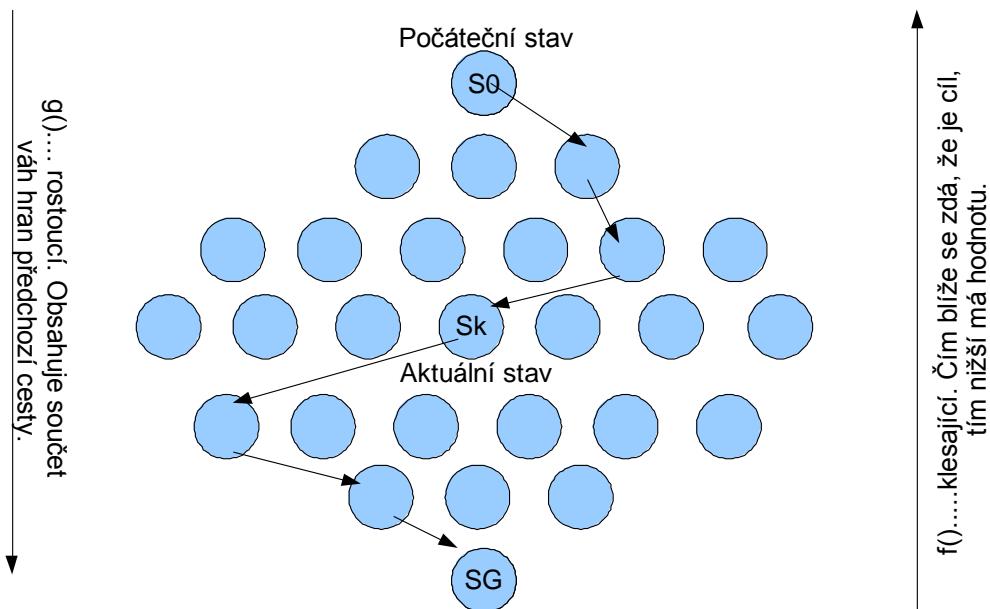
$$F(S_k) = g(S_k) + h(S_k) = g(S_k - S_0) + h(S_G - S_k)$$

$g(S_k)$... funkce zobrazení, ohodnocuje váhu hran, neboli – kolik už mě současná cesta stála?

Čím dále od počátku S_0 , tím je vyšší (tj. rostoucí) (viz Obr. 105).

$h(S_k)$... heuristická funkce, pokouší se odhadnout, jak jsem blízko ke hledanému stavu S_G .

Čím blíže cíli S_g , tím je nižší (tj. klesající) (viz Obr. 105).



Obr. 105: Prohledávání stavového prostoru s využitím BFS algoritmu, kde je předpovídána nejkratší cesta k řešení.

Pro implementaci lze pak jednoduše použít algoritmus BFS, kde je jediná modifikace – namísto obyčejné fronty se použije fronta prioritní. Stavy S_k se budou řadit dle hodnoty fitness funkce $F(S_k)$ a v tomto pořadí budou také prováděny. Volbou vhodného poměru hodnot funkcí $h(S_k), g(S_k)$ výrazně ovlivníme chování algoritmu. K optimální volbě bohužel neexistuje nějaké obecné vodítko a závisí na konkrétním případě.

POZN: Který ADT je vhodný pro implementaci prvků, které jsou řazeny přímo při vkládání? Neboli, jinými slovy, který ADT je vhodný pro implementaci prioritní fronty⁵.

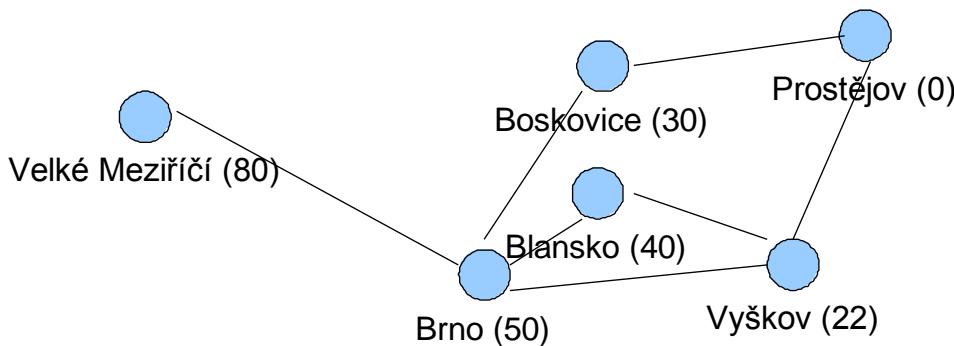
Algoritmus BestFS se dle tvaru funkce $f()$ mohou jmenovat jako **Greedy Search (GS)** a již dříve zmíněná **Uniform Cost Search**, popřípadě také A* (čti A star). Greedy search se řídí jen dle odhadu a nikoli dle počtu kroků, UCS se naopak řídí jen cenou uražené cesty, A* je kombinací obou těchto variant.

7.4.3.2 Greedy Search (GS)

Rovnice se tedy dá upravit na:

$$F(S_k) = g(S_k) + h(S_k) = h(S_G - S_k), \text{ kde } g(S_k) = 0$$

V praxi si to můžete představit jako vyhledávání cesty na mapě. Každé město ohodnotíme počty km , které zbývají k dosažení cíle. Při procházení stavů se pak upřednostňuje stav s nejmenším ohodnocením (neboli, dle pořadí, jak jsou uspořádány v prioritní frontě, o pořadí rozhoduje hodnota funkce $F(S_k)$, je zřejmé, že stavů se stejnou fitness funkcí se mohou přepsat jen v tom případě, shoduje-li se i aktuální město).



Díky tomu, že UCS je vlastně speciálním případem BestFS se někdy také řadí i mezi metody informované. Správně by měla být klasifikována jako metoda neinformovaná, protože se nepokouší o odhad hodnoty stavu. Její popis se dá matematicky vyjádřit jako:

$$F(S_k) = g(S_k) + h(S_k) = g(S_G - S_k), \text{ kde } h(S_k) = 0$$

7.4.3.3 Hill Climbing

Hill climbing je algoritmus, který optimalizuje řešení z pohledu časové složitosti. Je velice podobný algoritmu BFS – stejně jako BFS vybírá nejlepší stav a expanduje jej. Na rozdíl od BFS odstraňuje všechna starší řešení a ponechává pouze tyto poslední expandované stavы.

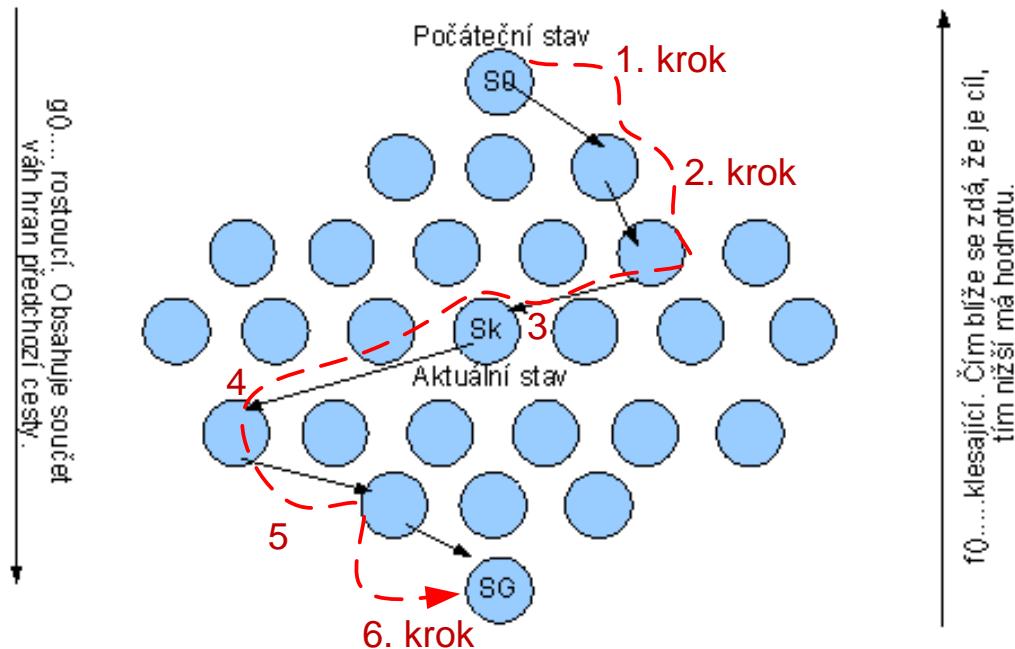
Optimalizace má pozitivní vliv na nároky v paměti, na druhou stranu algoritmus nemusí najít řešení i přesto, že alespoň jedno existuje (algoritmus není úplný). Čas pro nalezení řešení konverguje k nekonečnu, na druhou stranu, paměťová složitost je rovna rádu větvení b (stupni stromu stavového prostoru).

⁵ Prioritní fronta je často používaný ADT, zejména v metodách umělé inteligence a v systémech hromadných obsluh.

7.4.3.4 Algoritmus A*

A*(čtěte: A Star) je velice podobný navíc bere v **potaz i kompletní cestu**, kterou již algoritmus **prošel** (nikoli jen aktuální krok). To má za následek **kompletnost i optimálnost algoritmu** (na rozdíl od BestFS).

$$F(S_k) = g(S_k) + h(S_k) = g(S_k - S_0) + h(S_G - S_k)$$

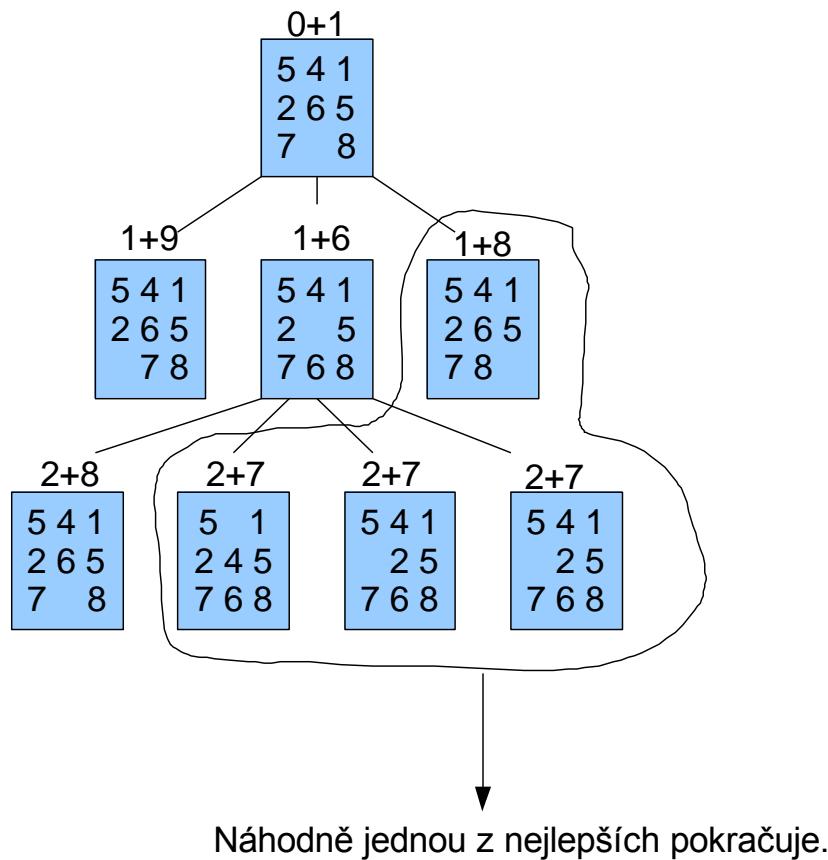


```

public void A*(start,goal) {
    TreeSet closed := [] // prazdna mnozina
    PriorityQueue open := [ start ] // prioritni fronta
    while open.isNotEmpty() {
        var act := q.removeFirst(); // vraci nejlepe ohodnoceny stav

        if (act.equals(goal)) {
            // Nalezeno reseni
            // vypisi cestu, jak jsme k reseni dospeli
            return p;
        }
        closed.add(p);
        foreach y in naslednici(act) { // vsechny nasledne uzly
            y.computeFitness();
            y.setParent(act);
            open.addOrReplace(y); // pridej do fronty,
            // jestliže stejny stav již v open existuje v closed,
            // porovnej jejich fitness a ponech lepsiho
        }
        closed.add(act);
    }
    return failure; // reseni nenalezeno
}

```



Obr. 106: Řešení hry čísla s využitím algoritmu A*.

Časová složitost algoritmu je $O(b^d)$, kde b je stupeň stromu stavového prostoru a d je hloubka, kde se řešení nachází. Prostorová složitost je obdobně jako časová rovna $O(b^d)$. Algoritmus je úplný a je i optimální.

Pro použití A* pro hledání cesty viz [5].

7.5 Další využití vyhledávání v grafech

- Stolní hry – Šachy, dáma
- Hanojské věže
- „Posunování čísel“
- Paralelizace
- Průchod bludištěm

7.6 Literatura

- [1] ŠEDA, M. Teorie grafů, Vysoké učení technické v Brně, Fakulta strojního inženýrství <www.uai.fme.vutbr.cz/~mseda/TG03_MS.pdf>
- [2] HLINĚNÝ, P., Teorie Grafů, Fakulta informatiky, Masarykova univerzita <<http://www.fi.muni.cz/~hlineny/Vyuka/GT/Grafy-text07.pdf>>
- [3] Demonstrace prohledávání do šířky,

- <http://ui.fpf.slu.cz/diplomky/umela_inteligence/Do_Sir.htm>
- [4] Demonstrace prohledávání do hloubky,
<http://ui.fpf.slu.cz/diplomky/umela_inteligence/Do_Hl.htm>
- [5] The map path fading with A*
<<http://www.cokeandcode.com/pathfinding>>

8 Algoritmy řazení

8.1 Úvod

Tato kapitola se zabývá řadícími algoritmy. Řadící algoritmus je takový algoritmus, který má

- **VSTUP:** sekvence n čísel: $\{a_1, a_2, \dots, a_n\}$
- **VÝSTUP:** sekvence přeuspořádaných čísel $\{a'_1, a'_2, \dots, a'_n\}$, kde platí, že $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Řazené položky jsou zpravidla kolekcí několika hodnot. V této kapitole se pro jednoduchost budeme zabývat pouze řazení celočíselných typů, nicméně stejně tak, jak řadíme celá čísla, mohou být řazena jakákoli data. Jediné co musíme definovat je způsob, jak jednotlivé prvky porovnávat (např. v případě třídy auta dle roku výroby, najetých kilometrů atd.).

Mnoho počítačových vědců považuje algoritmy řazení za stěžejní část počítačové vědy.

Algoritmu řazení je v dnešní době celá řada a dokonce dosud stále nové vznikají. Asi nejznámějšími zástupci z řady řadících algoritmů jsou Bubble sort, Selection sort, Insertion sort, Merge sort, Heap sort a Quick sort. Tyto algoritmy je dále možné rozdělit na jednoduché řadící algoritmy (Bubble sort, Selection sort, Insertion sort) a pokročilejší (Merge sort, Heap sort a Quick sort). V případě jednoduchých řadících algoritmů jsou dosažené výsledky z hlediska času nepříliš kvalitní a jejich nasazení pro větší množství prvků (ve většině případů) vede k časově náročnému výpočtu. Jinými slovy, časová složitost těchto algoritmů není příliš dobrá. Algoritmy řazení jsou obecně založeny na jednom z těchto 4 typů: výměna, výběr, vkládání, spojování a rozklad.

Pokud bude ve zbytku textu použita proměnná n , její význam je v počtu prvků pole, na které je aplikován řadící algoritmus.

Algoritmus řazení	Nejlepší časová složitost	Průměrná časová složitost	Nejhorší časová složitost	Paměťová složitost	Stabilita	Způsob
Bubble sort	$O(n)$	—	$O(n^2)$	$O(1)$	Ano	Výměna
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ne	Výběr
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ano	Vkládání
Shell sort	—	—	$O(n^{1.5})$	$O(1)$	Ne	Vkládání
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Ano	Spojování
Optimalizovaný merge sort (in-place)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Ano	Spojování
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Ne	Výběr
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Ne	Rozkládání

Tab. 8: Přehled algoritmů řazení a jejich časových složitostí.

8.1.1 Stabilita řadícího algoritmu

Stabilní řadící algoritmy jsou takové algoritmy, které zachovávají relativní pořadí prvků u položek, kde jsou klíče ekvivalentní. Proto, řadící algoritmus je stabilní, jestliže pro dvě položky R , a S se shodnými klíči S a R , které se vyskytuje před S v původním poli/seznamu, R se opět objeví R opět před S . Pro názornost viz následující příklad.

Příklad:

Mějme pole dvojic prvků a dva různé řadící algoritmy, kde klíčem pro řazení prvků je v případě prvního řadícího algoritmu první číslo z dvojice čísel a druhé dle druhého čísla z dvojice čísel (tj. řadí dle odlišných klíčů). Vstupem algoritmu č. 1 bude následující tabulka:

[5, 4]	[1, 7]	[2, 3]	[3, 7]
--------	--------	--------	--------

Pokud tedy pole seřadíme dle prvního řadícího algoritmu, algoritmus bude brát v potaz jen první číslo a na jeho základě prvky seřadí. Pořadí prvků bude tedy následující:

[1, 7]	[2, 3]	[3, 7]	[5, 4]
--------	--------	--------	--------

Vezměme nyní výstup z předchozího řazení a seřadíme dle druhého algoritmu řazení. Tento algoritmus porovnává prvky pouze dle druhého čísla. Správné výsledky tedy mohou být dva:

[2, 3]	[5, 4]	[1, 7]	[3, 7]
--------	--------	--------	--------

[2, 3]	[5, 4]	[3, 7]	[1, 7]
--------	--------	--------	--------

V případě položek [2,3] a [5,4] je pořadí naprosto jednoznačné. Ale v případě prvků [1,7], [3,7] mohou nastat dva různé případy. Pokud použitý algoritmus řazení je stabilní, nemůže nastat případ, kdy by se položky [1,7] a [3,7] vyskytly v opačném pořadí (tj. [3,7] a [1,7]). Jinými slovy:

Stabilní algoritmus ponechá předchozí pořadí pro položky o stejně hodnotě klíčů.

8.2 Proč je nutné algoritmy řazení znát

V dnešní době se snad již nesetkáte s programovacím prostředím, které by v sobě implicitně neimplementovaly algoritmus řazení a to často efektivněji, než bychom byli v rozumném čase schopni implementovat. Na první pohled se tedy může zdát, že se samotnou implementací řadícího algoritmu nebudeste muset přijít nikdy do styku.

Důvodem, proč je nutné řadící algoritmy znát, je skutečnost, že počítačová věda a hardware, který používáme, se velice rychle mění. Je to jedna z nejdynamičtěji se vyvíjejících oborů vědců a je téměř nevyhnutelné, že se v následujících několika letech setkáme s paralelními a distribuovanými systémy, které umožní další zvyšování výpočetního výkonu. V tomto ohledu je také nevyhnutelné znát základní algoritmy a případně být schopen na základě těchto znalostí optimalizovat výpočetní výkon.

8.3 Jednoduché řadící algoritmy

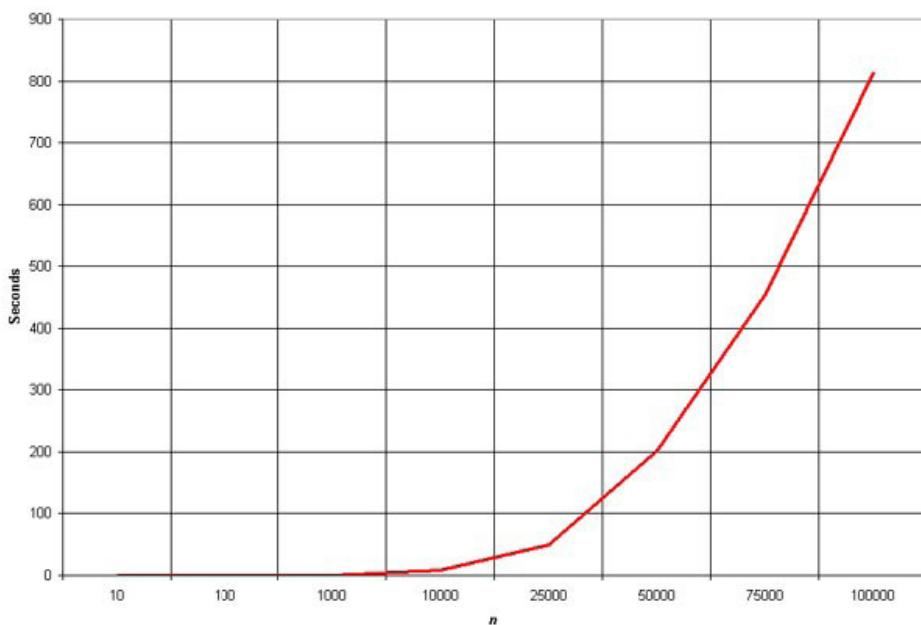
8.3.1 Bubble sort

Je nejjednodušším, ale současně také nejpomalejším třídícím algoritmem. Bublinkové řazení (angl. Bubble sort) se chová tak, že porovnává každý prvek se svým následníkem, a je-li tento větší, pak je zaměnění. Toto provede pro celou posloupnost n prvků. Celý postup (n porovnání a záměn) musíme aplikovat n -krát, abychom si byli jisti, že prvky jsou seřazeny. Největší prvky tak „probublají“ na konec seznamu, odtud název algoritmu.

Existují některá drobná vylepšení algoritmu, přes to všechno se ale nezdá být příliš dobrý. Asi nejčastěji používanou optimalizací je hlídání, zdali v posledním průchodu došlo alespoň k jednomu prohození. Pokud tomu tak není, znamená to, že algoritmus již má seřazenou posloupnost a nemusí tedy ve výpočtu pokračovat.

Pokud necháme algoritmem projít již seřazenou posloupnost, je tato metoda jedna z nejrychlejších. Toho můžeme využít, pokud chceme např. zjistit, zda je pole již seřazené.

Na Obr. 107 je znázorněn graf doby řazení v závislosti na počtu řazených položek.



Obr. 107: Časová složitost řadícího algoritmu bubble sort v závislosti na počtu řazených prvků (n).

Kód bez optimalizace:

```
public static void bubbleSort1(int[] x) {
    int n = x.length;
    for (int pass=1; pass < n; pass++) { // count how many times
        // This next loop becomes shorter and shorter
        for (int i=0; i < n-pass; i++) {
            if (x[i] > x[i+1]) {
                // exchange elements
                int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp;
            }
        }
    }
}
```

Příklad: (**červeně** je značeno prohození prvků, **zeleně** žádná změna, **modře** seřazená část pole)

VSTUP:						
5	3	6	1	4	2	

1. krok	<u>3</u> 5	6	1	4	2	
2. krok	3	1	4	2		
3. krok	3	5	4	2		
4. krok	3	5	1	2		
5. krok	3	5	1	4		
Další iterace						
6. krok	<u>3</u> 5	1	4	2	6	
7. krok	3	4	2	6		
8. krok	3	1	2	6		
9. krok	3	1	4	6		
Další iterace						
10. krok	<u>1</u> 3	4	2	5	6	
11. krok	1	2	5	6		
12. krok	1	3	5	6		
Další iterace						
13. krok	<u>1</u> 3	2	4	5	6	
14. krok	1	4	5	6		
Další iterace						
15. krok	<u>1</u> 2	3	4	5	6	

VÝSTUP:						
1	2	3	4	5	6	

Varinty Bubble-Sort

- **Ripple-Sort** - pamatuje si, kde došlo v minulém průchodu k první výměně a začíná až z tohoto místa
- **Shaker-Sort** - prochází oběma směry a "vysouvá" nejmenší na začátek a největší na konec
- **Shuttle-Sort** - dojde-li k výměně, algoritmus se vrací s prvkem zpět, pokud dochází k výměnám. Pak se vrátí do místa, odkud se vrácel a pokračuje
- **Odd-even sort** je paralelní variantou tohoto algoritmu. S výhodou se využívá skutečnosti, že prohazování prvků může s využitím více procesorů probíhat paralelně.

Optimalizace 1 - kontrola nutnosti prohození

```
public static void bubbleSort2(int[] arr) {
    int n = arr.length;
    for (int pass=1; pass < n; pass++) { // count how many times
        boolean swap = false;
        // This next loop becomes shorter and shorter
        for (int i=0; i < n-pass; i++) {
            if (arr[i] > arr[i+1]) {
                // exchange elements
                int temp = arr[i]; arr[i] = arr[i+1]; arr[i+1] = temp;
                swap = true;
            }
        }
        if (!swap)
            break;
    }
}
```

```
        printArr(arr);
    } else {
        printArr(arr);
    }
}
if(!swap) {
    break;
}
System.out.println("Další iterace");
}
}
```

VSTUP:						
1	2	3	4	5	6	

1.	krok	1	<u>2</u>	3	4	5
2.	krok	1	<u>2</u>	<u>3</u>	4	5
3.	krok	1	2	<u>3</u>	<u>4</u>	5
4.	krok	1	2	3	<u>4</u>	<u>5</u>
5.	krok	1	2	3	4	<u>5</u>

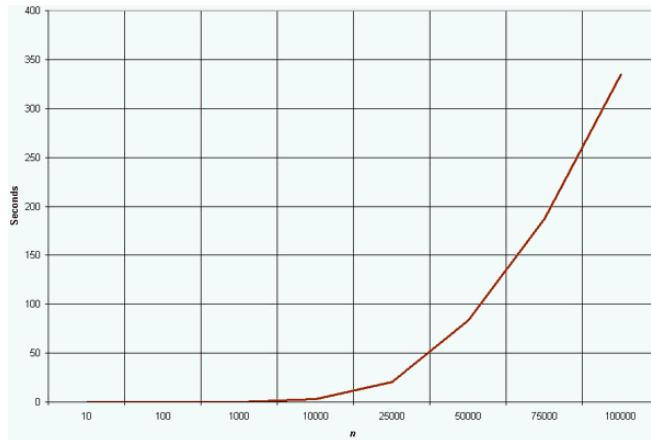
VÝSTUP:						
1	2	3	4	5	6	

Tato optimalizace se bohužel v případě vstupu opačně seřazené posloupnosti nijak neprojeví. I přes řadu optimalizací není Bubble sort s výjimkou paralelní varianty kvalitním řadícím algoritmem.

8.3.2 Insertion sort

Algoritmus Insertion sort patří do skupiny jednodušších řadících algoritmů a jeho výsledky pro větší počty prvků nejsou příliš dobré. Algoritmus pracuje na principu, kde několik prvních prvků je již seřazených a následující neseřazený prvek je vkládán na správnou pozici. Insertion sort je inkrementální algoritmus, kde vznikne vždy jeden seřazený prvek za jeden blok operace (zejména u složitějších algoritmů toto nemusí platit). Této vlastnosti může být využito u paralelních algoritmů.

Výhoda tohoto algoritmu je, že je vcelku efektivní pro malé objemy dat, je stabilní a nevyžaduje žádnou nadbytečnou paměť, tj. paměťová složitost je konstantní O(1). Na Obr. 108 je zobrazena časová závislost řazení algoritmu na základě počtu řazených prvků.



Obr. 108: Časová složitost řadícího algoritmu insertion sort v závislosti na počtu řazených prvků (n).

```
public static void insertionSort(int[] a) {
    for (int i=1; i < a.length; i++) {
        [1]vlož prvek a[i] seřazené části pole
    }
}
```

Algoritmus pro vložení prvku $a[i]$ do seřazené části pole:

```
int v = a[i];
int j;
for (j = i - 1; j >= 0; j--) {
    if (a[j] <= v) break;
    a[j + 1] = a[j];
}
a[j + 1] = v;
```

Příklad: **zeleně** je označen aktuálně vkládaný prvek, **červeně** jsou označeny posunuté prvky

VSTUP:							
5	3	6	1	4	2		

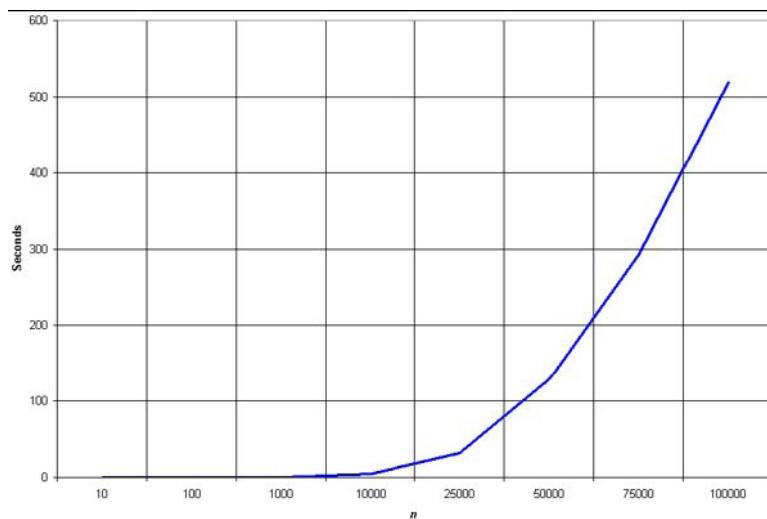
3	5	6	1	4	2		
Vkládám prvek i = 1							
3	5	6	1	4	2		
Vkládám prvek i = 2							
1	3	5	6	4	2		
Vkládám prvek i = 3							
1	3	4	5	6	2		
Vkládám prvek i = 4							
1	2	3	4	5	6		
Vkládám prvek i = 5							

VÝSTUP:							
1	2	3	4	5	6		

8.3.3 Selection sort

Tato metoda pracuje tak, že vyhledá nejmenší prvek v nesetříděné části a zařadí ho na konec již setříděné části. Musíme takto projít celé pole, najít nejmenší prvek a zařadit ho na první místo. Poté se znova musí projít pole od druhého prvku pole (první prvek má již svou konečnou pozici a není třeba jej měnit) a vyhledá opět nejmenší prvek. Ten zařadí na druhou pozici. Tato činnost se opakuje tak dlouho, dokud neprojde celou posloupnost a nesetřídí ji.

Druhou možností je, že algoritmus vyhledává největší prvek v nesetříděné části a zařadí ho na konec nesetříděné části. Tzn. že musíme projít celé pole, najít největší prvek a zařadit ho na poslední místo. Poté znova musí projít pole od předposledního prvku pole (poslední prvek má již svou konečnou pozici) a vyhledá opět největší prvek. Ten zařadí na předposlední pozici. Tato činnost se opakuje pro celou posloupnost N-krát, kde N je počet prvků v seznamu.



Obr. 109: Časová složitost řadícího algoritmu selection sort v závislosti na počtu řazených prvků (n).

Algoritmus selection sort:

```
static void selectionSort(int numbers[]) {
    int arraySize = numbers.length;
    int i, j;
    int min, temp;
    for (i = 0; i < arraySize-1; i++) {
        min = i;
        for (j = i + 1; j < arraySize; j++) {
            if (numbers[j] < numbers[min])
                min = j;
        }
        // prohod hodnoty
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

```

    printArr(numbers);
}

8.3.3.1 }

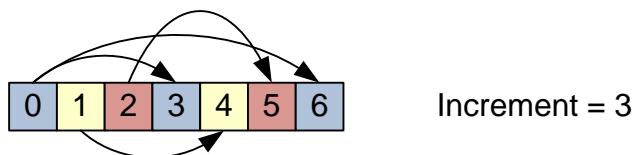
```

8.3.3.2 Příklad

VSTUP:									
5	3	6	1	4	2				
<hr/>									
1	3	6	5	4	2				
1	2	6	5	4	3				
1	2	3	5	4	6				
1	2	3	4	5	6				
1	2	3	4	5	6				
<hr/>									
VÝSTUP:									
1	2	3	4	5	6				

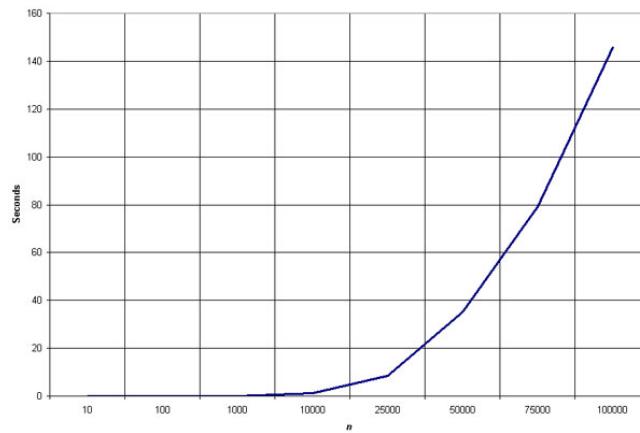
8.3.4 Shell sort

Shell sort je nejfektivnější z modifikovaných jednoduchých řadících algoritmů a je vhodný pro pole o menší m počtu prvků. Metoda se jmenuje dle jejího objevitele Donalda Shella, který zjistil na základě pozorování a testů zjistil, že prvek se průměrně posune o jednu třetinu vzdálenosti. Je velice podobná algoritmu bubble sort – tedy porovnává a zaměňuje hodnoty. Rozdíl je v tom, že neporovnává sousedící prvky, ale prvky o vzdálenosti d a v následujícím kroku se vždy změní vzdálenost d na $\frac{d}{2}$.



Obr. 110: Porovnávání s prvky pomocí Shell sort a krokem rovným 3.

Bohužel je také nejvíce komplikovaný z této třídy algoritmů. V porovnání oproti složitějším algoritmům jako je heap, merge, či quick sort nedosahuje zdaleka tak dobrých výsledků, nicméně empirické zkušenosti s ním dávají pro malé počty položek relativně dobré výsledky.



Obr. 111: Složitost řadícího algoritmu shell sort v závislosti na počtu řazených prvků (n).

8.3.4.1 Příklad

1. krok (první prohodím s nejmenším) 2. krok (druhý prohodím s nejmenším) 3. krok (třetí prohodím s nejmenším) 4. krok (čtvrtý prohodím s nejmenším) 5. krok (prohazuju sama na sebe)	VSTUP: 5 3 6 1 4 2 ----- increment = 3 1 3 6 5 4 2 1 3 6 5 4 2 1 3 2 5 4 6 increment = 1 1 3 2 5 4 6 1 2 3 5 4 6 1 2 3 4 5 6 1 2 3 4 5 6 ----- VÝSTUP: 1 2 3 4 5 6
---	--

Implementace potom může vypadat nějak takto:

```

public static void shellSort(int[] a) {
    for (int increment = a.length / 2; increment > 0;
         increment = (increment == 2 ? 1 : (int) Math.round(increment / 2.2))) {
        for (int i = increment; i < a.length; i++) {
            for (int j = i; j >= increment && a[j - increment] > a[j];
                 j -= increment) {
                System.out.println("INCR:" + increment +
                                   " " + (j - increment) + " -> " + j);
                int temp = a[j];
                a[j] = a[j - increment];
                a[j - increment] = temp;
            }
            printArr(a);
        }
    }
}
  
```

8.4 Pokročilejší algoritmy řazení

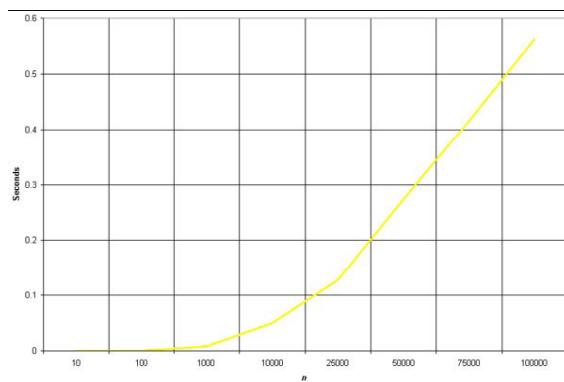
8.4.1 Merge sort

Merge sort je řadící algoritmus, jehož průměrná i nejhorší možná časová složitost je $O(N * \log N)$. Patří do skupiny složitějších algoritmů z hlediska složitosti algoritmu.

Nevýhodou algoritmu ve srovnání s ostatními algoritmy je, že navíc potřebuje pole o velikosti n (neboli, paměťová složitost algoritmu je $O(n)$). Proto existuje in-place varianta algoritmu, která pracuje přímo v poli.

Výhodou algoritmu mergesort je, že algoritmus je stabilní a je lze jej relativně vůči ostatním dobrě paralelizovat.

Merge sort je kvalitní algoritmus jak z hlediska časové, tak i paměťové náročnosti (viz Obr. 112).

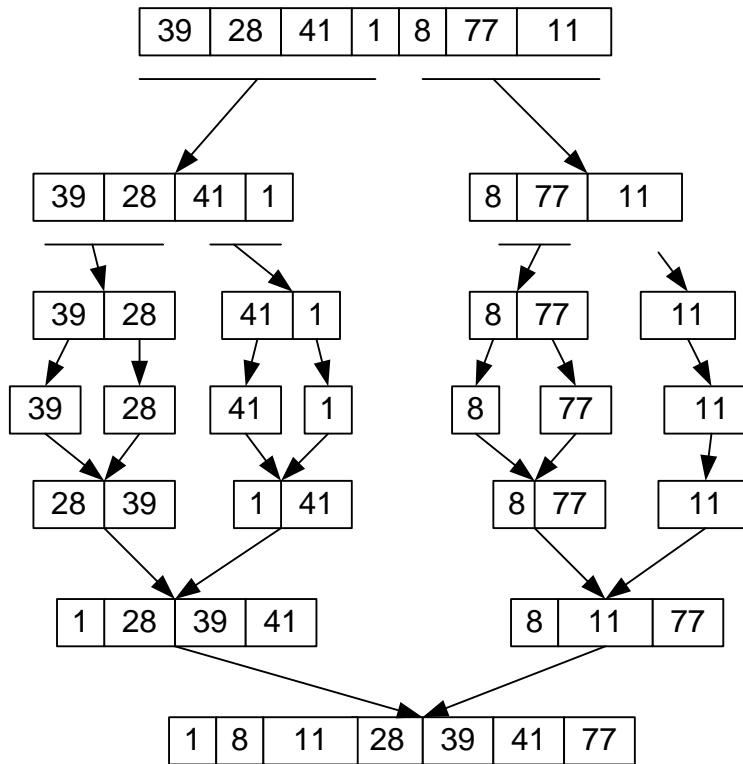


Obr. 112: Složitost řadícího algoritmu merge sort v závislosti na počtu řazených prvků (n).

Algoritmus pracuje následovně:

1. Rozdělí neseřazenou množinu dat na dvě podmnožiny o přibližně stejné velikosti
2. V případě, že velikost podmnožiny je větší než jedna, aplikuj na ni opět algoritmus Merge Sort
3. Spojí seřazené podmnožiny do jedné seřazené množiny

Algoritmus vytvořil v roce 1945 John von Neumann. Na Obr. 113 je příklad řazení pole o délce 7 prvků.



Obr. 113: Příklad řazení s využitím algoritmu merge sort.

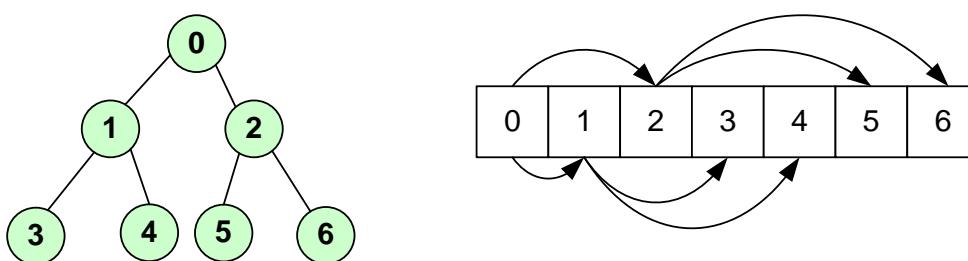
Při slučování jednotlivých posloupností se potom mohu opřít o skutečnost, že jednotlivé posloupnosti již jsou seřazeny, tedy jedná se o výrazně rychlejší operaci než by bylo řazení.

8.4.2 Heap sort

Základní myšlenkou tohoto kroku je využití datové struktury označované jako halda (angl. heap). Tato struktura umí velmi efektivně provést operaci vložení prvku a operaci výběr největšího prvku. Proto lze pomocí haldy setřídit dodaná data od největšího k nejmenšímu již při jejich vkládání do haldy a následného postupného vybírání největšího prvku.

Pokud srovnáme řadící algoritmus Heapsort s ostatními řadícími technikami, je doba běhu algoritmu stejně jako merge sort $O(n \lg n)$. To je výrazně lepší než např. insertion sort. Stejně jako insertion sort a na rozdíl od merge sort algoritmus řadí „in-place“, tedy nevyžaduje žádnou dodatečnou paměť.

V praxi lze haldu vystavět přímo ve vstupním poli tím způsobem, že jsou následovníci prvku n uloženi do prvků $2n$ a $2n+1$ (při indexování od jedničky), a také následné vybírání prvků lze provádět pouhým přeuspořádáváním dat v tomto poli (viz Obr. 114).



Obr. 114: Reprezentace ADT strom za pomocí ADT pole.

Výpočet se skládá se ze dvou fází:

- ustavení hromady
- protřásání hromadou

8.4.2.1 Ustavení haldy

Při ustavování haldy se postupuje postupným prodlužováním haldy z rozsahu $(N/2 \text{ až } N)$ na $(1 \text{ až } N)$. Po provedení $N/2$ kroků je halda vytvořena.

8.4.2.2 Využití haldy

Halda, která splňuje popsané podmínky, má na vrcholu (index 1) prvek s největší hodnotou. Ve druhém kroku haldového třídění se tento prvek vymění za poslední prvek pole. Tak se na konec pole dostane největší prvek (tím je zatříden na správné místo) ale prvkem, přesunutým z konce pole dojde k porušení pravidel haldy. Je třeba spustit pomocný algoritmus, tentokrát pro prvky s indexy $(1 \text{ až } N-1)$.

Výše zmíněný postup se opakuje pro stále se zmenšující haldu. Při každém kroku je na vrcholu haldy největší ze zbývajících prvků, a ten je výměnou s posledním prvkem této menší haldy zařazen na správné pořadí v poli.

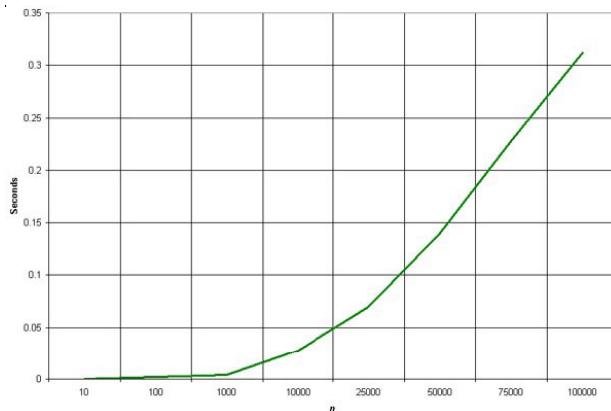
8.4.2.3 Srovnání

Hlavními konkurenty algoritmu heapsort jsou mergesort a quick sort. Výhoda heapsort je, že nejhorší případ časové složitosti $O(n\log(n))$ oproti $O(n^2)$ v případě quick sort. Naopak v ohledu časové složitosti nepotřebuje heapsort žádnou dodatečnou paměť, zatímco mergesort vyžaduje dodatečnou paměť s lineární složitostí.

8.4.3 Quick sort

Quicksort[2], neboli rychlé řazení, je jeden z nejrychlejších známých algoritmů. Jeho průměrná časová složitost je pro algoritmy této skupiny nejlepší možná ($O(N \log N)$), v nejhorším případě (kterému se ale v praxi jde obvykle vyhnout) je však jeho časová náročnost $O(N^2)$. Algoritmus je nejblíže architekturám dnešních počítačů a z toho důvodu lze vnitřní cyklus tohoto algoritmu velice efektivně implementovat. Algoritmus quick sort se používá velice často. Na Obr. 115 je zobrazena experimentálně změřená časová závislost na počtu řazených prvků.

Nevýhodou algoritmu je, že se dá hůře paralelizovat. Hlavním konkurentem tohoto algoritmu je heapsort, který má zaručenou lepší složitost v notaci Omikron[1].



Obr. 115: Časová složitost řadícího algoritmu Quick sort v závislosti na počtu řazených prvků (n).

Princip fungování se dá popsát přibližně takto: Zvol náhodnou hodnotu pivotu z oboru hodnot řazených prvků (ideálně by vybral prostřední hodnotu, časová složitost této operace by ale efektivitu celého algoritmu degradovala, proto se vybere náhodná)

- Prvky menší než pivot přesuň doleva, prvky větší než pivot přesuň doprava
- Rekurzivně proved' to samé pro levou a pravou část.

Pseudo-kód příkladu řadícího algoritmu quicksort (varianta s využitím dodatečné paměti):

```

public Array quicksort(q) {
    var list less, pivotList, greater
    pivotList = { pivot }
    if length(q) ≤ 1
        return q
    select a pivot value from q
    for each x in q
        if x < pivot then add x to less
        // kdyby byly povoleny i více shodných hodnot,
        // můsely by se x == pivot vkládat do pivotList.
        if x > pivot then add x to greater
    return concatenate(quicksort(lessList), pivotList,
                        quicksort(greaterList))
}

```

V ideálním případě by se při výběru mediánu měl zvolit medián celého pole. Jelikož by složitost této operace výrazně zpomalila celý algoritmus, volí se náhodná hodnota. A jelikož předpokládáme náhodně seřazenou posloupnost, můžeme zvolit nejpravější položku a považovat ji za medián.

8.4.3.1 Volba pivota

Velkým problémem celého algoritmu je volba pivota, kterému se nyní budeme věnovat. Pokud se daří volit číslo blízké mediánu řazené části pole, je algoritmus skutečně velmi rychlý. Pokud ne, je jeho paměťová i časová náročnost horší než u všech používaných řadicích algoritmů. Medián můžeme tedy vypočítat a zvolit jej za pivota. Toto je ale velmi neefektivní metoda, protože hledání mediánu by konzumovalo čas o velikosti $O(N)$. To by

celý algoritmus degradovalo a ostatní by byly daleko efektivnější. Zde je seznam, pouze několika metod:

- **První prvek** - popřípadě kterákoli jiná fixní pozice. **Velmi nevýkonná především na částečně seřazených množinách.** My jej budeme používat pro ruční řazení a to z toho důvodu, že by bylo velice obtížné hledat chyby.
- **Náhodný prvek** - často používaná metoda. Jde dokázat, že pokud je pozice pivotu skutečně náhodná, algoritmus poběží v $O(N \log N)$. Skutečně náhodné čísla generují ale pouze hardwarové generátory, které nemusí dodávat data dostatečně rychle. V praxi mnohdy stačí použít pseudonáhodný algoritmus.
- **Metoda mediánu tří** - případně pěti, či libovolné jiné konstanty. Pomocí pseudonáhodného algoritmu (používají se i fixní pozice) se vybere X prvků z množiny, ze kterých se použitím některého primitivního řadícího algoritmu najde medián a ten je zvolen za pivota.

Přestože **Quicksort** nemá zaručenou časovou složitost $O(N \log N)$, reálné aplikace a testy ukazují, že na pseudonáhodných datech je vůbec nejrychlejší ze všech obecných řadících algoritmů (tedy i rychlejší než Heapsort a Mergesort, které jsou formálně rychlejší). **Maximální časová náročnost $O(N^2)$** ho však diskvalifikuje pro použití v kritických aplikacích.

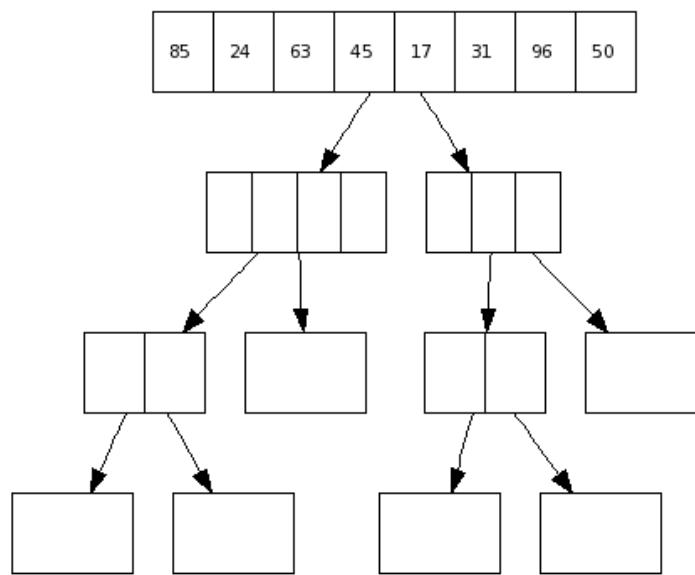
8.4.3.2 Algoritmus

```
public static int partition(int low, int high, keytype[] S) {
    int j = low;
    keytype pivot = S[high];
    for (int i = low; i < high; i++) {
        if (S[i] < pivot) {
            j++;
            swap(S[i], S[j]);
        }
    }
    swap(S[low], S[j]);
    return j;
}
```

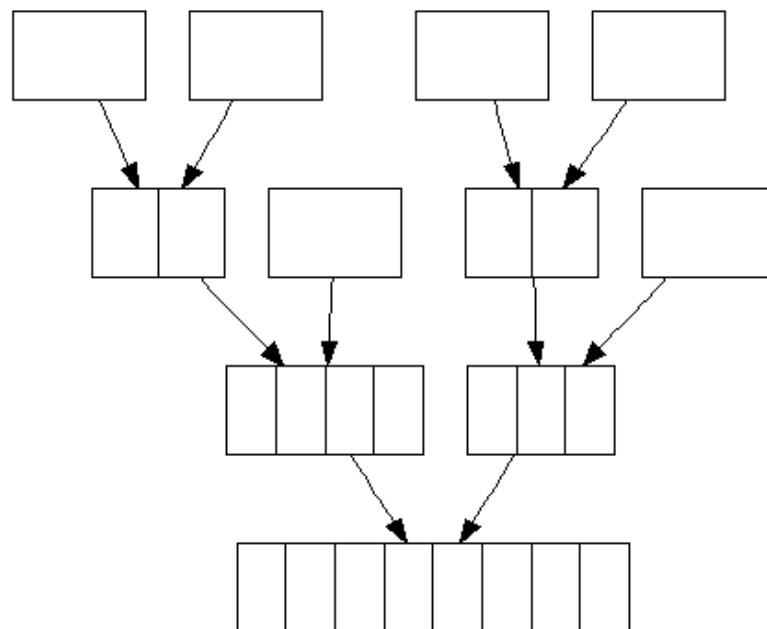
```
public static void quicksort(array, left, right)
    if right > left
        select a pivot index (e.g. pivotIndex = left)
        pivotNewIndex := partition(array, left, right, pivotIndex)
        quicksort(array, left, pivotNewIndex-1)
        quicksort(array, pivotNewIndex+1, right)
```

8.4.3.3 Příklad

(POZN: V materiálech přednášek naleznete animaci níže popsaného příkladu)

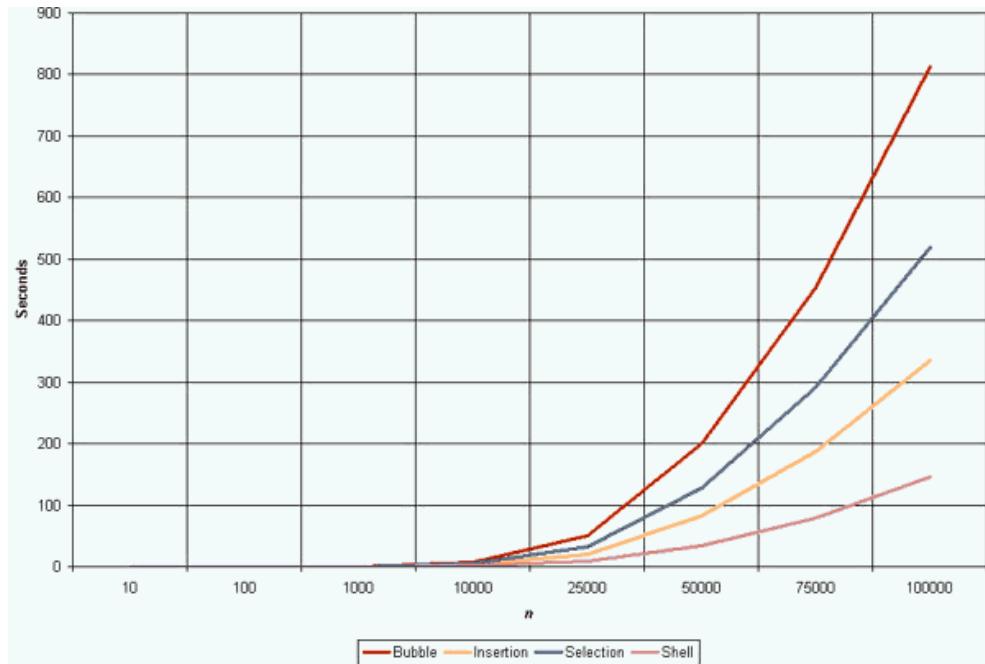


Obr. 116: První fáze algoritmu quick sort - volba pivota a rozklad.

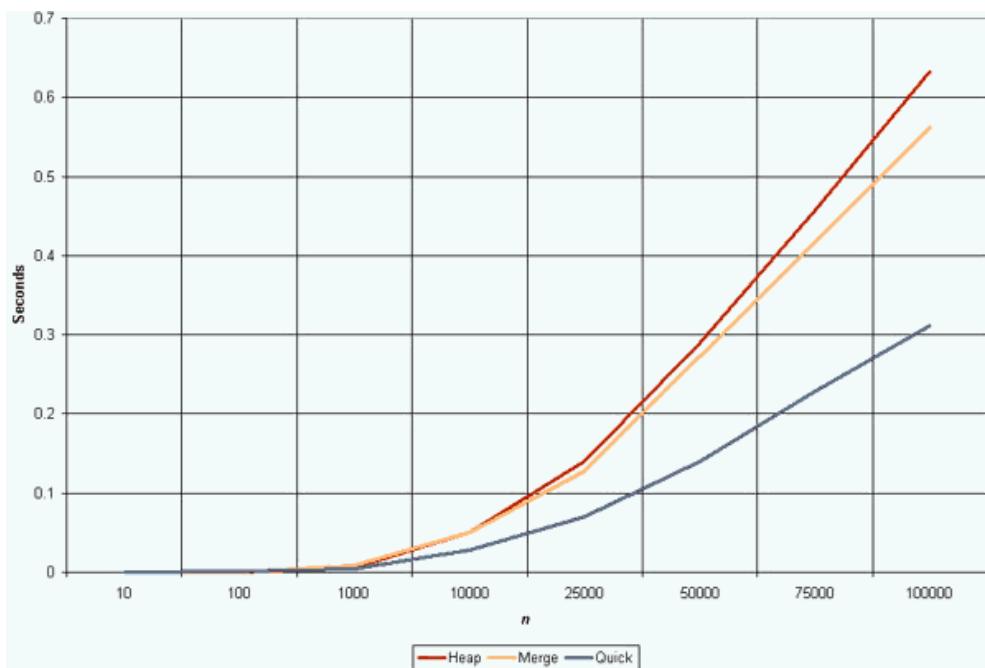


Obr. 117: Druhá fáze algoritmu quick sort - slučování posloupností.

8.5 Srovnání řadících algoritmů



Obr. 118: Srovnání jednoduchých $O(n^2)$ řadících algoritmů.



Obr. 119: Srovnání složitějších řadících algoritmů $O(n \log n)$.

Přestože na základě těchto grafů by se na první pohled mohlo zdát, že si vystačíme s algoritmy Quick sort popřípadě Shell sort, není tomu tak. Architektury výpočetních systémů se mění a postupně neustále vyvíjí. V budoucnu se předpokládá, že paralelní systému budou hrát stále větší roli a požadavky na řadící algoritmy se mohou a pravděpodobně i budou měnit.

8.6 Prostředky řazení jazyka JAVA

V jazyce java jsou k dispozici následující prostředky jak seřadit pole:

```
java.util.Arrays.sort(...)
```

či jinou kolekci prvků:

```
java.util.Collections.sort(...)
```

Řazení se proveden na základě porovnání prvků.

Stejně tak i v ostatních programovacích jazycích je řadící funkce v dnešní době již k dispozici přímo ve funkcích jazyka.

V rámci jazyka JAVA jsou implicitně používány algoritmy řazení Quick sort či Merge Sort [3]. Quick sort z důvodu vynikajících průměrných časových výsledků a Merge sort z důvodu stability a zaručené nejhorší složitosti v řádu $O(n \log n)$.

8.7 Literatura

- [1] BURGET, R. Hodnocení algoritmů, Přednáška MTIN.
- [2] Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)
- [3] Class Arrays, JAVA API documentation,Sun Microsystems LTD.,
[<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Arrays.html>](http://java.sun.com/j2se/1.4.2/docs/api/java/util/Arrays.html)

9 Algoritmy vyhledávání

Jsou tři hlavní důvodů, proč informační systémy existují: aby udržovaly informaci, aby na základě vložených informací byly schopny odvodit informaci novou a aby zpřístupnily informaci. Dále pak, aby byly schopny tyto procesy automatizovat. V rámci této kapitoly se budeme zabývat primárně třetí částí, tj. zpřístupnění neboli vyhledání informace. Jelikož existuje velké množství ADT [1] a každý má jinou topologii, jsou i metody vyhledávání informace v nich různé a každý má své výhody a každý bohužel i nevýhody.

9.1 Vyhledávání v neseřazeném poli

Vyhledávání v neseřazeném poli probíhá **vždy sekvenčně, prvek po prvku**. Na každé pozici v poli se kontrolují dva parametry 1) byl hledaný prvek nalezen 2) jedná se o poslední prvek pole a může být algoritmus ukončen. **V případě nenalezení prvku se postupuje na následující položku pole**. Při nalezení hledaného prvku algoritmus vyhledávání končí s tím, že předá informaci, kde byla požadovaná položka nalezena. V případě, že se jedná o poslední položku pole, algoritmus odpoví, že položka nebyla ve struktuře nalezena (viz Obr. 120).

1	10	4	19	7
---	----	---	----	---

Obr. 120: Příklad neseřazeneho pole o velikosti pěti prvků.

Z předchozího je patrné, že **časová složitost hledání v neseřazeneém poli je $O(n)$** , předpokládáme-li pole o délce n prvků. Výhodou je, že operace **vkládání nového prvku má konstantní $O(1)$ časovou složitost**.

Drobné vylepšení tohoto algoritmu představil Edsger Wybe Dijkstra (holandský vědec). S použitím tzv. zarážky na konci pole(viz Obr. 121), kam vložíme hodnotu hledaného prvku, nemusíme kontrolovat, zda-li jsme narazili na konec. Hodnota bude nalezena vždy a pouze jedenkrát za celý algoritmus je nutné ověřit, zdali nalezená hodnota není právě zarážkou (tj. hledaná hodnota v poli neexistuje). Časová složitost je opět v třídě $O(n)$, avšak v praktickém nasazení dává lepší výsledky než předchozí metody.

1	10	4	19	7	?
---	----	---	----	---	---

Obr. 121: Příklad Disjkstrovy varianty vyhledávání v poli. Velikost pole je rovna 5 a na konci pole je vyhrazeno místo pro zarážku, tj. hodnotu hledaného klíče.

9.2 Vyhledávání v seřazeneém poli

V případě seřazeneho pole jsme schopni **s časovou složitostí $\Theta(\frac{n}{2})$ [5]⁶** rozhodnout, zdali **hledaná položka neexistuje v seznamu** (viz Obr. 122). **Pokud překročíme jistou hodnotu, vím,**

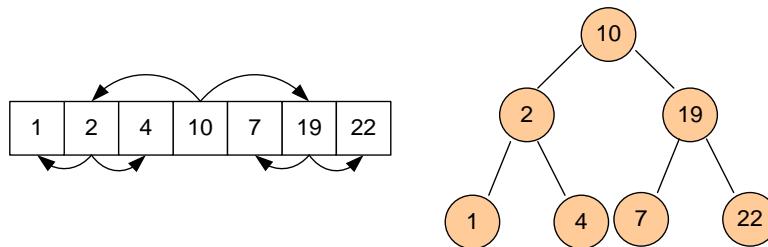
⁶ Notace Θ (Theta) je v tomto případě použita proto, protože vyšetřujeme průměrný čas trvání a nikoli nejhorší možný čas trvání.

že hodnota se v poli již nemůže vyskytnout a algoritmus může být ukončen. Avšak v případě úspěšného hledání se nejedná o naprostou žádnou úsporu času. Navíc operace vkládání prvku má časovou složitost: $O(n)$, oproti konstantní v předchozím případě.

1	4	10	7	19
---	---	----	---	----

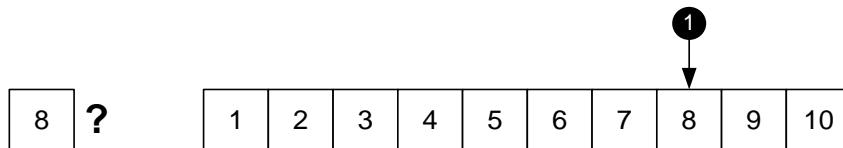
Obr. 122: Příklad seřazeného pole a vyhledávání.

V případě pole si lze ADT pole představit jako ADT binární strom a postupovat **metodou půlení intervalu**. V tomto případě dosáhneme významné optimalizace $O(\log_2 n)$ (viz Obr. 123) a lineární datová struktura se bude chovat jako binární strom.



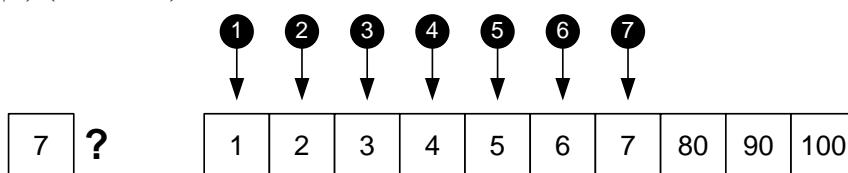
Obr. 123: Příklad reprezentace ADT pole jako ADT strom. K vyhledání jakékoli hodnoty klíče je zapotřebí max. 3 porovnání.

Další variantou je **interpolační vyhledávání**. Metoda funguje podobně jako binární půlení s tím rozdílem, že pole se nepůlí přesně v polovině, ale pokoušíme se interpolovat pravděpodobný výskyt hledaného prvku lineární interpolací. Algoritmus funguje nejlépe, pokud jsou hodnoty prvku v poli rovnoměrně rozdělené a samozřejmě i seřazené. V takovém případě můžeme dosáhnout výsledku časové složitosti $O(\log \log_2 n)$ (viz Obr. 124).



Obr. 124: Příklad interpolační metody. Díky rovnoměrnému rozložení mohu odhadnout pozici hledaného klíče v poli na první pokus.

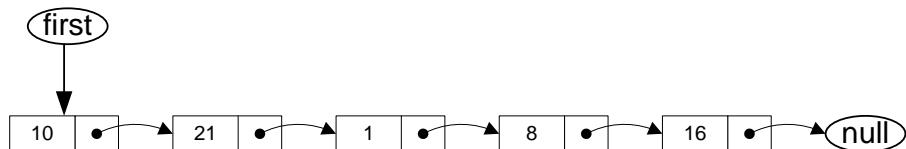
Pokud podmínka rovnoměrného rozložení není zaručena, interpolační odhad nebude přesný a nesprávný odhad může v krajním případě degradovat výpočetní sílu algoritmu na lineární procházení $O(n)$ (Obr. 125).



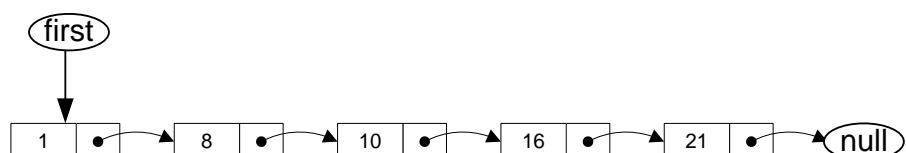
Obr. 125: Příklad interpolační metody. Z důvodu nerovnoměrného rozdělení hodnot klíčů bude hledání prvku interpolováno chybně a namísto jediného kroku bude nutných kroků 7. V takovém případě by metoda půlení intervalu dosahovala lepších výsledků.

9.3 Vyhledávání v ADT seznam

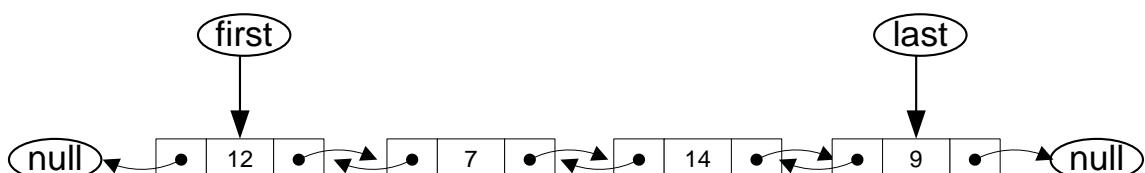
Vyhledávání klíče v ADT seznam je identické s vyhledáváním v ADT pole. Jediný rozdíl oproti ADT pole je nemožnost efektivně reprezentovat ADT seznam jako binární strom. Z toho důvodu také stojí za zvážení, zdali je výhodné jeho použití a neinvestovat do plýtvání pamětí (viz Obr. 126, Obr. 127, Obr. 128, Obr. 129).



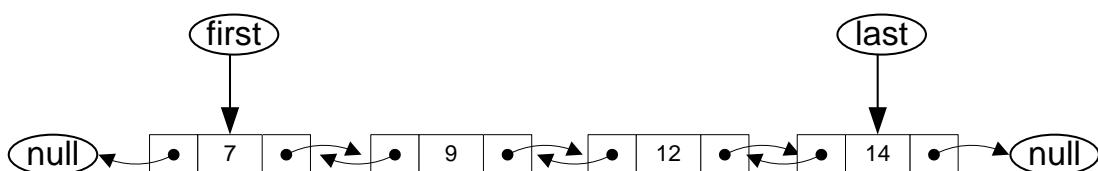
Obr. 126: Příklad neseřazeného jednosměrně vázaného lineárního seznamu.



Obr. 127: Příklad seřazeného jednosměrně vázaného lineárního seznamu.



Obr. 128: Příklad neseřazeného obousměrně vázaného lineárního seznamu.

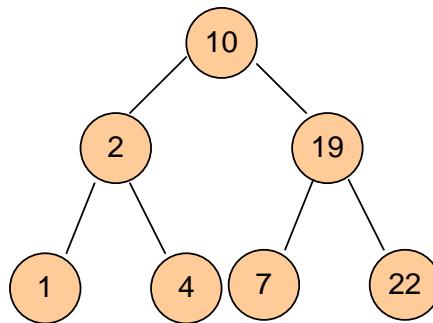


Obr. 129: Příklad seřazeného obousměrně vázaného lineárního seznamu.

Výhodou ADT seznam je operace vkládání. V případě ADT seznam není nutné přesunovat ostatní prvky pole a z toho důvodu se jedná o významnou úsporu z hlediska průměrné časové náročnosti.

9.4 Vyhledávání v ADT binární vyhledávací strom

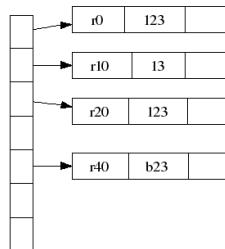
V případě neseřazených položek v ADT strom by bylo nutné, obdobně jako v případě neseřazeného pole, projít všechny položky. Pokud použijeme ADT binární vyhledávací strom, dosáhneme v nejhorším případě $O(\log_2 n)$ počtu kroků v případě ideálně vyváženého stromu, kde n reprezentuje počet prvků ve struktuře. **V praxi se bude jednat o ADT AVL stromy, či ADT Red-Black stromy,** které zaručují o málo horší vlastnosti, nicméně i přesto zaručují vyváženosť binárního stromu a současně **udržují složitost operaci vkládání prvku na přijatelné hodnotě.** Na Obr. 130 je vyobrazen příklad ADT binární strom.



Obr. 130: Příklad ADT binárního stromu.

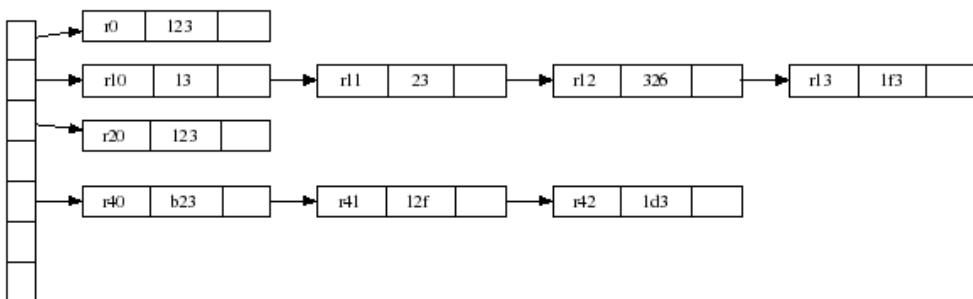
9.5 Vyhledávání v ADT hashovaní tabulka

V případě hashovaní tabulky závisí časová složitost vyhledání prvku na počtu klíčů a počtu hodnot. V případě dostatečného počtu klíčů a relativně k tomu málo hodnotami v pole je časová složitost vyhledání prvku v poli konstantní $O(1)$ (viz Obr. 131).



Obr. 131: Příklad hashovací tabulky s dostatečným počtem klíčů.

V případě relativně velkému počtu vložených hodnot vzhledem k počtu dostupných klíčů může být hashovaní tabulka v extrémním případě degradována na ADT lineární seznam, tedy $O(n)$.

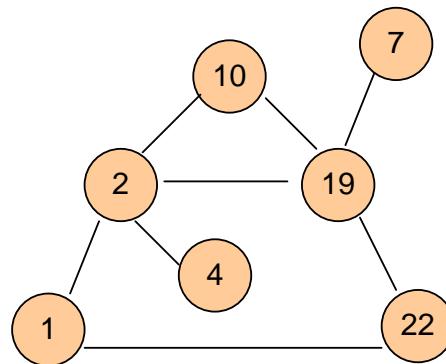


Obr. 132: Příklad hashovací tabulky s velkým počte vložených hodnot vzhledem k počtu klíčů.

9.6 Vyhledávání v ADT graf

Popisu ADT Graf i metodám vyhledávání budou věnovány samostatné přednášky. Bližší informace viz [3], [4].

Pro představu struktury ADT graf viz Obr. 133. Na rozdíl od stromů mohou obsahovat cykly, nemusejí být hierarchické, mohou být orientované i neorientované.



Obr. 133: Příklad neorientovaného ADT graf.

9.7 Literatura

- [1] BURGET, R. Abstraktní datové typy 1, Přednáška MTIN.
- [2] BURGET, R. Abstraktní datové typy 2, Přednáška MTIN.
- [3] BURGET, R. Abstraktní datový typ graf, Přednáška MTIN.
- [4] BURGET, R. Metody vyhledávání v ADT graf, Přednáška MTIN.
- [5] BURGET, R. Hodnocení algoritmů, Přednáška MTIN.

10 Optimalizace

10.1 Metody přímého vyhledávání

10.1.1 Metoda Monte Carlo

Nebo též slepý algoritmus. Ve zkratce řečeno – náhodně zkouší různá místa a z této množiny vybírá to nejlepší. Algoritmus je velice jednoduchý a dal by se vyjádřit takto:

$t = 0$

Vytvoř P , ohodnoť P

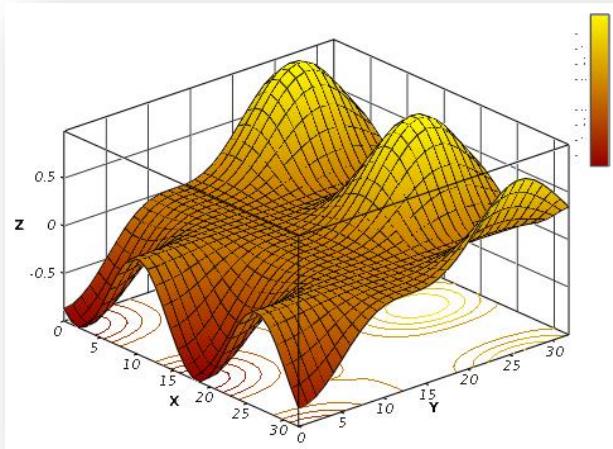
while (not zastavovací podmínka) {

$t = t + 1$

Náhodně vytvoř N

Pokud je N lepší, pak nahradí P

}



Obr. 134: 2D funkce (vstupem jsou parametry x, y), kde výsledná hodnota je značena hodnotou y .

10.1.2 Horolezecký algoritmus

Horolezecký algoritmus je variantou předchozího. Rozdíl oproti metodě Monte Carlo je v tom, že nevolí stavy zcela náhodně, ale hledá

$t = 0$

Vytvoř P , ohodnoť P

while (not zastavovací podmínka) {

$t = t + 1$

Náhodně nebo systematicky vytvoř N_i v okolí P , ohodnoť N_i

Nejlepší N_i nahradí P

}

10.2 Genetické algoritmy

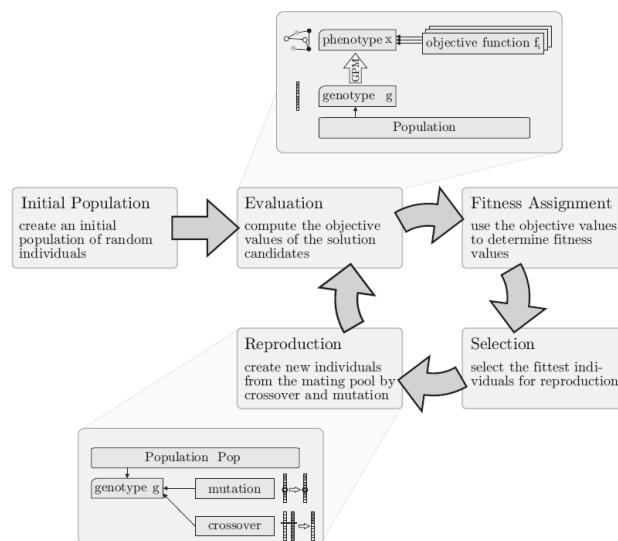
Genetický algoritmus (GA) je heuristický postup, který se snaží aplikací principů evoluční biologie nalézt řešení složitých problémů, pro které je obtížné nalézt přesný algoritmus. Genetické algoritmy patří do skupiny evolučních algoritmů a jako takové se

opírají o principy evolučních procesů známé z biologie. Patří mezi ně dědičnost, mutace, přirozený výběr a křížení.

Obecný genetický evoluční algoritmus je založen na principu, jak je naznačeno na obrázku níže. Nejprve dojde k inicializaci náhodné populace, poté probíhá výběr jedinců, následuje reprodukce (generování nových potomků) a pokud je dosaženo požadované přesnosti, je algoritmus ukončen.



Obr. 135: Inicializace chromozomu



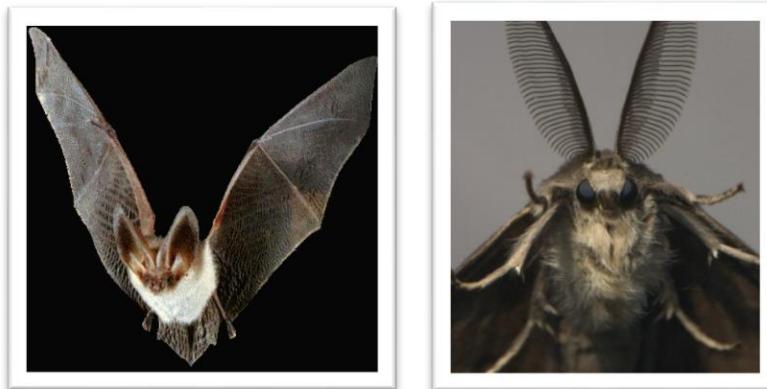
Obr. 136: Obecný genetický algoritmus.

Genetické algoritmy (GA) – začínají s jedinci, kteří jsou reprezentováni jako řetězce binárních hodnot.

10.2.1 Demonstrační efektivity genetických algoritmů

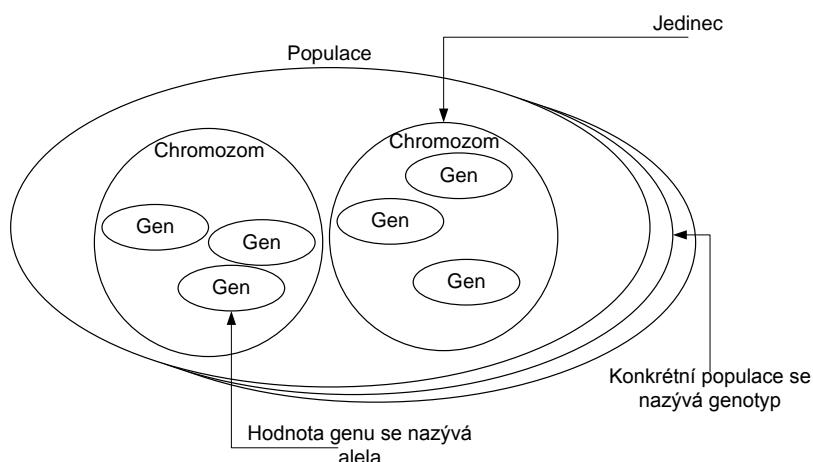
- netopýři mají sonar, kterým hledají můry, sonar je sám o sobě komplikovaný

- můry vyvinuly měkké pokrytí těla, které absorbuje netopýří vysílání
- netopýři přešli na nové frekvence
- můry přišly s novým pokrytím a s „rušičkou“ (vlastní signál, který interferuje s netopýřím)
- netopýři přišli s novými leteckými manévrovy a naučily se vypínat sonar (čímž dělají rušení méně efektivním)



Obr. 137: Ilustrační foto, netopýr vs. můra

10.2.2 Základní pojmy



Obr. 138: Názvosloví a jejich vzájemné vztahy

10.2.3 Mutace

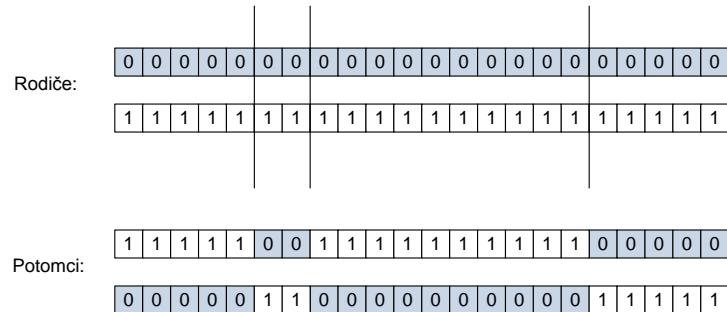
Mutace jsou vhodné zejména pro ten případ, kdy konvergujeme k nějakému řešení, které uvázne v nějakém lokálním optimu. Díky mutaci lze zanést do chromozomu takovou „chybu“, která je schopna překonat lokální optima a hledat i v jiných oblastech. Jeho nevýhoda je, že většinou nevede ke zlepšení a proto se také používá s menší

pravděpodobností. Velice často se používá jen s velice malou pravděpodobností, aby zbytečně neměl podstatný vliv na zdokonalování současné populace.

10.2.4 Operace křížení

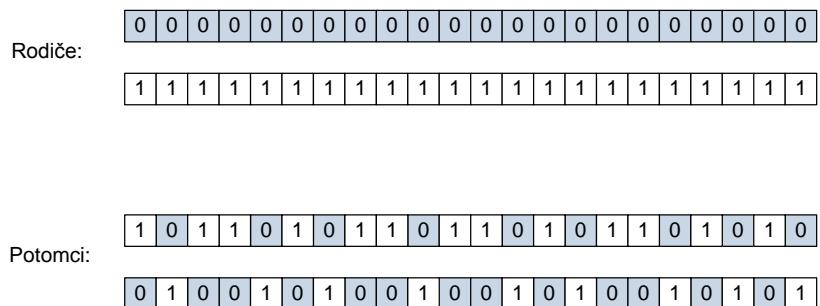
Při operaci křížení dochází ke kombinaci dvou jedinců (reprezentovanými chromozomem).

10.2.5 N-Bodové křížení (N-point crossover)



Obr. 139: Několika bodové křížení genomu. Genom je rozdělen na několik částí a u nich pak dochází k záměnám.

10.2.6 Rovnoměrné křížení (Uniform crossover)



Obr. 140: Rovnoměrné křížení genomu, kde jsou rodiče a jejich potomci

10.2.7 Křížení vs. Mutace

Nasazení křížení či mutace závisí na konkrétním typu problému, který má genetický algoritmus řešit. **Obecně se ale považuje za optimální nasazení jak křížení tak mutace současně a to z toho důvodu, že každý z nich má jinou roli.** Dále platí, že zatímco křížení může být nasazeno samostatně, pouhá mutace fungovat nebude. Jednalo by se o neinformovanou metodu náhodného hledání a tomu by také odpovídala úspěšnost a časové nároky algoritmu. **Úkolem křížení je posunout řešení blíže směrem k předpokládanému řešení a kombinuje výhody dvou potomků.** Naopak mutace vytváří drobné odchylky a zanáší do řešení náhodnost, neboli novou infomaci. Pokud mutace převážila efekt křížení, eliminoval by konvergenci k řešení a algoritmus by nekonvergoval k řešení. Naopak, pokud bude mutace velmi malá, popřípadě žádná, je více pravděpodobné, že algoritmus uvázne v lokálním minimu a globální optimum se nepodaří nalézt. To ale bohužel v případě genetických algoritmů hrozí vždy a je k tomu zapotřebí náhoda.

Jakmile proběhne operace křížení, je zapotřebí odstranit nepovedené geny. K tomu slouží tzv. fitness funkce. Existuje několik přístupů Soutěžení, Přežití.

V případě soutěžení se vyhodnocují geny na základě statistik celé skupiny. Zde nastává problém s paralelizací na víceprocesorových architekturách. Současně musí existovat některá fitness funkce, která vyhodnocuje jednotlivé vlastnosti.

Algoritmus přežití je přítomna v přírodě. Zde rozhoduje o přežití schopnost přežití a schopnost reprodukce. Výběr bývá volen buď s náhodným výběrem „úmrtí“ genu (nedoporučuje se). Anebo geny jsou založeny na ADT FIFO a odstraněny jsou nejstarší potomci. Další přístup je na základě fitness funkce, kde po překročení jistého hlídaného parametru dojde k zániku jedince.

10.2.8 Přirozený výběr

Přirozený výběr je proces, který vybírá ty chromozomy, které mají přežít do další generace. Existuje mnoho variant. Mezi nejběžněji používané patří klasický přirozený výběr, výběr dle nejlepšího chromozomu, post výběr, prahový výběr, soutěživý výběr či výběr způsobem rulety.

K tomu, aby mohly být chromozomy vzájemně porovnávány, musí implementovat tzv. fitness funkci. Jedná se o funkci, která je navržena vždy pro každý druh genu a na základě jeho stavby se pokouší o jeho hodnocení a tedy i porovnávání.

Výhodou genetických algoritmů je skutečnost, že nemusíme znát způsob, jak se dostat k požadovanému cíli – tj. kvalitnímu sub-optimálnímu v ideálním případě dokonce optimálnímu genu. Jediné co nám postačuje, že víme jak jedince s různým genetickým materiélem vzájemně porovnat, tedy fitness funkci a umíme ji navrhnut.

10.2.8.1 Výběr dle nejlepšího chromozomu

Výběr dle nejlepšího chromozomu probíhá na základě seřazení chromozomů dle hodnoty jejich fitness funkce. Z tohoto seznamu je poté vzato N nejlepších jedinců, kteří jsou vyvoleni pro pokračování v další evoluci.

10.2.8.2 Prahový výběr

Prahový výběr, neboli také výběr dle prahové hodnoty jest podobný předchozí variantě. Jediný rozdíl je skutečnost, že počet potomků není dán konstantním počtem, ale jejich kvalitou. V tomto případě se do druhého kroku evoluce dostane pouze ten jedinec, jehož fitness funkce přesahuje určitou hranici.

10.2.8.3 Původní přirozený výběr

Přirozený výběr vychází z předpokládaného modelu v biologii a je podobný výběru dle nejlepšího chromozomu s jistou mírou náhody. Vybírá nejlepších N genů, nicméně není zaručeno, že pořadí bude striktně doručeno a že se do další evoluce nedostane i gen nižší kvality, zatímco gen vyšší kvality vypadne.

Na první pohled se může zdát tato náhodnost jako neužitečná, nicméně vezměme si inspiraci ve hře šachu. Pokud obětujeme v jednom tahu nejsilnější figuru dámu, může se to zdát za špatný tah ale to jen pokud nevidíme do budoucnosti. V následujícím tahu můžeme mít

možnost dát šach-mat a tím dosáhnout vítězství. Jde tedy o vhodné zvolení kompromisu mezi náhodou a vztí v úvahu i geny, které se zdají být neperspektivní.

10.2.8.4 Post výběr

Post výběr porovnává svojí kvalitu až na základě kvality genetického materiálu, který předá svým potomkům. Neboli, jinými slovy – ohodnocuje svoji kvalitu až po aplikaci genetických operátorů. Obdobu v biologii bychom mohli nalézt ve společenstvech, kde kvalita potomků rozhoduje o přežití svých rodičů a současně celého společenstva.

Nutno ještě brát na vědomí, že tento druh bude klást výrazně vyšší nároky jak na paměť počítače, tak na jeho výkonnost.

10.2.8.5 Soutěživý výběr

Soutěživý výběr je založen na pořádání turnajů mezi jednotlivými geny. Ty, které dosáhnou nejlepších výsledků při řešení úkolu, jsou upřednostněny pro generování následujícího genotypu.

10.2.8.6 Výběr pomocí rulety

Ruleta je založena na poměrně jednoduchém algoritmu. Jakmile je gen přidán, zabere tolik počtu míst na ruletě (náhodně), který odpovídá hodnotě jejich fitness funkce. Jakmile ruleta označí jisté místo, jsou vybrány ty geny, které mají toto místo rulety obsazeno.

10.2.9 Genetické operátory

Ve fázi vytváření další generace potomků existuje mnoho různých operací, které mohou svým charakterem ovlivnit rychlosť konvergence popřípadě kvalitu nalezeného řešení. Jsou to například gausovský operátor, mutační operátor, operátor křížení, operátor dvoucestného křížení, hladové křížení a další.

10.2.9.1 Hladové křížení (greedy crossover)

Hladové křížení je specifický typ křížení a může být použit tehdy, jsou-li splněny následující dvě podmínky:

- Veškeré geny v chromozomu jsou různé
- Množina genů v chromozomu pro oba geny je identická a pouze jejich pořadí v chromozomu se může lišit.

Po dobu dokud jsou tyto podmínky splněny, mohou být stále dokola a dokola prováděny a opakovány.

Příklad: Určete rozložení padesáti polygonů tak, aby co nejvíce odpovídaly předloze obrázku. Barva těchto polygonů může být volena libovolně.

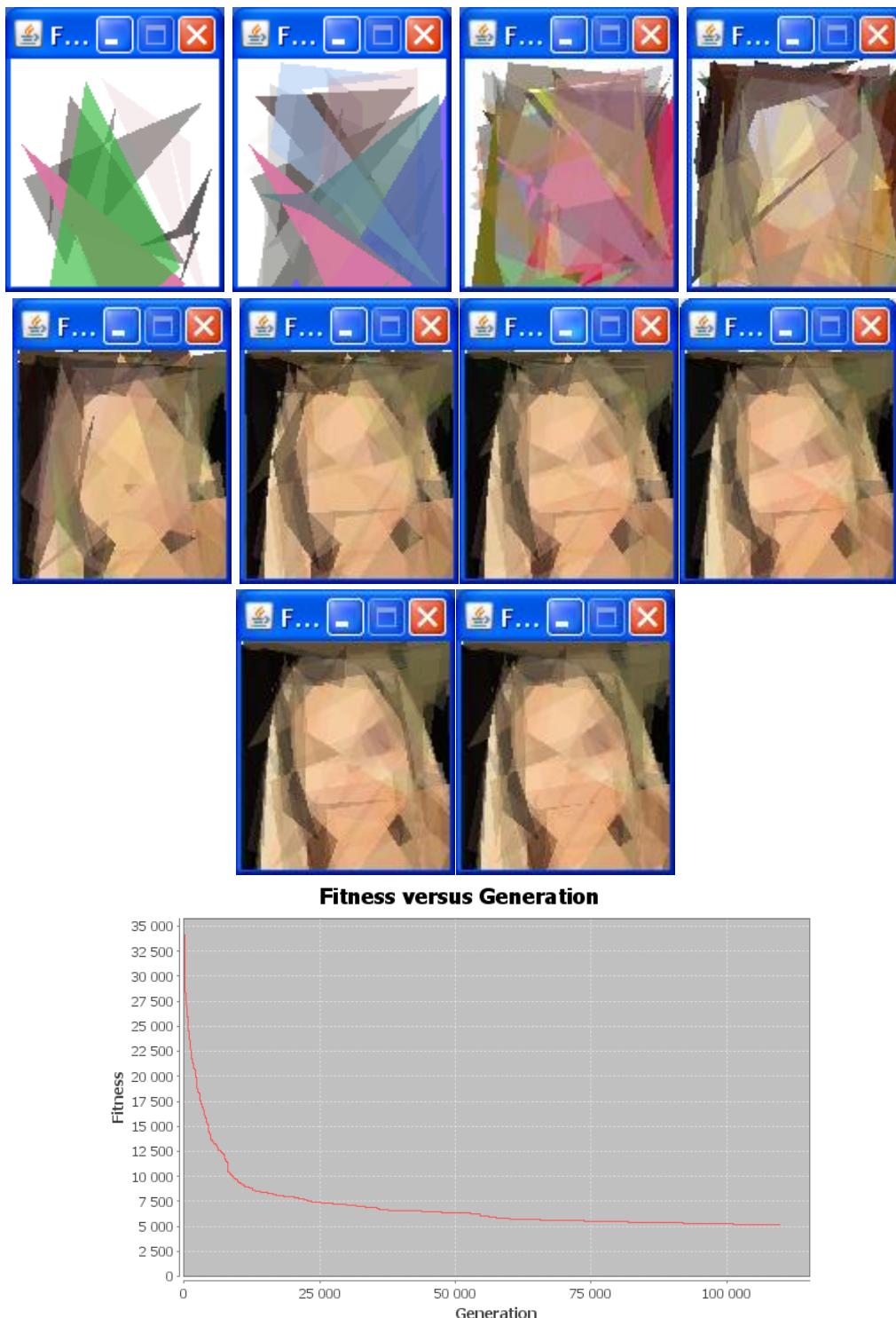
Řešení s pomocí genetických algoritmů lze v tomto případě provést tak, že v chromozomu zakódujeme polohy jednotlivých všech vrcholů padesáti polygonů a současně barvu tohoto polygonu. Jako počáteční populaci vybereme zcela náhodné polohy vrcholů těchto polynomů a



Obr. 141: Miss World, Tatána Kuchařová, originál obrázku [5], který se pokoušíme vytvořit s pomocí evolučních algoritmů .

vytvoříme tak počáteční populaci. Fitness funkce je nastavena tak, aby upřednostňovala chromozomy, které jsou více podobné cílovému řešení.

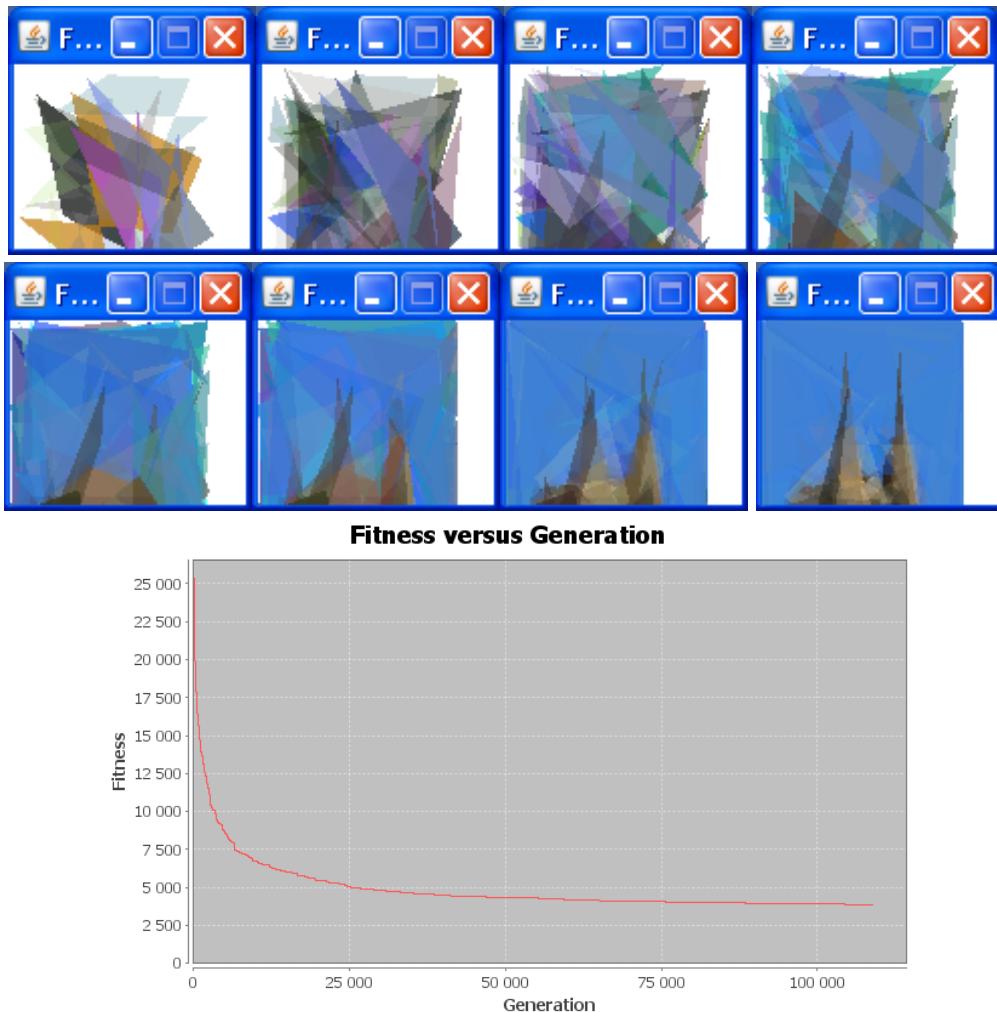
Poznámka: Zcela jistě by bylo možné nakreslit obraz s pomocí analytického řešení, které by zřejmě používalo analýzu obrazu a určilo by optimální rozložení ploch. Avšak tento přístup vyžaduje porozumění dané problematice. Náš přístup umožňuje specifikovat jen vstup a požadovaný výstup a nechť se s tím počítač vypořádá.



Jak je patrné, tak z počátku generace dochází k výrazným zefektivněním. To se s postupem doby mění. Nevýhodou evolučních pravidel je bezesporu jejich výkon – doba výpočtu tohoto případu byla přibližně tři hodiny.



Příklad: Podívejme se také na zcela jiný typ obrázku s odlišným složením barev a odlišným typem tvarů. První z obrázků je originál, následují postupné evoluce.



Jak je z grafu vývoje fitness funkce patrné, i nadále by docházelo ke zpřesněním, nicméně rychlosť konvergencie by se stále zpomalovala.

10.2.10 Uplatnění

- Plánování úloh a jejich optimální [6], [7], [8] , [9] , [10], [11] , [12]
- Plánování řetězce dodavatelů za účelem minimalizace ceny
- Chemie – vytváření nových sloučenin a posuzování jejich vlastností [13], [14], [15], [16], [17], [18]
- Lékařství [19], [20], [21], [22]
- Dolování informací – skrytých vzorů z historických dat (finanční trhy)

- Ekonomie a finančnictví – předpověď bankrotu zdánlivě zdravé společnosti
- Optimalizace protokolů
- Návrh obvodů a jejich optimalizace

10.3 Genetické programování

Jak již bylo řečeno, genetické algoritmy (GA) je založena na principu počátečních jedinců, kteří jsou reprezentováni jako řetězce binárních hodnot. Další přístupem mohou být evoluční strategie (ES), které obecně pracují nad oborem reálných hodnot (reálných čísel). Obě tyto techniky se po léta vyvívají, a přestože obě vzniky jako samostatné obory dnes se hranice mezi nimi víceméně prolínají. Zpravidla se tyto techniky používají k optimalizaci parametrů. Příklad – hledání optimálního rozložení stanic v síti, známe-li pouze jejich RTT vzdálenosti. Hledání sub-optimálního řešení problému obchodního cestujícího, atd.

Genetické programování se na rozdíl od GA a ES vyznačuje tím, že modifikují symboly, které představují program. Tyto symboly mohou mít proměnlivou délku a samozřejmě proměnlivý tvar. V obecnějším smyslu můžeme genetické programování dokonce považovat za způsob strojového učení, tedy vědního oboru, který se zabývá výzkumem algoritmů schopných se učit na základě předchozí zkušenosti. Podobnost je vidět zejména u algoritmů, které stály na počátku této vědy.

Genetické programování je ve svém principu schopno zastoupit ostatní techniky používané pro strojové učení (jako například neuronové sítě). Nicméně dnes se v reálném nasazení používají téměř výhradně pro data mining a optimalizační problémy.

Genetické programování lze použít k tomu, aby se vypořádal s širokou škálou problémů, počínaje satelitními ovladačů, jako odvětví umělé inteligence, či řešení složitých problémů, které závisí na spoustě parametrů a vzájemně se ovlivňují. Genetické algoritmy jsou obecně daleko mocnější než genetické algoritmy. Zatímco v případě genetických algoritmů bylo výstupem optimální parametry, v případě genetického programování to jsou programy. Je to tedy počátek počítačových programů, které píšou počítačové programy.

Genetické programování dává výborné výsledky zejména pro takové problémy, pro které neexistuje ideální řešení. Jako příklad může být problém řízení automobilu (ramene robota) – některá řešení řídí bezpečně, ale bohužel s velkými nároky na čas. Na druhou stranu jiné dělají dobré výsledky z hlediska času, ale řídí s velkou dávkou rizika nehody. V těchto případech genetické algoritmy naleznou řešení, která berou v potaz velké množství proměnných a okolních faktorů a bylo by velice obtížné nalézt analytické řešení.

10.3.1 Základní princip

Základní princip fungování je podobný jako v případě GA. Nejprve se vytvoří počáteční populace s náhodným složením operátorů a terminálů. V následujícím kroku jsou spuštěny jednotlivé programy a je jim přiřazena hodnota fitness na základě toho, jak dobře řeší problém. V třetím kroku dojde ke:

- Kopírování nejlepších existujících programů
- Vytváření nových programů na základě mutací

- **Vytváření nových programů na základě křížení existujících programů.**

Řešením je potom vybrán takový program, který ze všech populací dával nejlepší výsledky.

10.3.2 Zvyšování výpočetní výkonnosti

- Pozastavení běhu, čtení a možnost změny parametrů
- Paralelizace – dává dokonce vyšší urychlení nežli jen lineární (**Paralelní Genetické Programování – může běžet i v jednom procesu/vlákně**)
- Paralelní ohodnocování fitness funkce (několik procesorů paralelně)

10.3.3 Automaticky definované funkce

Většina dnes známých programovacích jazyků jsou tzv. procedurální (C, C++, JAVA, ...). Oproti tomu stojí programy vyvíjené s pomocí genetického programování, které jsou založeny na principu programování funkcionálního (Lisp, Scheme, Lambda-kalkul, ...). Pro lepší představu, níže je kód, zapsaný v programovacím jazyce Lisp:

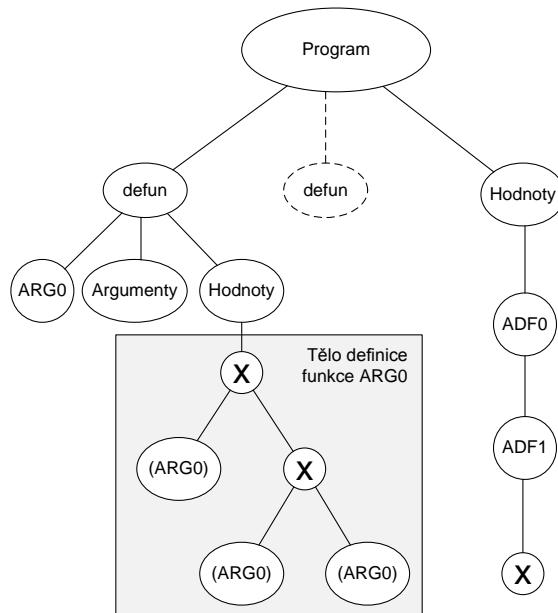
```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (- n 1)))))
```

Jejich výhodou je to, že vychází z matematické definice programu, kde se program vypočítává stejně jako matematické funkce. ADF se snaží překonat tyto rozdíly a vytvořit cestu od funkcionálního programování k procedurálnímu.

S použitím ADF se, stejně jako u klasických GP programů, jedná o stromovou strukturu. Avšak v případě použití ADF se tento strom dělí do dvou typů větví:

- Větev, která **tvoří výsledek**. Pomocí ní je program ohodnocován s využitím fitness funkce.
- Větev, která **definuje funkce**. Zde jsou popsány jednotlivé ADF funkce (může být jedna či více)

Tyto větve jsou poté podobné programování jaké známe v procedurálních programovacích jazycích (C, C++, JAVA, Pascal, ...). Zde metoda **main** představuje věti tvořící výsledek, zatímco **definiční část** kódu představuje větev definující ADF funkce. Při evolučním vývoji programů se na vývoji podílejí současně obě tyto části. Výstupem GP programu tak potom může být modulárně členěný kód tvořený funkciemi (viz Obr. 142). Zde je vpravo větev vyhodnocující a vlevo ADF funkce počínající kořenovým uzlem defun. Pokud je těchto ADF funkcí více, je zde těchto defun definicí také více (naznačeno čárkovaně). V případě, že má program pouze jednu výstupní hodnotu, tak větev tvořící výsledek se vyskytuje právě jednou. V případě, že návratová hodnota programu je pole, je může se vyskytovat i více těchto větví, kde každá z nich se vyhodnocuje zvlášť. Zkratka ADF0 v obrázku představuje název funkce. Následuje definice argumentů této funkce. V poslední části jsou samotné hodnoty – tj. definice těla programu funkce.



Obr. 142: Příklad ADF stromu. Klíčové slovo "defun " označuje definici funkce.

Příklad: Jako příklad nám v tomto případě může posloužit hra Robocode [4]. Jedná se o projekt vyvinutý společností IBM, kde je simulován souboj tanků. Každý z tanků má možnost se pohybovat dopředu a dozadu, otáčet, otáčet hlaveň a v neposlední řadě otáčet radar, díky kterému je schopen detekovat ostatní tanky. Nejedná se však o hru interaktivní, ale každý má možnost si naprogramovat chování každého z tanků a ten poté vpustit do bitvy. Vyhrává vždy jen ten nejlepší (viz Obr. 143).



Obr. 143: Příklad simulace boje u hry Robocode.

Jednou z možností je samozřejmě začít programovat „ručně“ a mnohdy takové výsledky mohou být vynikající. Další cestou je napsat kód s použitím genetického programování, který se na základě principů evoluce „vyšlechtí“ v „neporazitelné monstrum“ ☺.

10.4 Literatura

- [1] Jyh-Shing Roger Jang, Chuen-Tsai Sun, Eiji Mizutani. Neuro-Fuzzy and Soft Computing. Prentice Hall, 1997, ISBN: 0-13-261066-3
- [2] Jyh-Shing Roger Jang, Neuro-Fuzzy and Soft Computing, <<http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=2173>>
- [3] ROBOCODE, <<http://robocode.sourceforge.net/>>
- [4] Robocode, <<http://en.wikipedia.org/wiki/Robocode>>
- [5] Tat'ána Kuchařová, Miss Czech, <http://www.missczech.com/2008/mcr2006.php>
- [6] Lawrence Davis, editor. Handbook of Genetic Algorithms. Thomson Publishing Group / Van Nostrand Reinhold Company, New York, USA, January 1991. ISBN: 0-4420-0173-8, 978-0-44200-173-5.
- [7] Randy L. Haupt and Sue Ellen Haupt. Practical Genetic Algorithms. Wiley-Interscience, December 19, 1997. ISBN: 0-4711-8873-5, 978-0-47118-873-5, 978-0-47145-565-3. Second edition: June 2004, partly online available at http://books.google.de/books?id=WO64iOg_abIC
- [8] Mitsuo Gen and Runwei Cheng. Genetic Algorithms and Engineering Design. Wiley-Interscience, January 7, 1997. ISBN: 0-4711-2741-8, 978-0-47112-741-3. Partly online available at <http://books.google.de/books?id=MCHCaJAHFJAC> [accessed 2008-04-01].
- [9] Lance D. Chambers, editor. Practical Handbook of Genetic Algorithms: Applications, volume I of Practical Handbook of Genetic Algorithms. Chapman & Hall/CRC Press, Inc., Boca Raton, FL, USA, 2nd ed: 2000 edition, 1995. ISBN: 0-8493-2519-6, 1-5848-8240-9, 978-1-58488-240-4.
- [10] Lance D. Chambers, editor. Practical Handbook of Genetic Algorithms: New Frontiers, volume II of Practical Handbook of Genetic Algorithms. CRC Press, Inc., Boca Raton, FL, USA, 1995. ISBN: 0-8493-2529-3. Partly online available at <http://books.google.de/books?id=9RCE3pgj9K4C> [accessed 2008-07-19].
- [11] Lance D. Chambers, editor. Practical Handbook of Genetic Algorithms: Complex Coding Systems, volume III of Practical Handbook of Genetic Algorithms. CRC Press, Inc., Boca Raton, FL, USA, December 23, 1998. ISBN: 0-8493-2539-0, 978-0-84932-539-7.
- [12] Mitsuo Gen and Runwei Chen. Genetic Algorithms (Engineering Design and Automation). Wiley Series in Engineering Design and Automation. John Wiley and Sons Ltd, February 2001. ISBN: 978-0-47131-531-5.

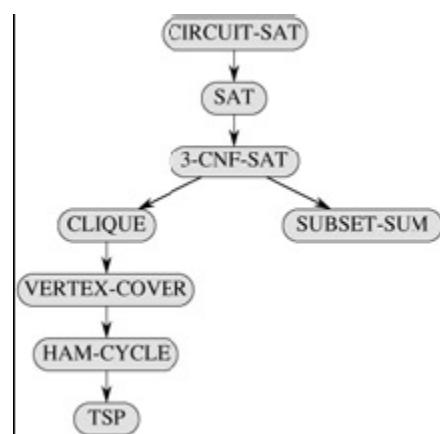
- [13] Erick Cant' u-Paz. Efficient and Accurate Parallel Genetic Algorithms, volume 1 of Genetic Algorithms and Evolutionary Computation. Springer, December 15, 2000. ISBN: 978-0-79237-221-9, 0-7923-7221-2.
- [14] Jochen Heistermann. Genetische Algorithmen. Theorie und Praxis evolutionärer Optimierung. Teubner Verlag, 1994. ISBN: 3-8154-2057-1, 978-3-81542-057-7.
- [15] Eberhard Schoneburg, Frank Heinzmann, and Sven Feddersen. Genetische Algorithmen und Evolutionsstrategien. Addison Wesley Verlag, 1993. ISBN: 3-8931-9493-2, 978-3-89319-493-3.
- [16] Tomasz Dominik Gwiazda. Crossover for single-objective numerical optimization problems, volume 1 of Genetic Algorithms Reference. Lomianki, e-book: Tomasz Dominik Gwiazda, May 2006. ISBN: 8-3923-9583-2.
- [17] Robert Schaefer and Henryk Telega. Foundations of Global Genetic Optimization, volume 74/2007 of Studies in Computational Intelligence. Springer Berlin / Heidelberg, 2007. ISBN: 978-3-54073-191-7. doi:10.1007/978-3-540-73192-4. Series editor: Janusz Kacprzyk.
- [18] Charles L. Karr and L. Michael Freeman, editors. Industrial Applications of Genetic Algorithms. International Series on Computational Intelligence. CRC Press, Boca Raton, FL, December 29, 1998. ISBN: 0-8493-9801-0, 978-0-84939-801-8.
- [19] Sanza Kazadi. Conjugate Schema and Basis Representation of Crossover and Mutation. *Evolutionary Computation*, 6(2):129–160, September 1998. Online available at http://www.jisan.org/research_collaborators/former_research/conjugate_schemata/papers/CONJUGATE98.PDF [accessed 2007-08-29].
- [20] Richard A. Caruana and J. David Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In John E. Laird, editor, Machine Learning, Proceedings of 5th International Conference on Machine Learning, pages 153–161, June 1988, Ann Arbor, Michigan, USA. Morgan Kaufmann, San Mateo, California. ISBN: 0-9346-1364-8.
- [21] James D. Watson, Tania A. Baker, Stephen P. Bell, Alexander Gann, Michael Levine, and Richard Losick. Molecular Biology of the Gene. Benjamin Cummings, FIFA (december 3, 2003) edition, 1987. ISBN: 978-0-80534-635-0.
- [22] J. Peter Gogarten and Elena Hilario. Inteins, introns, and homing endonucleases: recent revelations about the life cycle of parasitic genetic elements. *BMC Evolutionary Biology*, 6(94), 2006. doi:10.1186/1471-2148-6-94. Online available at <http://dx.doi.org/10.1186/1471-2148-6-94> and <http://www.biomedcentral.com/content/pdf/1471-2148-6-94.pdf> [accessed 2008-03-23].

11 Vyčíslitelnost a složitost

11.1 Úvod

Algoritmy, kterými jsme se doposud převážně zabývali, měly složitost polynomickou, tedy $O(n^k)$, kde k představuje konstantu, výjimečně exponenciální. Jak již víte z předchozích přednášek [1], existují dokonce i složitější algoritmy. Otázkou nyní je, zdali:

- jsou algoritmizovatelné veškeré problémy
- a zdali neexistuje způsob, jak v polynomiálním čase převést tzv. „těžký“ problém na tzv. „lehký“ problém, tedy z vysokých řad složitosti na polynomiální řadu složitosti



Obr. 144: Příklady NP-úplných problémů

11.2 Základní pojmy

P je jednou z nejzákladnějších tříd složitosti a představuje problémy, které je možné řešit pomocí **deterministického** Turingova stroje v polynomiálním čase.

NP je jednou z nejzákladnějších tříd složitosti a představuje problémy, které je možné řešit pomocí **nedeterministického** Turingova stroje v polynomiálním čase.

NP-úplné je skupina problémů, která je podmnožinou množiny NP a paří sem ty nejtěžší úlohy z NP.

Rozhodovací problém: je takový problém, který má libovolný vstup ale množina možných výstupů je dvouprvková – odpovídá vždy pouze ano či ne: {Ano, Ne}.

Výpočetní složitost: závislost trvání výpočtu na velikosti vstupu (počtu prvků, rozměru hracího pole atp.). Říká nám, jak s růstem velikosti vstupu **roste doba výpočtu**. Problemy, jejichž výpočetní složitost roste se vstupem exponenciálně či faktoriálně, se obecně pokládají za prakticky nezvládnutelné.

NP úplné problémy: Třída problémů, pro které není znám algoritmus řešící je v polynomiálně rostoucím čase. Pokud je ovšem nějaké řešení už známo, lze jeho platnost v polynomiálně rostoucím čase ověřit. Do této třídy problémů spadá například problém

obchodního cestujícího, splnitelnost či určení pravdivostní hodnoty logické formule a další. **Jednotlivé NP úplné problémy jsou na sebe vzájemně převoditelné.**

DNA počítací: Výpočetní stroj, využívající značnou dávku paralelismu. Při výpočtech zapisuje informaci do molekul nukleových kyselin. Tato látka však není pouze paměťovým médiem, ale příslušné chemické reakce mohou sloužit jako výpočetní činnost počítací. Při výpočtu se výpočetní postupy realizují prostřednictvím toho, jak se jednotlivé molekuly DNA "hodí/párují" k sobě. Je založen na obdobném principu jako současně známé výpočetní systémy (paralelní Turingův model⁷).

Heuristika: Snaha o předpovídání nejbližšího stavu, který by mohl vést k řešení.

11.3 Turingův stroj

Turingův stroj se v roce 1936 stal odpovědí na otázku, zdali existuje mechanický proces, kterým je možno rozhodnout o pravdivosti libovolného matematického teorému nebo výroku. Byl navržen Alanem Turingem za účelem **realizace kroků matematických důkazů posloupnosti symbolů a měl tak nahradit složitý a časově zdlouhavý matematický aparát**, který tehdy vědci museli používat. Tento stroj měl simulovat postup matematika při vytváření důkazu. Má několik základních vlastností:

- musel nahradit složitou symboliku matematických kroků. V takovém případě šlo každou konečnou množinu symbolů nahradit pouze dvěma symboly (jako je 0 a 1) a prázdnou mezerou, která by oba symboly oddělovala.
- podobně jako si matematik zapisuje poznámky na papír, **má Turingův stroj k zápisu nekonečnou pásku skládající se z buněk, do/ze kterých se symbol zapisuje/čte.**
- **nad touto páskou je možné provádět za pomocí čtecí hlavy operace čtení, zápisu a posunu pásky (read, write, shl, shr).**
- protože je možné symboly číst, zapisovat nebo se posunovat po páse, **je pro Turingův stroj důležitý vnitřní stav, ve kterém operaci čtení provádíme (čtený symbol a stav tak určují další akci a přechod do dalšího stavu).**

Protože se chování tohoto stroje vyvíjí podle tabulky přechodů, můžeme říci, že každý následující stav lze jednoznačně určit na základě čteného symbolu a aktuálního stavu. Jeho chování je proto deterministické.

Turingův stroj se stal teoretický model s nejvyšší dnes známou výpočetní silou. **Turingův stroj dokáže realizovat každou algoritmizovatelnou úlohu.**

⁷ Co je to Turingův model výpočetního stroje se dozvíte



Obr. 145: Schéma Turingova stroje

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_F) \quad (3)$$

- Q reprezentuje konečnou vnitřních stavů,
- Σ je konečná množina vnitřních stavů, nazývaná **vstupní abeceda**. Δ (prázdný symbol) není součástí Σ
- Γ je konečná množina vnitřních symbolů. Je to tzv. **pásková abeceda** a platí pro ni $\Gamma \subset \Sigma, \Delta \notin \Sigma$
- δ je parciální funkce a popisuje chování Turingova stroje, neboli, jinak řečeno, program. V tomto případě nelze měnit program za běhu počítače tak jak jsme zvyklí z dnešních výpočetních strojů. Je nazývána **přechodová funkce** a definována je jako

$$\delta: (Q \setminus \{q_F\}) \times \Gamma \rightarrow Q \times (\Gamma \cup \{L, R\}) \quad (4)$$

Kde $\{L, R\} \notin \Gamma$, a L je symbol pro přesun hlavy o jednu pozici doleva a R je symbol pro přesun hlavy o jednu pozici vpravo.

- q_0 je počáteční stav $q_0 \in Q$ a
- q_F je konečný stav $q_F \in Q$.

11.3.1 Univerzální Turingový stroj

Univerzální Turingův stroj (UTS) je takový TS, který jako vstup přijímá kód jiného TS a vstupní slovo stroje T. Dokáže rozkódovat přechodovou funkci stroje T a výpočet tohoto stroje simulovat. Univerzální TS dokáže vypočítat libovolnou částečně rekurzivní funkci (neboli je ekvivalentní k univerzální částečně rekurzivní funkci), rozhoduje jakýkoliv rekurzivní jazyk a přijímá libovolný rekurzivně spočetný jazyk.

Jinými slovy – UTS je stroj, který dokáže načíst program a pak podle něj provádět program.

11.3.2 Nedeterministický Turingův stroj

Nedeterministický TS (NTS) je takový TS, který má obdobné vlastnosti jako klasický TS s tím rozdílem, že přechodová funkce je

$$\delta: (Q \setminus \{q_F\}) \times \Gamma \rightarrow Q \times 2^{(\Gamma \cup \{L,R\})} \quad (5)$$

Pokud by někdo dokázal sestrojit NTS (popřípadě pokud již některá vláda světa něčím podobným disponuje), mělo by to za důsledek skutečnost, že bychom byli schopni řešit jakkoli složité problémy v polynomiálním čase. Mělo by to za následek konec veškeré dnes používané kryptografie, hra eternity II by se dala řešit v krátkém čase a jakkoli by rostl počet políček, nebyl by obtížný problém vyčkat déle/sestrojit výkonnější variantu a k výsledku bychom se dobrali.

11.3.3 Paralelní Turingův stroj

Paralelní turingův stroj, si lze představit jako několik TS, které pracují nad jedinou páskou (pamětí) současně, popřípadě nad jisté vymezené části pásky.

11.3.4 Kvantový Turingův stroj

Kvantový Turingův stroj model výpočetního stroje, který by v budoucnu mohl hrát významnou roli pro výpočet složitých problémů. Pro názornost se nyní omezme, že pásková abeceda může obsahovat pouze dva stavy $\Gamma = \{0, 1\}$. V případě klasického TS se tedy mohl na aktuální pozici TS vyskytovat stav buďto 0 či 1.

V případě kvantových TS se na aktuální pozici vyskytuje jak 0 tak 1 současně, respektive superpozice těchto dvou stavů. Stejně jako jeho klasický TS, může kvantový TS manipulovat s páskou, přesouvat a přeměňovat bity. Ale protože nyní to jsou kvantové bity (neboli q-bity), každé číslo, které máme otestovat, bude zakódováno současně na téže pásce v kvantové superpozici. Čísla mohou být otestována najednou, během jednoho jediného kroku stroje.

Blíže realitě se to dá představit jako řada atomů. Ty mají elektrony, které obíhají ve směru proti hodinovým ručičkám či ve směru hodinových ručiček (opačný spin atomů). Čtecí a zapisovací hlava je pak realizována pomocí laseru:

- Osvítí-li ho správnou frekvencí, dojde k rezonanci, kterou je možné detektovat stav (což bohužel změní stav q-bitu).
- Osvítíme-li na jistou dobu, změníme jeho spin
- Osvítíme-li jej na poloviční dobu, elektron zůstane současně obíhat v obou směrech (tj. je v superpozici obou stavů)

Díky tomuto jevu jsme schopni dosáhnout masivního paralelismu, jehož význam ve výpočetní technice bude velice podstatný. Pokud bychom byli schopni sestrojit kvantový Turingův stroj, byli bychom schopni čelit exponenciální explozi⁸. Problém, který expanduje 2^N bychom byli schopni realizovat pomocí superpozice 2^N stavů přidáním N atomů (q-bitů, tj. kvantových

⁸ Tedy alespoň v ideálním případě.

bitů). V současnosti jsme schopni řešit tento druh problémů pouze s 2^N TS. Naneštěstí, existují případy, kdy se nám výpočetní složitost podaří snížit pouze o odmocninu.

Nejdůležitější problém, který kvantové stroje v současnosti postihuje, je nemožnost číst hodnotu aniž bychom současně tuto hodnotu vymazali. Princip výpočtů bude postaven na principu, že se v určitém okamžiku podíváme, co spočítal. Přečtením, tedy vlastně měřením, dojde ke kolapsu systému, a kvantový počítač se musí reinicializovat a znova spustit výpočet. Z toho důvodu také pravděpodobně tyto stroje budou pracovat jako jakési servery, které nám budou poskytovat výpočetní výkon. Kvantové počítače již dnes jsou k dispozici. Jejich hlavním problémem je malý počet superpozic stavů (přibližně 5).

Zejména v neseriózní literatuře se můžete setkat s informací, že kvantovým TS je ekvivalentní NTS. To bohužel není pravda a s příchodem kvantových počítačů stále nebudeme schopni lámat šifry s využitím hrubé síly. Za poznámku stojí také tzv. DNA-počítače. Je to technologie, která také využívá masivního paralelismu ke zvýšení výkonnosti, nicméně tento druh strojů se považuje za klasický TS.

Pro zájemce o více informací ke kvantovým TS: [6], [7], [8], [9], [10].

11.4 Church-Turingova teze

Church-Turingova teze⁹, Churchova teze či Turingova teze označuje hypotézu o povaze a výpočetní síle mechanických strojů počítajících matematické funkce. Teze je pojmenována po Alonzu Churchovi a Alanu Turingovi, kteří s s tímto závěrem nezávisle přišli – i když každý jinou cestou. Alonz Church prostřednictvím zkoumání λ -calculu¹⁰[5] a Alan Turing z pohledu zkoumání TS.

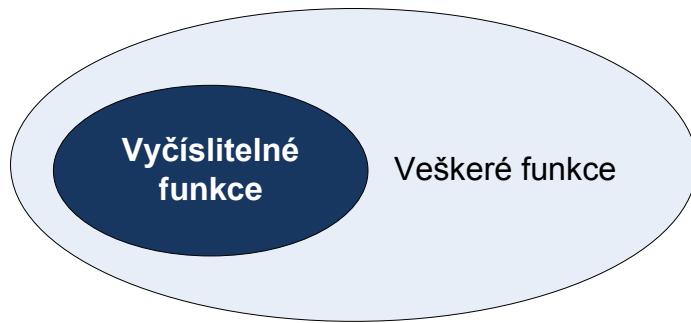
Church-Turingova teze říká, že každý algoritmus[1] může být vykonán Turingovým strojem.

Současně platí, že λ -calculus, konvenční programovací jazyky (včetně jazyků C, C++, JAVA, ...) jsou tzv. Turingovsky úplné, tedy že je pomocí nich možné popsat veškerou množinu algoritmů, které jsou vyčíslitelné.

Existuje-li tedy rozhodovací problém, pro jehož řešení neexistuje program například v jazyce JAVA, potom je tento problém algoritmicky neřešitelný a naopak.

⁹ Teze – něco co se předpokládá, ale neexistuje k tomu důkaz. Lze ji vyvrátit tím, že bude nalezena metoda, která bude akceptována jako algoritmus a nepůjde realizovat pomocí TS.

¹⁰ λ -kalkul je formální matematický systém, pro zkoumání funkcí. Ovlivnil vývoj funkcionálních programovacích jazyků jako je např. Lisp.



Obr. 146: Množina vyčíslitelných funkcí je podmnožinou množiny všech funkcí (současně tyto množiny nejsou ekvivalentní).

11.5 Nevyčíslitelné problémy

11.5.1 Rozhodovací problém

Zjednodušme si chápání počítače takto:

- **vstup programu** – jeho kódování můžeme interpretovat jako celé číslo (možná velmi veliké)
- **Řešení** – výstup programu, můžeme kódovat 1=ano, 0=ne

Rozhodovací problém je potom funkce $f: \mathbb{N} \rightarrow \{0, 1\}$, na základě vstupního čísla odpovídá ANO, či NE na konkrétní.

Příklad:

Problém: Je vstupní přirozené číslo liché?

Vstup: $n \in \mathbb{N}$	1	2	3	4	5	6	7	8	9
Výstup: $f(n)$	1	0	1	0	1	0	1	0	1

11.5.1.1 Algoritmy:

```
int jeLiche1(int n) {
    return n%2;
}
```

```
int jeLiche2(int n) {
    for( ; n > 1; i-=2):
        return n;
}
```

K řešení problému existuje více než jeden algoritmus! Obecně nekonečně mnoho – sice drtivá množina z nich velice neefektivních, nicméně existují.

Obecně takto lze zapsat libovolný algoritmus. Jakýkoli vstup si lze představit jako posloupnost bitů (tj. celého čísla) a každý algoritmus může odpovědět pravdu či nepravdu.

1.2. Nerozhodnutelný problém

Otázka nyní zní, existuje vůbec nějaký problém, který nejsme schopni algoritmizovat? K tomu abychom se dobrali k odpovědi, použijeme takzvanou diagonalizaci.

11.5.1.2 Důkaz pomocí diagonalizace

Předpokládejme množinu P všech rozhodnutelných problémů (algoritmů) zapsaných v jazyce JAVA. Do této množiny tedy musí patřit i předchozí funkce *jeLiche1()* a *jeLiche2()*. Těmto funkcím dejme na vstup čísla 1, 2, 3, … n , kde n náleží do množiny celých čísel N . Na základě toho může sestrojit následující tabulku:

<i>Algoritmy \ Vstup</i>	1	2	3	...	<i>k</i>	...
P_1	$P_1(1)$	$P_1(2)$	$P_1(3)$...	$P_1(k)$	
P_2	$P_2(1)$	$P_2(2)$	$P_3(3)$...	$P_1(k)$	
...
<i>jeLiche1()</i>	1	0	1		<i>jeLiche1(k)</i>	
...
<i>jeLiche2()</i>	1	0	1		<i>jeLiche2(k)</i>	
...
P_n	$P_n(1)$	$P_n(2)$	$P_n(3)$...	$P_n(k)$...
...

Dejme tomu, že bychom nyní chtěli program, který má na výstupu opačnou hodnotu daného algoritmu, tedy:

$$\begin{aligned} D(i) &= 0 && \text{jeli } P_i(i) == 1; \\ &= 1 && \text{jeli } P_i(i) == 0; \end{aligned}$$

To znamená, postupoval by diagonálně. Taková funkce se tedy oproti každému algoritmu liší nejméně v jednom řádku. Současně nezapomínejme, že jestliže množina P obsahuje všechny algoritmy, s D se liší minimálně v jednom řádku. Nastává tedy spor a také dokazujeme, že existuje algoritmus, který není rozhodnutelný.

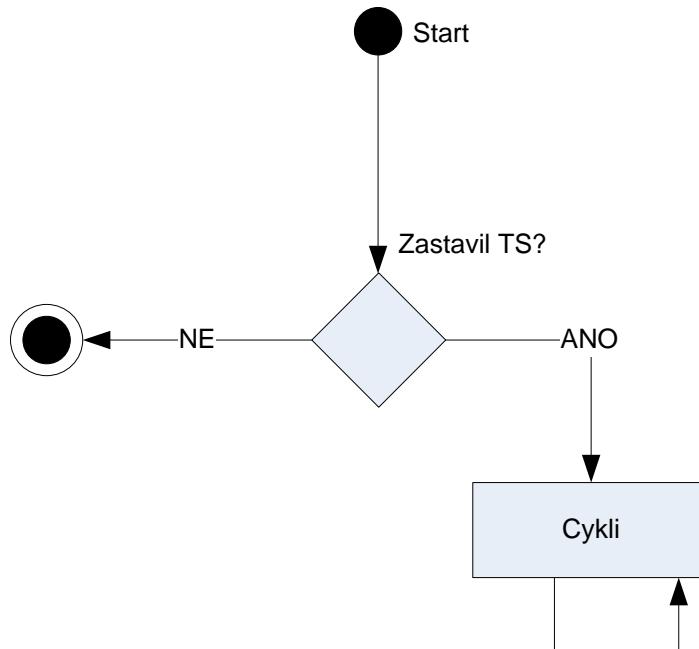
Pro rozhodovací problém D tedy neexistuje v Javě žádný program a tedy ani žádný algoritmus. Z toho plyne závěr:

Existují problémy algoritmicky neřešitelné (nerozhodnutelné).

11.5.2 **Problém zastavení TS**

Předchozí závěr má poměrně vážné důsledky. **Jedním je také to, že neexistuje algoritmus, který by byl schopen rozhodnout, zdali výpočet někdy skončí či zdali bude cyklist do nekonečna.**

Jediným způsob, jak by toto bylo možno odhadnout, zdali TS již cyklí je, pokud o algoritmu víme, že jeho časová složitost je nějaké a algoritmus již řeší problém déle.



11.6 Třídy složitosti P, NP

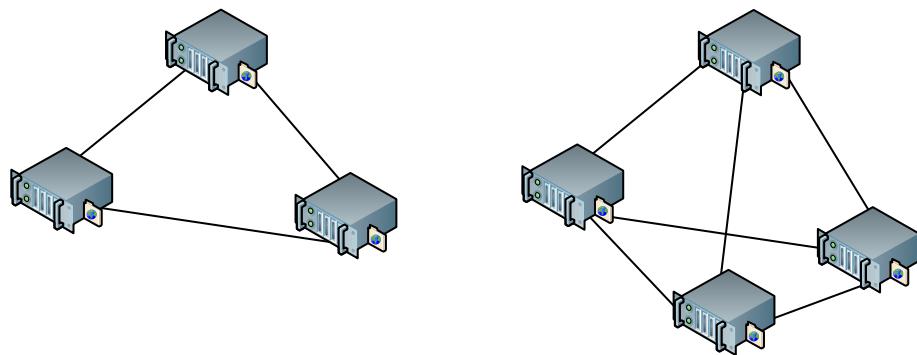
Otázka náležitosti P vs. NP je jeden z největších problémů současnosti v oblasti složitosti a matematiky vůbec. Teorie složitosti rozděluje problémy na třídy, které vyjadřují jak náročný výpočet je nezbytný, abychom problém vyřešili. Neboli jinak řečeno, jak „složité“ vlastně jsou. Vstupním argumentem funkce, která posuzuje tento počet, je n a její výstupem je počet operací. Jestliže počet operací pro výpočet takového problému lze vyjádřit jako polynomickou funkci:

$$a + bn^x + cn^y + \dots + dn^z \quad (6)$$

kde $a, b, c, \dots d$; x, y, z jsou konstanty a n je proměnná. Takový problém spadá do třídy složitosti P (deterministicky-polynomiální čas). Naopak, pokud lze funkci vyjádřit jako exponenciální funkci, či dokonce faktoriální funkci [1], znamená to, že problém spadá do třídy problémů NP (nedeterministicky-polynomiální čas). Někdy jsou zjednodušeně označovány tyto problémy jako lehké či těžké.

Příklad 1:

Mějme množinu síťových stanic, jejichž počet je N . Jak je časově složité, aby si každá dvojice stanic vzájemně vyměnily nějakou informaci? V takovém případě to znamená, že první bude komunikovat s $(N-1)$, druhá s $(N-2)$, atd. Z toho důvody se jedná o operaci, která má polynomiální časovou složitost.



Obr. 147: Ilustrace ke komunikaci serverů, kde N=3 a N=4.

Příklad 2:

Mějme problém obchodního cestujícího, který je typickým zástupcem NP třídy problémů. Obchodní cestující má za úkol navštívit veškerá města, a který musí navštívit každé z n měst právě jednou a vrátit se zpět domů (do počátku) a to navíc s nejkratší možnou výslednou cestou. Jak tento problém lze vyřešit „hrubou silou“ je tak, že vezme seznam vzdáleností mezi všemi městy a uděláme seznam všech možných variant, neboli řečeno matematicky – počet všech možných variant potom je variaci bez opakování z n prvků, což odpovídá faktoriálu: $n!$. Například pro Prahu, Brno, Olomouc a Ostravu ($N=4$) existují kombinace:

Praha, Brno, Ostrava, Olomouc
Praha, Brno, Olomouc, Ostrava
Praha, Ostrava, Olomouc, Brno
Praha, Ostrava, Olomouc, Brno
Praha, Olomouc, Brno, Ostrava
Praha, Olomouc, Ostrava, Brno
Olomouc, Brno, Praha, Ostrava
Olomouc, Brno, Ostrava, Praha
Olomouc, Ostrava, Brno, Praha
Olomouc, Ostrava, Praha, Brno
Olomouc, Praha, Ostrava, Brno
Olomouc, Praha, Ostrava, Brno
Brno, Olomouc, Praha, Ostrava
Brno, Olomouc, Praha, Ostrava
Brno, Ostrava, Olomouc, Praha
Brno, Ostrava, Praha, Olomouc
Brno, Praha, Ostrava, Olomouc
Brno, Praha, Olomouc, Ostrava
Ostrava, Brno, Olomouc, Praha
Ostrava, Brno, Praha, Olomouc
Ostrava, Olomouc, Praha, Brno
Ostrava, Olomouc, Brno, Praha

Ostrava, Praha, Brno, Olomouc
Ostrava, Praha, Olomouc, Brno

Tedy $4! = 4 \cdot 3 \cdot 2 = 24$ možností. Pro každou tuto variantu musíme navíc vyčíslit i vzdálenosti a musíme také provést srovnání výsledných vzdáleností, což výslednou složitost dále navýšuje. My se omezme na zjednodušení $n!$.

Počet prvků N	5	10	20
Komunikace serverů (P)	10	45	190
Obchodní cestující (NP)	120	3'628'800	2'432'902'008'176'640'000

11.6.1 NP-úplné problémy

Jistá podmnožina problémů z množiny NP jsou označovány jako NP-úplné (NP-complete). Do této skupiny spadá například i výše uvedený příklad obchodního cestujícího a spadá sem i mnoho dalších problémů postavené na podobném principu. Existují stovky takových příkladů od nejlepšího tahu v šachu, dešifrování RSA šifry (používána v elektronickém bankovnictví ale dnes i vojenství), Hamiltonský okruh a samozřejmě již zmíněný problém obchodního cestujícího.

11.6.2 Problém ekvivalence P vs. NP

Jelikož všechny tyto problémy jsou matematicky ekvivalentní a jelikož se jedná o nejsložitější podmnožinu problémů z množiny NP, jestliže někdo nalezne řešení, jak některý takový problém řešit v polynomiálním čase, znamenalo by to, že stejně tak lze řešit i všechny ostatní problémy. Je zřejmé, že to by mělo velké důsledky pro většinu kryptografie, která je dnes používána a považována za bezpečnou. Druhou možností je, že bude rozhodnuto, že takový algoritmus neexistuje a že tedy nemůžeme řešit NP-úplný problém v polynomiálním čase.

Clayův matematický ústav zařadil tuto otázku vztahu P vs. NP mezi 7 největších matematických problémů současnosti - každá z těchto úloh je přitom oceněna milionem dolarů. „Tržní“ cena problému P/NP je ovšem nejspíš mnohem vyšší.

P schůdné algoritmy běžící nejhůře v polynomiálním čase (dále jen PT), příklad: násobení **NP** neschůdné algoritmy; pouze správnost řešení lze ověřit v PT, příklad: faktorizace **NP-úplný** Množina problémů z množiny NP, které jsou považovány za nejsložitější.

11.7 Literatura

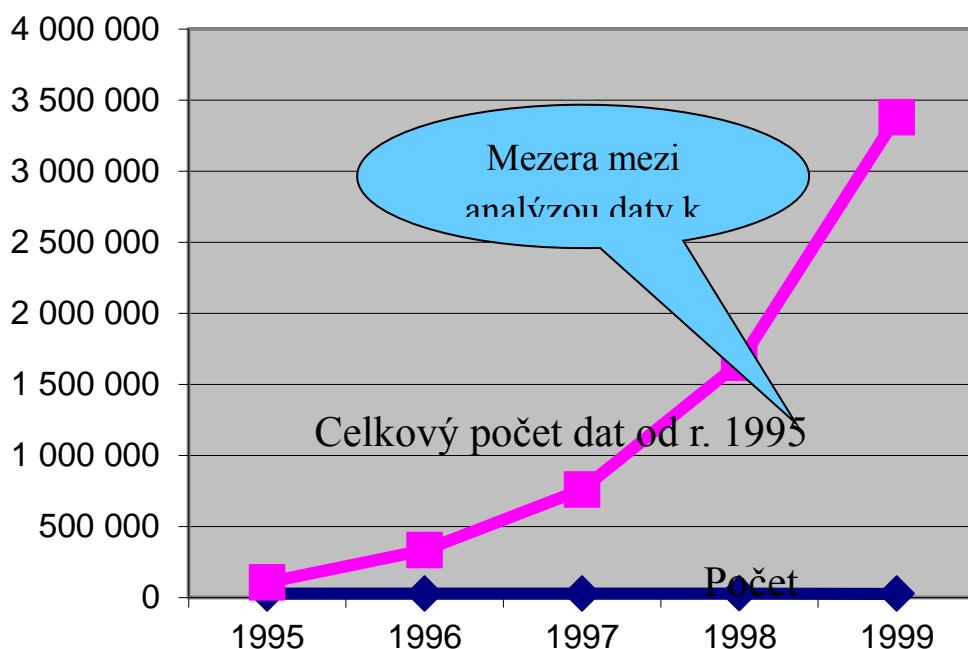
- [1] BURGET, R. Hodnocení algoritmů, Přednáška MTIN.
- [2] Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)

- [3] Clay Mathematics Institute (CMI), Cabridge, Massachusetts
<http://www.claymath.org/>
- [4] Millennium Prize Problems, Cabridge, Massachusetts
<http://www.claymath.org/millennium/>
- [5] Lambda Calculus
<http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/>
- [6] Cesta ke kvantovému počítači (1) - faktorizace a kvantový Turingův stroj
<http://www.scienceworld.cz/sw.nsf/ID/87C5E59169CF6DF9C1256F2D004F9E1B>
- [7] Cesta ke kvantovému počítači (2) - kvantový celulární automat
<http://www.scienceworld.cz/sw.nsf/ID/BEB82A3388A8DD5CC1256F2D0050F51E>
- [8] Andrew Yao (1993). "Quantum circuit complexity". Proc. of the 34th Ann. Symp. on Foundations of Computer Science: 352-361.
- [9] Lance Fortnow (2003). "One Complexity Theorist's View of Quantum Computing". Theoretical Computer Science 292: 597--610.
- [10] Quantum Computation <www.cs.rpi.edu/~bushs/csci_4967/notes7.pdf>

12 Analýza dat

12.1 Úvod

Souvislost s vědním oborem teoretická informatika je zřejmý – snažíme se dostat na (za) hranici toho, co v současnosti považujeme za spočitatelné a s pomocí současných hardwarových prostředků vyčíslitelné. Znalosti, ke kterým je možné se dostat, je snažou nejen nalézt, ale také zobecnit a transformovat do nové formy znalostí. Z takto získaných znalostí je v budoucnu možné vyvodit další poznatky o zkoumané problematice a stanovit nová pravidla. V dnešní době již není, díky moderním technologiím, problémem přístup k datům. **Nárůst objemu dat** je v posledních letech exponenciální a nic nenasvědčuje tomu, že by se v tomto ohledu mělo něco změnit. Problémem v současnosti je, jak se dostat ke kvalitním datům resp. takovým datům, které nás právě zajímají. Analýza provedená **člověkem může trvat týdny**. Informace bývá v datech často skrytá a není patrná na první pohled. Významná část dat tak nebývá často analyzována vůbec.



Obrázek 1: Zdroj: R. Grossman, C. Kamath, V. Kumar, "Data Mining for Scientific and Engineering Applications".

Jedna z nejpoužívanějších architektur, pomocí které je možné skladovat data je nazývána **datové sklady** (spíše zažito pod anglickým „**data warehouse**“, terminologie je v rámci českého jazyka nejednotná a často se tak označují i věci s tímto nesouvisející). Jedná se o **mnoho různorodých dat, jednotně organizovaných na jediném místě zpravidla za účelem analýzy a vytváření rozhodnutí**. Tyto datové sklady mají za úkol pročistění dat, vzájemnou integraci dat s jinými daty a on-line analytické zpracování (**OLAP**), kde se používá

multidimensionální kubická reprezentace dat. Tato operace přináší funkce, jako jsou: summarizace, konsolidace (slučování), agregaci (shromažďování) a možnost nahlížet na data v různých úhlech pohledu [5], [6], [7]. Za zmínu stojí operátory roll-up a drill down, které umožňují zobecnění či naopak rozšíření detailů při pohledu na data.

Uplatnění dolování znalostí z báze dat je široké a pokrývá téměř všechny obory. Zahrnuje oblast rozhodování (veškeré finanční instituce při rozhodování o poskytnutí úvěru / hypotéky, rozhodování o investici, marketing, prodej, ...), diagnóza (expertní systémy, předpověď selhání stroje v kritických částech výrobního řetězce, analýza medicínských dat, rozpoznávání a identifikace osob v kamerových dohledových systémech, strojová identifikace nevyhovujících součástek na výrobní lince, ...), předpověď dalšího vývoje (investice do finančních trhů, počasí, elektrická rozvodná síť), klasifikace (určení mluvčího z hlasu, určení emoce v hlase, identifikace mluvčího), identifikace mimořádných událostí (identifikace útoku v síti, ...).

Výhody takových systémů jsou zřejmě – výrazně nižší provozní náklady, mnohonásobně rychlejší a mnohdy přesnější v porovnání s člověkem. Na druhou stranu, stroje vychází jen z toho, na co byly natrénovány. Pokud se podmínky mění často a je potřeba pokaždé čerpat z různých zdrojů, je zpravidla nasazení strojového učení obtížnější a i finančně nákladnější. V takových případech je třeba strojové učení kombinovat se zásahem experta v oblasti dolování znalostí / zpracování signálů.

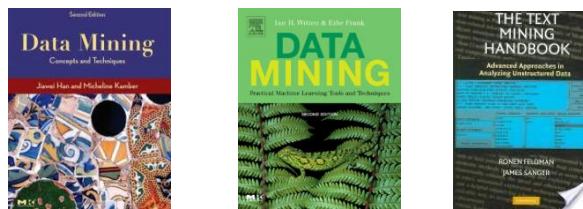
Je třeba také říci, že strojové učení není všemocné. **Nevýhod** strojového učení je také mnoho. Obecně lze říci to, že naučený model může být jen do té míry kvalitní, jak kvalitní trénovací data jsme schopni poskytnout. Je-li kvalita těchto dat nízká, nemůžeme očekávat, že naučený model bude svojí kvalitou převyšovat kvalitu obdržených dat. S tím také souvisí další problém - pro natrénování kvalitního modelu je nezbytné, datům dokonale porozumět. Jelikož je trénovací množina vždy konečná a mnohdy velice omezená, co do počtu vzorků (cena získávání takových znalostí může být drahá, někdy i obtížná či téměř nemožná), je možné natrénovat např. neuronovou síť i na náhodná data a dosáhnout relativně vysoké míry přesnosti. Je asi zřejmé, že při aplikaci natrénovaného modelu na jiná data (nezávislá na trénovací množině, jeho hodnota nebude žádná). **Typickým příkladem** je např. technická analýza finančních trhů (tj. vysoká míra šumu, málo relevantních trénovacích dat k dispozici) – i přesto, že téměř určitě není trh zcela efektivní (jako každá činnost člověka), relevantních dat pro trénování je velice málo. Pokud nejste experty na finanční trhy a přesně nevíte, co chcete rozpoznat i tak vysoko pravděpodobně dosáhnete natrénování takového modelu, který bude vykazovat relativně dobrou spolehlivost. Přesto všechno je při reálném obchodování vysoká pravděpodobnost, že ve skutečnosti má předpověď přesnost 50 : 50.

Proces dolování znalostí potom zahrnuje následující kroky:

- **Čištění dat** (odstranění nekvalitních dat a nekonzistentních dat)
- **Integrace dat** (např. kombinace několika datových zdrojů, kdy může být i různá sémantika dat např. km vs. míle, °C vs. °F)
- **Výběr dat** (z databáze jsou získána data relevantní k záležitosti, kterou chceme analyzovat)

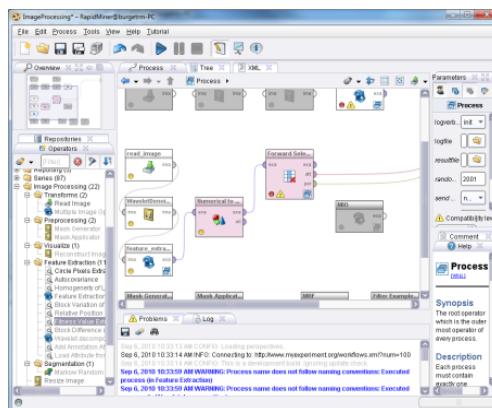
- **Transformace dat** (neboli preprocessing, konsolidace či transformace dat do podoby vhodné pro dolování znalostí)
- **Dolování znalostí** (proces aplikace učících se algoritmů za účelem extrakce vzorů a pravidel)
- **Ohodnocení naučeného modelu** (posouzení kvality na základě objektivního kritéria)
- **Reprezentace znalosti** (naučená znalost je uživateli vizualizovaná či reprezentována v jiné podobě)

Problematika dolování znalostí by svým rozsahem klidně pokryla několik předmětů. Pro získání hlubších znalostí z oblasti dolování znalostí doporučuji 1) experimentovat s daty a 2) literaturu [1], [2] a [8]:



Obrázek 2: Doporučená literatura

Jako nástroj pro dolování znalostí doporučuji zejména open-source nástroj RapidMiner¹¹, který je v současnosti jedním z nejčastěji používaných analytických nástrojů a je zdarma ke stažení. V nabídce „HELP>Rapid Miner tutorial“ naleznete řadu praktických příkladů, které vám představí alespoň některé z možností tohoto nástroje. Dalšími nástroji může být Weka, [MLC++](#) a další.



Obrázek 3: Nástroj RapidMiner.

12.2 Vstup: Formátování vstupních dat

Vstupní data mohou nabývat následujících formátů:

- Záznamy
 - Matice dat

Projection of x Load	Projection of y load	Distance	Load	Thickness
10.23	5.27	15.22	2.7	1.2
12.65	6.25	16.22	2.2	1.1

¹¹ RapidMiner - <http://rapid-i.com>

- Dokumenty (frekvence významných slov v dokumentu)

	team	coach	y	pla	ball	score	game	n	wi	lost	timeout	season
Document 1	3	0	5	0	2	6	0	2	0	0	2	
Document 2	0	7	0	2	1	0	0	0	3	0	0	
Document 3	0	1	0	0	1	2	2	0	3	0	0	

- Transakční data (jsou speciálním případem matic dat, zahrnují množinu položek)

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

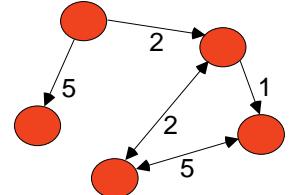
- Grafy

- World Wide Web

```

<a href="papers/papers.html#bbbb">
Data Mining </a>
<li>
<a href="papers/papers.html#aaaa">
Graph Partitioning </a>
<li>
<a href="papers/papers.html#aaaa">
Parallel Solution of Sparse Linear System of Equations </a>
<li>
<a href="papers/papers.html#ffff">
N-Body Computation and Dense Linear System Solvers

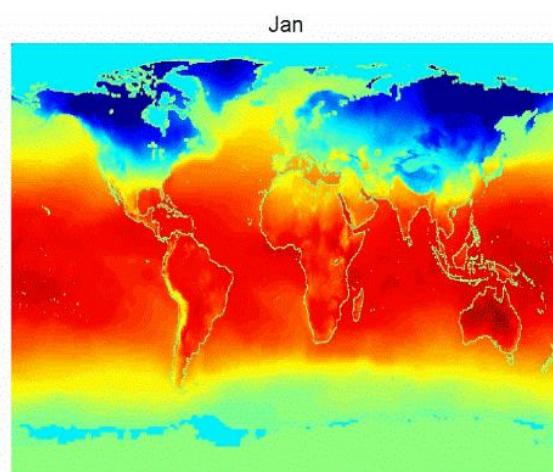
```



- Schémata zapojení
- Molekulární struktury

- Řazená data

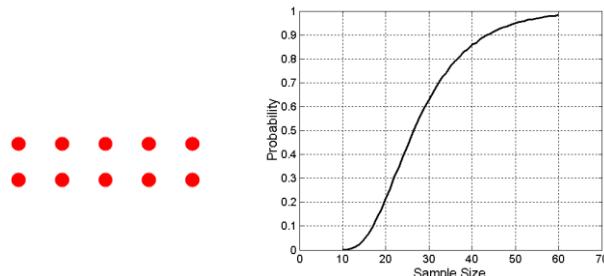
- Prostorová data



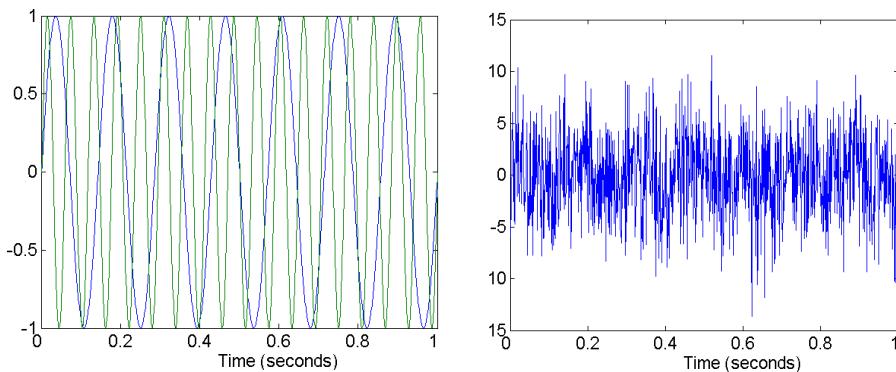
- Dočasná data
- Sekvenční data

Problémy, se kterými se budeme setkávat, jsou:

- Velký počet dat - podvzorkování

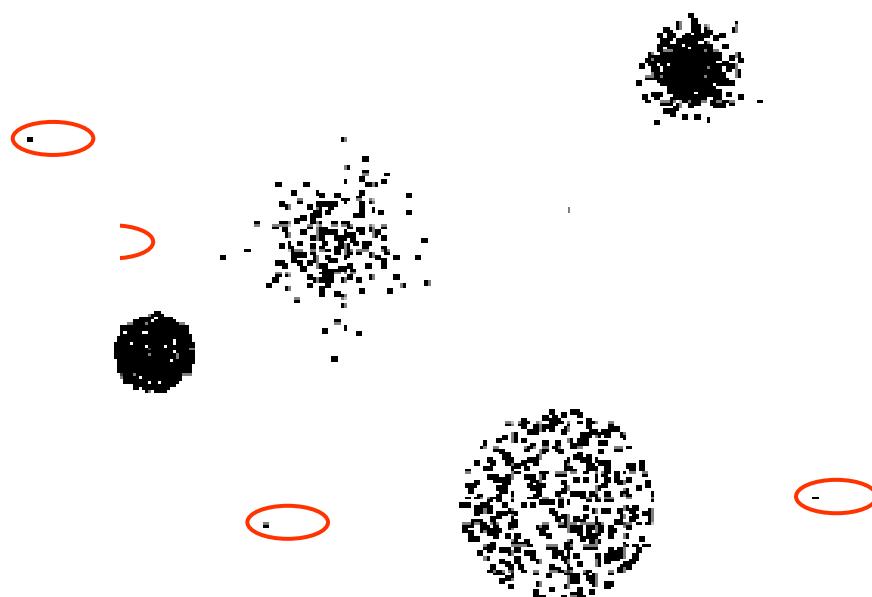


- Počet rozměrů (dimensionalita – s jejím růstem narůstají paměťové nároky, hůře se hledají shluky a často pak nemají ani smysl => vizualizovat data, odstranit nepotřebné)
- Řídká data (zaznamenána je pouze přítomnost v dané kategorii)
- Rozlišení (vzory jsou závislé na rozlišení a počtu detailů)
- Kvalita trénovacích dat
 - Poměr šumu



Obrázek 4: Sinusoida bez šumu a s přidaným šumem.

- Body ležící mimo trénovací převážnou část (Outliers). Tendence trénovacích algoritmů je zprůměrování hodnot.



Typický příklad vstupních dat může vypadat následovně:

ID	PSČ	SLUŽBA	POPLATEK	DATUM PODPISU
1	612 00	CCA	350	2010-03-01
2	637 00	CCA	350	2003-01-01
3	796 01	AA3	699	2006-01-01
...				

Data jsou reprezentovány jako kolekce **datových záznamů** (neboli také: instancí, příkladů, vzorků, entit, hodnot atributů, či hodnot příznaků) a **atributů** (či také příznaků, příkladem jsou položky: ID, PSČ, SLUŽBA) těchto záznamů. Atribut udává typ těchto atributů a přiděluje jim i sémantický význam – výška domu v metrech, milimetrech, stopách, relativně vůči referenční budově atp. Různé atributy mohou být mapovány ke stejné hodnotě: rodné číslo / ID. Vlastnosti těchto atributů ale mohou být odlišné – zatímco např. ID může být i jedna či záporné číslo, rodné číslo může nabývat jen jistého definovaného oboru hodnot.

12.2.1 AML formát dat

Obecně lze říci, že datových formátů pro reprezentaci trénovacích dat je nekonečně mnoho. Jedním společným a snad i **nejčastěji používaným formátem je formát CSV** (Comma Separand Values, neboli čárkou oddělené hodnoty). Jedná se o velice jednoduchý formát, snadno čitelný pro člověka a snadno převoditelný do libovolného jiného formátu. Díky této jednoduchosti je často podporován napříč celou škálou aplikací a je možné s jeho pomocí sdílet vstupy i výstupy aplikací.

Závažnou nevýhodou formátu CSV je skutečnost, že formát **nedefinuje parametry atributů a uvádí pouze jejich hodnoty**. Takto by např. mohla být PSČ chybně interpretována jako typ

numerický či ordinální, což by velice pravděpodobně vedlo ke zmatení klasifikátoru a natrénování učícího se algoritmu na nesmyslné hodnoty.

My se zejména v rámci cvičení budeme zabývat formátem **AML**. Data uložená v tomto formátu obsahují dva soubory: *.AML a *.DAT. V rámci DAT souboru jsou jednotlivé hodnoty odděleny mezerou. V rámci souboru AML jsou objasněny významy jednotlivých atributů (sloupců) těchto dat.

Příklad DAT souboru:

1.476699 1.384402 1.338097

1.776699 1.3454871 3.545454

Příklad AML souboru:

```
<attribute name="Feature1" sourcecol="1" valuetype="real"/>
<attribute name="Feature2" sourcecol="2" valuetype="real"/>
<attribute name="Feature3" sourcecol="3" valuetype="real"/>
```

12.2.2 Řídká data

0,	26,	0,	0,	0,	55,	10,	„Třída A“
3,	0,	13,	0,	0,	0,	0,	„Třída B“

Lze daleko efektivněji reprezentovat pomocí řídké reprezentace:

{ 1 26, 5 55, 6 10, „Třída A“}

{ 0 3, 2 13, „Třída B“}

Jak tento tak předchozí případ reprezentuje zcela stejnou informaci. Řídký formát ovšem udává dvojice (pozice, hodnota). Pro hodnoty 0 není záznam uveden vůbec.

Reprezentovaná znalost je ekvivalentní, ale paměťové nároky jsou výrazně náročnější. Při řešení reálných problémů je počet atributů v řádeku stovek či dokonce tisíců.

12.2.3 Typy atributů

Každý učící se algoritmus má jiné chování a s tím souvisí, že má i jiné požadavky na vstupní data. Zdaleka ne každý algoritmus si dokáže poradit se vstupem v podobě nominálních či reálných dat, což závisí silně na charakteru a konkrétním případu.

- Nominální
 - Binominální
 - Polynomínální
- Ordinální
 - Jdou postupně za sebou
 - Data
 - Čas
- Numerické
 - Reálné
 - Celočíselné
 - Interval

- Procenta
- Speciální
 - Obraz
 - Zvuk
 - Časová řada
 - Video
 - Grafy
 - PSČ
 - A mnoho dalších...

To o jaký druh atributu se jedná, pak vede k tomu, jaké operace je možné s nimi provádět. Možné operace jsou:

- Rovnost: $= \neq$
- Pořadí: $< >$
- Sčítání: $+ -$
- Násobení: $* /$

To jaké operátory je možné využít, ovlivňuje, jaké učící se algoritmy je možné nasadit. Jednotlivým skupinám pak naleží následující operace:

- Nominální: rovnost
- Ordinální: rovnost a pořadí
- Interval: rovnost, pořadí & součet
- Procenta: všechny 4 vlastnosti

Zejména u ostatních atributů (grafy, multimediální data, obrazy, zvuk) je nutné z dat extrahat příznaky zpravidla v podobě numerických hodnot, na které je pak možné aplikovat klasické algoritmy učení.

Typ atributu	Popis	Příklad	Operace
Nominální	Hodnoty jsou zkrátka PSČ, zaměstnanecké jen různé názvy a je ID, barva tedy možné jen odlišit, zdali se hodnoty rovnají či nikoli. ($=, \neq$)	PSČ, zaměstnanecké ID, barva očí, kvadrát test	Entropie, korelace, chí
Ordinální	Data je možné vzájemně porovnávat ($<, >$)	Tvrnost materiálu (nízká, střední, vysoká)	Medián, percentil, korelace
Interval	V případě intervalů jsou smysluplné rozdíly mezi hodnotami	Data v kalendáři	Střední hodnota, odchylka, Pearsonova korelace, t a F test
Procento	V takovém případě jsou smysluplné i poměry k jiným atributům	Teplota v Kelvinech, věk, délka, elektrické napětí	Geometrická a harmonická střední hodnota, procentuální odchylka

12.2.4 Chybějící hodnoty

V praxi se s chybějícími daty setkáváme velice často, daleko častěji nežli tomu je v prostředí výukových příkladů a je nezbytné umět si s těmito případy poradit.

V celku často se setkáte s případy, kdy hodnoty nejsou známy a jsou přiřazeny nesmyslné hodnoty, které mohou významně poškodit přesnost algoritmu. Příkladem může být délka o hodnotě -1. Současně je zapotřebí mít na paměti, že převážná většina algoritmů předpokládá, že chybějící hodnota je nezajímavá. Přesto při podrobnějším prozkoumání dat z toho mohou a často i vyplynout zajímavé skutečnosti, které významně ovlivní přesnost výsledného modelu.

Jak se vypořádáváme s chybějícími záznamy:

- Odstranit záznamy s chybějícími položkami
- Odhad chybějících hodnot
- Použití algoritmů, které se dokážou vypořádat s chybějícími položkami
- Náhodné hodnoty s využitím distribučního rozložení zbytku dat

12.2.5 Nepřesné hodnoty

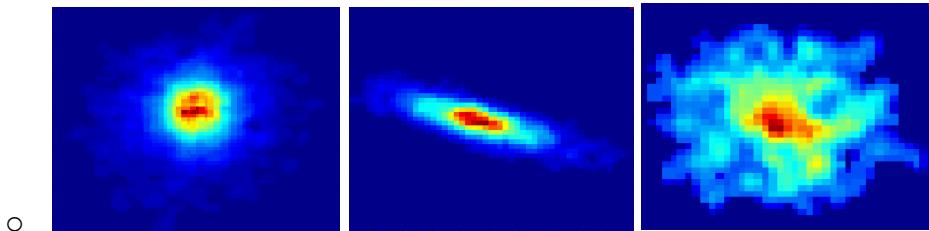
Vždy platí, že trénovaný model může být jen do té míry přesný, jak přesná jsou vstupní data. Je tedy snahou získat co nejpřesnější data a pokud možno co největší množství hodnot.

Při konsolidaci dat (slučování z několika různorodých zdrojů), také často narazíme na problém, že se položky mohou opakovat. Jejich odstranění nemusí být vždy triviální. Je zapotřebí trénovací data znát a rozumět jím

12.3 Výstup: Reprezentace znalostí

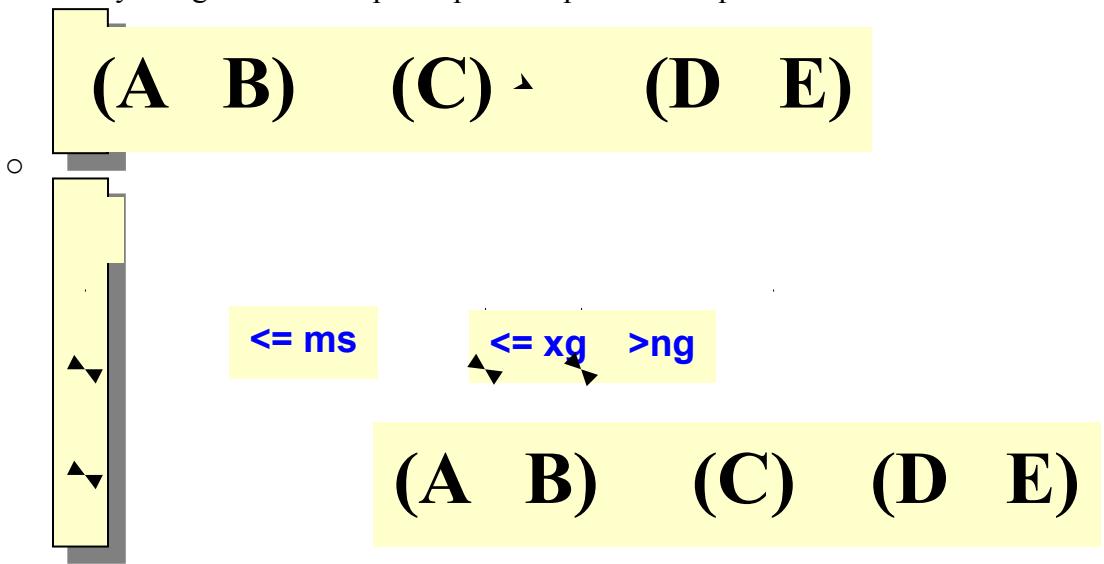
Výstupem dolování znalostí je vždy nějaký druh znalosti. Úlohy řešené v rámci dolování znalostí lze rozdělit na tyto základní třídy:

- **Klasifikace [Předpověď]**
 - Identifikace nemoci z rentgenu / CT / MRI a klasifikace závažnosti
 - Je daný zákazník návyklý, aby odešel ke konkurenci? (Nabídnout mu akční cenu / slevový kupón?)
 - Oslovování jen vybrané podmnožiny zákazníků, kde je nejvyšší pravděpodobnost, že o službu bude mít zájem (IP telefonie, IPTV, vysokorychlostní internet, ...)
 - 72 milionů hvězd, 20 milionů galaxií
 - skorá
 - střední
 - pozdní fáze galaxie



- 1) segmentace obrazu, 2) měření vlastností (cca 40 pro každý objekt), příprava trénovací množiny – výsledkem bylo nalezení 16 nových kvazarů ve vesmíru

- **Shlukování (Clustering) [Popis]**
 - Kategorizace podobných dokumentů
 - Kategorizace nahrávek call centra
- **Nalezení asociativních pravidel [Popis]**
 - V internetovém obchodu nalézt takový druh zboží, který bývá často nakupován společně a nabídnout akční balíček => zvýšení prodeje
- **Zkoumání sekvenčních pravidel [Popis]**
 - Analýza logů na serveru pro napadení / pokusu o napadení



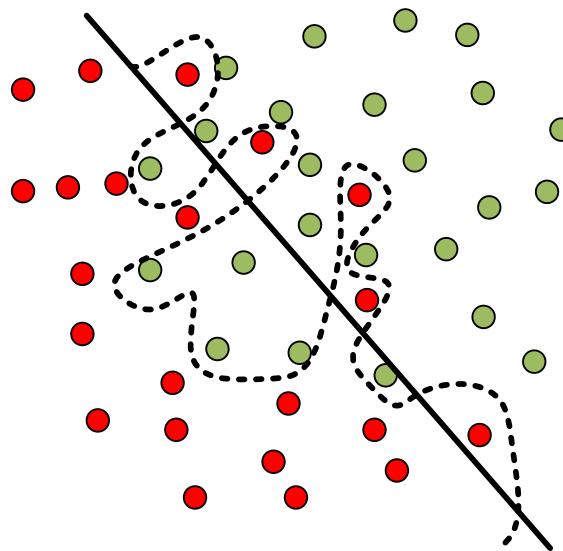
- **Regresi [Předpověď]**
 - Předpověď zájem (počet prodejů) o konkrétní službu (internet, IPTV, mobilní internet, telefonní služby, ...)
 - Předpověď budoucí nároky na danou službu (ČEZ, ...)
- **Detekce odchylek [Předpověď]**
 - Detekce zneužití kreditní karty
 - Detekce útoku v síti (Network Intrusion Detection)

12.4 Věrohodnost: Zhodnocení, co bylo naučeno

Jedna z nejčastějších chyb začátečníků je nasazení trénovacího algoritmu na problém, následná optimalizace parametrů tak, aby výsledná chyba byla co nejnižší a na základě získaných parametrů vyvození závěru, že jsme dosáhli té či oné přesnosti. Takto to bohužel

nefunguje a na ověření kvality natrénovaného modelu je nezbytné použít **data nezávislá na učení**.

Demonstrujme si to na následujícím příkladu (nutné podotknout, že výsledky jsou samozřejmě ovlivněny charakterem dat, obecně lze ale říci, že to v 95 % případů platí tak, jak je níže popisováno).



Obrázek 5: Klasifikace dat s pomocí přímky a křivky.

Na Obrázek 5 jsou znázorněny dvě třídy, které se snažíme klasifikovat. Jak je také z obrázku patrné, dvě třídy nelze zcela triviálně oddělit a obsahují jistou míru šumu, díky kterému se tyto dvě množiny do jisté míry prolínají. Pokud použijeme trénovací algoritmus, lze téměř vždy dosáhnout poměrně přesného rozdělení dvou množin. Jelikož nikdy nejsme schopni poskytnout nekonečnou trénovací množinu (velikost ovlivňuje čas i paměťové nároky) lze se takto ve skutečnosti naučit i na zcela náhodná data. Takový stav je naznačen na obrázku přerušovanou oddělovací čarou a říkáme mu **přetrénování** a takový model zpravidla dává špatné výsledky. Charakter dat je ve skutečnosti oddělitelný lineárně a oddelením čarou jsme schopni na nezávislých datech dosáhnout výrazně lepších výsledků.

12.5 Trénování a testování

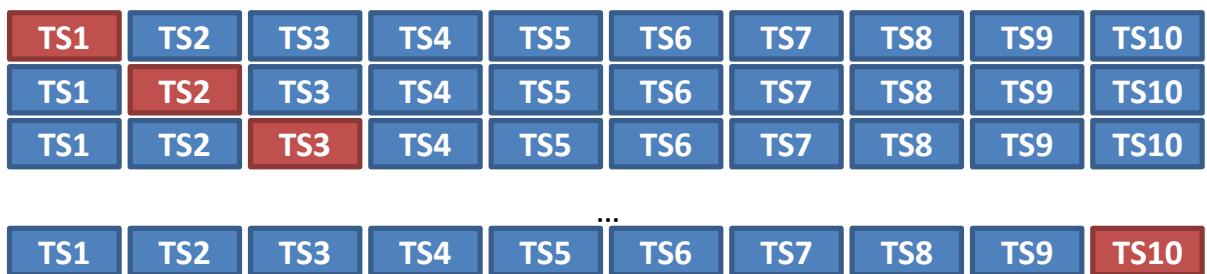
Za účelem dosažení co nejlepších výsledků by data měla být rozdělena na **data trénovací** a **data testovací**. Obecný postup je, že se následně nastavují parametry učícího se algoritmu tak, aby se při natrénování s pomocí trénovacích dat a ověření na testovacích datech, dosáhlo co nejlepších výsledků. Tím se ovšem nezávislost dat částečně vytrácí a pro objektivní posouzení přesnosti by měla existovat **data validační**.

Pokud je množství dat dostatek, tak s přípravou těchto tří množin problém není. Pro všechny ostatní případy (nastává ve většině případů) je nutné použít některou z více sofistikovaných metod posouzení natrénovaného modelu, jako je například cross-validace.

Je v každém případě nutné dodržet, aby data validační byla zcela nezávislá

12.5.1 Cross-validace a její varianty

Cross-validace je metoda, s pomocí které lze efektivně ohodnotit kvalitu naučeného algoritmu (a správnosti nastavených parametrů) na omezeném vzorku dat. Jak pracuje, je naznačeno na obrázku níže. V cross-validaci jsou data určena pro trénování současně daty testovacími. Data jsou nejprve rozdělena do n skupin. Nejčastějším počtem těchto skupin je 10, jak je také vyobrazeno na obrázku. Následně je 9 skupin použito pro učení a jedna zbývající pro otestování. To je možné opakovat n -krát tak, že se vystřídají všechny skupiny pro testování právě jedenkrát.



Obr. 148: Princip křížové validace.

Speciálním případem cross-validace je, když počet skupin n je roven počtu vzorků v množině. Taková metoda je označována „jeden vynechej“, neboli **leave-one-out**. Jedná se tak o nejpřesnější možné zhodnocení. Na druhou stranu je pro množinu čítající 1000 vzorků nutné 1000x natrénovat a 1000x testovat, což sebou přináší značné časové nároky (uvažme, že zpravidla chceme otestovat přesnost s různými nastaveními učícího se algoritmu).

Rozdělení metod pro vytváření těchto množin je následující:

- Lineární vzorkování (data jsou rozdělena do skupin, jak jdou za sebou)
- Promíchané (shuffled) vzorkování (data jsou promíchaná, jsou zahrnuty všechny vzorky a nikdy se žádný neopakuje)
- Vrstvené (stratified) vzorkování (data jsou promíchaná, některé hodnoty se mohou opakovat a některé se dokonce pro trénování mohou vynechat – cílem je dosáhnout vyšší odolnosti na proměnlivost dat)

12.5.2 Bootstrap

Bootstrap je metoda, která se snaží vyrovnat s problémem omezeného množství trénovacích dat a s její pomocí se odhadují parametry učícího se algoritmu s ohledem na standardní chybu a intervaly důvěryhodnosti. Je založena na výběru vzorků s opakováním a ostatní vzorky, které nebyly vybrány, tvoří testovací množinu. Tento postup je opakován po definovaný počet iterací a následně jsou získané výsledky zprůměrovány do výsledné hodnoty. S pomocí této metody lze vytvořit z jedné množiny dat mnoho mírně odlišných. Její výhodou je relativně značná jednoduchost.

Nevýhodou této metody je, že poskytuje poměrně optimistické výsledky.

Metodu je vhodné používat v těchto případech:

- **Když je teoretické statistické rozložení složité či neznámé.** Metoda bootstrap je nezávislá na distribučním rozložení a nabízí tak nepřímou metodu pro určení parametrů učícího se algoritmu, který by měl být schopen se vypořádat i komplexní distribucí dat.
- **Když je velikost trénovací množiny malá.** Bootstrap pomůže vyrovnat se s případy, kdy je vinou omezeného počtu vzorků není distribuční rozložení dat plně čitelné (distribuční rozložení dat může být známa).

12.6 Srovnání kvality učících algoritmů

V ideálním případě by mělo být možné na základě srovnávacích vztahů vzájemně srovnávat jednotlivé algoritmy. Bohužel to ve skutečnosti není tak snadné a často je nezbytné z pohledu člověka a z pohledu konkrétní řešené úlohy přesně určit, jakým způsobem se mají řešení srovnávat.

Jedním z nejčastěji používaných nástrojů pro analýzu je matice záměn (confusion matrix), ze které jsou pak vyvozeny některé další vztahy pro posuzování přesnosti klasifikace:

		Předpověděná třída	
		Class=Ano	Class=Ne
Skutečná třída	Class=Ano	A (TP)	B (FN)
	Class=Ne	C (FP)	D (TN)

- **A: TP (true positive)**
- **B: FN (false negative)**
- **C: FP (false positive)**
- **D: TN (true negative)**

Z matice záměn je potom možné vyjádřit mnoho parametrů. Jedna z nejčastěji používaných je **přesnost (y):**

$$\text{Accuracy} = \frac{a + d}{a + b + c + d} = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

Přesnost (Accuracy)

Omezení přesnosti lze demonstrovat na následujícím příkladu. Mějme problém klasifikace ve dvou třídách. V případě třídy A existuje 9990 vzorků. V případě třídy B existuje 10 vzorků. Jestliže tedy natrénovaný model zahrne vše do třídy A, je jeho přesnost 99,9 %. Získaná informace ale nemá téměř žádnou hodnotu.

Dalšími parametry jsou:

Specifičnost je vyjádřením proporce negativních ohodnocení vůči všem negativním:

$$\text{Specifičnost} = \frac{TN}{FP + TN} \quad (2)$$

Senzitivita (R , recall) je vyjádřením správně klasifikovaných vzorků vůči všem pozitivně klasifikovaným:

$$R = \frac{TP}{TP + FN} \quad (3)$$

Youden (Y), tento parametr se nedoporučuje používat pro obecné případy

$$Y = R - \text{Specifičnost} - 1 \quad (4)$$

Pozitivní předpovídající hodnota, poměr správně předpovězených vůči všem pozitivně předpovězeným:

$$PPV = \frac{N_{TP}}{N_{TP} + N_{FP}} \quad (5)$$

Negativní předpovídající hodnota:

$$NPV = \frac{N_{TN}}{N_{TN} + N_{FN}} \quad (6)$$

F – míra (F-measure, F-score):

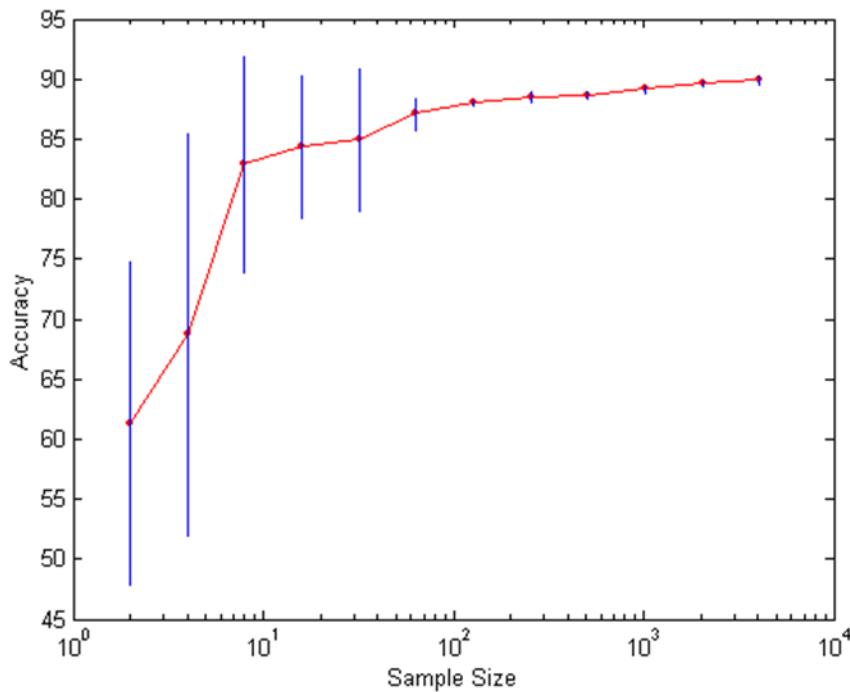
$$F_{\text{SCORE}} = \frac{2a}{2a + b + c} \quad (7)$$

Jednou z nejlépe vypovídajících hodnot pro vzájemné srovnávání výsledků je **Mathewův korelační koeficient**:

$$MCC = \frac{N_{TP} \times N_{TN} - N_{FP} \times N_{FN}}{\sqrt{(N_{TP} + N_{FP})(N_{TP} + N_{FN})(N_{TN} + N_{FP})(N_{TN} + N_{FN})}} \quad (8)$$

Jediné na co je zapotřebí si dát pozor, aby některá z hodnot v děliteli nezpůsobila, že celý jmenovatel bude roven hodnotě 0. V takovém případě je vhodné nastavit hodnotu výsledku součtu ve jmenovateli na 1.

Při trénování je třeba mít na paměti, že přesnost naučeného algoritmu lze dosáhnout nejen použitím parametrů algoritmů a vhodných učících se metod, ale také, že příliš malá trénovací množina může vést k značné míře chyby.

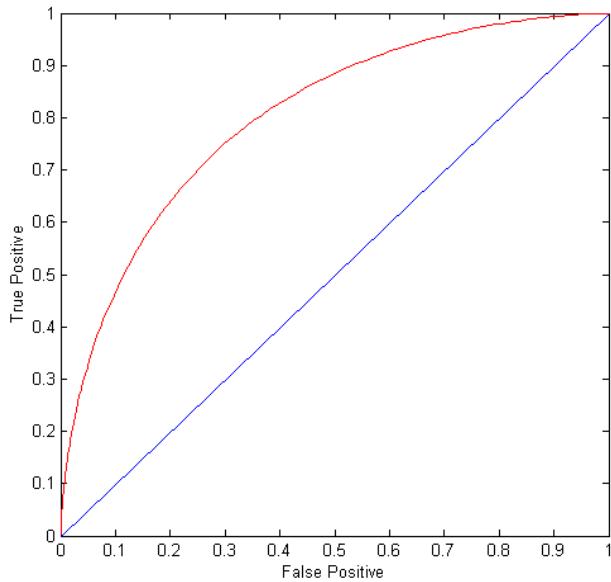


Obrázek 6: Závislost přesnosti na velikosti trénovací množiny.

Zejména v případě menšího počtu trénovacích vzorků může být členění výsledků na skupiny TP, FN, TN, FP **nepříliš vhodné**. Zanedbává se tím informace, s jakou pravděpodobností se ta či ona třída předpovídala. V takových případech se může hodit vyjádření pomocí ROC, AUC, RSME a dalších (viz dále).

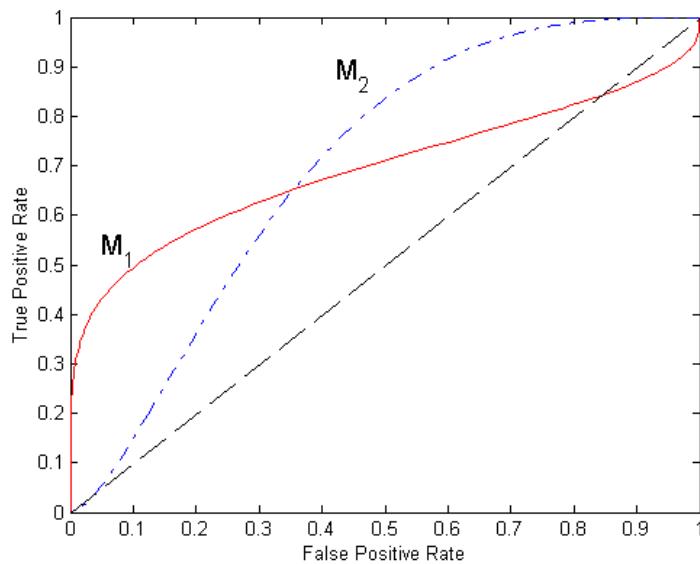
12.6.1 ROC křivky

ROC křivka je grafickým znázorněním poměru mezi TP a FP v závislosti na prahování – tj. míře jistoty, kterou naučený model uvádí při klasifikaci do dvou tříd. Ukázka ROC křivky je znázorněna na Obrázek 7. Modrá křivka znázorňuje naprosté náhodné hádání, tj. předpověď FP četnost vůči TP četnosti je 50 : 50. Čím výše červená křivka je, tím je větší přesnost daného algoritmu. Pokud se křivka dostane dokonce pod modrou čáru, znamená to, že model dokonce předpovídá opačně, než je skutečnost.



Obrázek 7: Ukázka ROC křivky

S využitím ROC křivek lze také poměrně přehledně porovnávat různé klasifikační modely mezi sebou. Učící se algoritmus nemusí dávat v různých případech použití stejné parametry přesnosti. Jak je patrné z Obrázek 8, algoritmus M1 dává lepší výsledky pro malý poměr falešných předpovědí, zatímco M2 dává výrazně lepší výsledky v případě vyšších hodnot poměru falešných předpovědí.



Obrázek 8: Srovnání dvou naučených modelů M1 a M2.

Příklad:

Mějme následující tabulkou dat, kde data byla klasifikována s následujícími hodnotami: $P(+|A)$ představuje předpověď naučeného modelu:

Vzorek	$P(+ A)$	Skutečná třída
1	0.95	+

2	0.93	+
3	0.87	-
4	0.85	-
5	0.85	-
6	0.85	+
7	0.76	-
8	0.53	+
9	0.43	-
10	0.25	+

Pro vyjádření ROC křivky je nezbytné nejprve 1) seřadit hodnoty dle atributu $P(+|A)$ v rostoucím pořadí. Následně se postupuje od nejnižší hodnoty k vyšším, kde je zapotřebí vyjádřit FP, TN, FN a následně také vypočít poměr správné pozitivní a negativní předpovědi na základě prahovací hodnoty. V prvním kroku je prahovací hodnota 0,25.

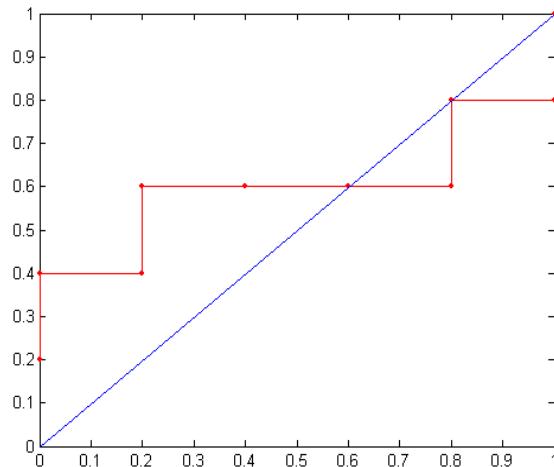
- Pod tímto prahem jsou hodnocena jako NEGATIVNÍ a nespadá sem žádný vzorek, můžu tedy rovnou napsat, že TN i FN je rovno nule,
- Nad hodnotou 0,25 jsou všechna ostatní ID a předpokládá se o nich, že jsou klasifikována jako POZITIVNÍ - tj. všechny ostatní 1 až 10. Skutečnost je ale v pořádku jen pro ID 1, 2, 6, 8 a 10 \Rightarrow TP je 5 vzorků. Pro ostatní, tj. ID 3, 4, 5, 7 a 9, je předpověď chybná \Rightarrow FP = 5 vzorků.

TP poměr, TPR = $TP / (TP + FN)$,

FP poměr, FPR = $FP / (FP + TN)$.

Ze získaných hodnot lze určit, že $FPR = 1$ a $TPR = 1$. Výsledek vyznačím v grafu na pozici (1,1). Takto postupuji tak dlouho, dokud nejsou do grafu vyneseny všechny hodnoty.

Třída	+	-	+	-	-	-	+	-	+	+	
	0.25	0.43	0.53	0.76	0.85	0.85	0.85	0.87	0.93	0.95	1.00
TP	5	4	4	3	3	3	3	2	2	1	0
FP	5	5	4	4	3	2	1	1	0	0	0
TN	0	0	1	1	2	3	4	4	5	5	5
FN	0	1	1	2	2	2	2	3	3	4	5
→ TPR	1	0.8	0.8	0.6	0.6	0.6	0.6	0.4	0.4	0.2	0
→ FPR	1	1	0.8	0.8	0.6	0.4	0.2	0.2	0	0	0



Z parametru ROC je možné odvodit funkci AUC (Area Under Curve, neboli plocha pod křivkou). Jedná se o velikost plochy, kterou červená křivka zabírá ve spodní části grafu. Hodnota 1 by značila naprostě ideální míru předpovědi, 0,5 značí zcela bezcenný naučený model (jeho výsledky se rovnají náhodnému hádání), hodnota 0 je naopak model, který dává zcela opačné výsledky, než které byly natrénovány. V takovém případě je možné dělat zcela opačná rozhodnutí, než které model předpovídá a dosáhneme tak ideálního modelu. V praxi ale takové chování není zcela doporučené a spolehlivost takového modelu by byla s vysokou pravděpodobností velice nízká.

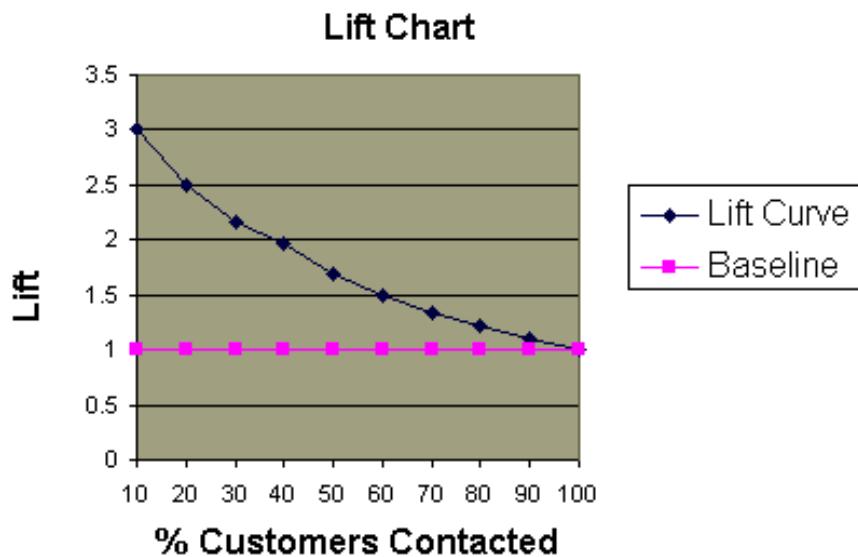
12.6.2 Lift grafy

Lift grafy vychází z ROC křivek a vyznačují, jaký je rozdíl mezi hodnotami získanými náhodným výběrem (tj. bez použití umělé inteligence) a s využitím námi natrénovaného algoritmu.

Příklad:

Po kontaktování 10% zákazníků bychom měli získat 10% kladných odpovědí při náhodném výběru. S použitím umělé inteligence při oslovování zákazníků bychom měli získat 30% kladných odpovědí. Hodnota y-ové osy lift křivky je tedy na 10 % rovna „s AI“ / „bez AI“ = $30 / 10 = 3$. Hodnotu vyneseme do grafu. Obdobně to provedeme pro další hodnoty.

Výhodou lift křivek je, že je na první pohled patrné, v které fázi je využití nejfektivnější a které pro nás má největší hodnotu.



Obrázek 9: Lift graf vs. základní hodnota.

Příklad 2:

Jméno dotazované osoby	Výška	Věk	Odpověď'
Alan	70	39	N
Bob	72	21	Y
Jessica	65	25	Y
Elizabeth	62	30	Y
Hilary	67	19	Y
Fred	69	48	N
Alex	65	12	Y
Margot	63	51	N
Sean	71	65	Y
Chris	73	42	N
Philip	75	20	Y
Catherine	70	23	N
Amy	69	13	N
Erin	68	35	Y
Trent	72	55	N
Preston	68	25	N
John	64	76	N
Nancy	64	24	Y
Kim	72	31	N
Laura	62	29	Y

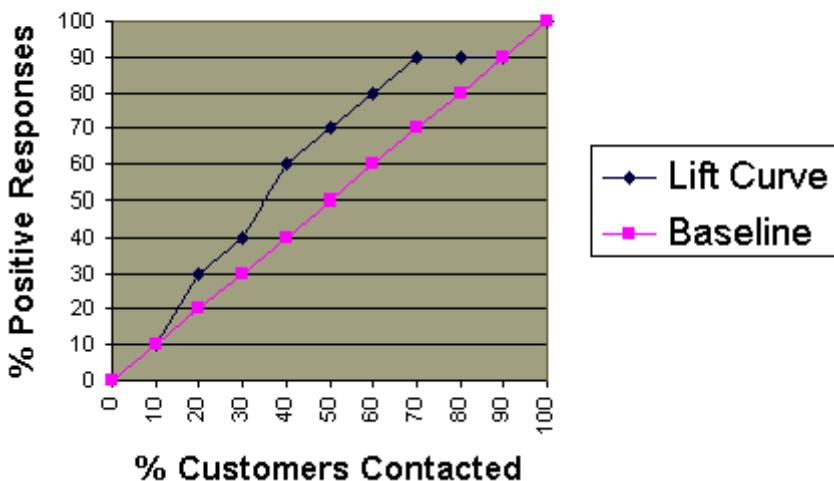
1. Zjistíme pravděpodobnost, s jakou systém předpokládá pozitivní odpověď od zákazníka.

- Seřadíme hodnoty sestupně podle tohoto ohodnocení pravděpodobnosti – hodnoty 0 – 50 představují předpověď, že zákazník neodpoví, 50–100 představuje předpověď, že zákazník odpoví:

Jméno dotazované osoby	Pravděpodobnost(x)	Věk
Alex	88	Y
Amy	87	N
Hilary	81	Y
Philip	80	Y
Bob	79	Y
Catherine	77	N
Nancy	76	Y
Jessica	75	Y
Preston	75	N
Laura	71	Y
Elizabeth	70	Y
Kim	69	N
Erin	65	Y
Alan	61	N
Chris	58	N
Fred	52	N
Margot	49	N
Trent	45	N
Sean	35	Y
John	24	N

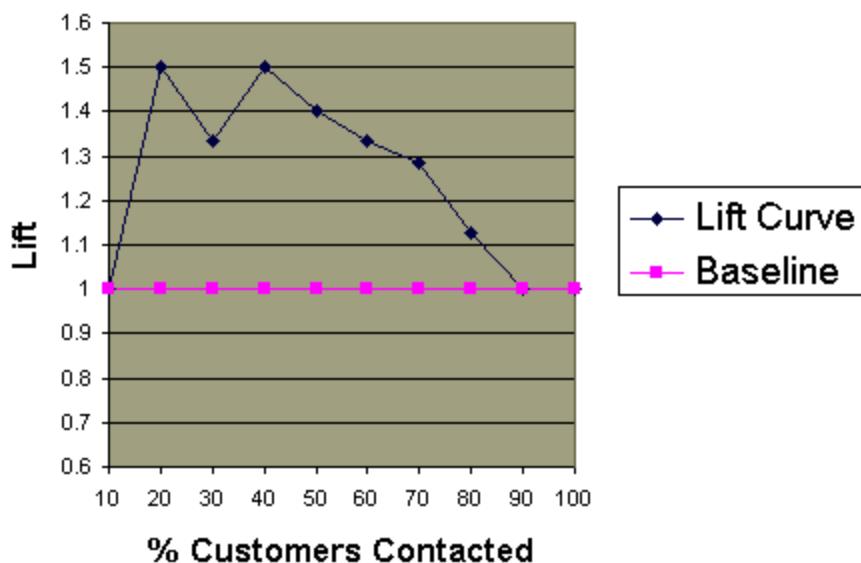
- Vyčíslíme počet kladných odpovědí na základě každé požadované hodnoty (10 %, 20 %, 30 %, 40 %, 50 %, ... 100 %).
- Vyjádříme hodnoty ve formě kumulativních hodnot:

Cumulative Gains Chart Problem 2



Obrázek 10: Kumulativní zisk na základě učícího se modelu vs. náhodného výběru.

- Na základě předchozího vyjádříme lift graf



Obrázek 11: Lift graf.

12.7 Učící se algoritmy

Významnou (nikoli stěžejní) úlohu v oblasti dolování znalostí z báze dat hrají učící se algoritmy. S využitím různých algoritmů lze pro různý typ dat a jejich charakter dosáhnout výrazně různých přesností a výsledků.

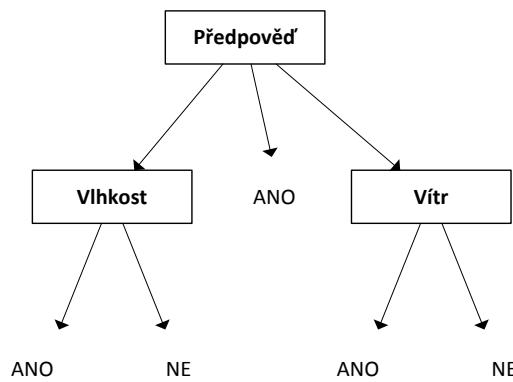
Učící se algoritmus ohodnocený metodami popisovanými dříve nemusí vždy dosahovat na nezávislých datech lepších výsledků. Je vždy na rozhodnutí experta, jaké metody jsou pro

daný typ problému vhodné. Ve většině případů dokáže osoba znalá charakteru dat dosáhnout výrazně lepších výsledků.

12.7.1 Rozhodovací stromy

Problematika rozhodovacích stromů je technika známá již řadu let. Největší předností této metody je snadná interpretovatelnost člověkem a to i v případech, kdy se nejedná o experty v oblasti dolování znalostí. Nevýhodou je relativně malá míra přesnosti a nepříliš uspokojivé výsledky.

Algoritmů pro indukci stromových struktur je v současné době řada. Mezi nejvýznamnější patří Huntův algoritmus, CART [11], ID3[12], C4.5 [13],[14], SLIQ[15], [16], SPRINT [17]. Podoba rozhodovacího stromu může vypadat následovně. Řekněme, že chceme na základě minulosti předpovědět budoucnost:



Obrázek 12: Příklad rozhodovacího stromu (prosím, abyste brali samotný obsah s rezervou :-).

Pokud považujeme výše zmíněný model za spolehlivý a dotážeme se některé osobě na otázky v rámci rozhodovacího stromu, jsme schopni určit, koho s největší pravděpodobností volil.

Při vytváření rozhodovacího stromu se vychází z trénovacích dat. Při posuzování stromů se berou v potaz vztahy:

Při vytváření rozhodovacího stromu se vychází z trénovacích dat. Při posuzování stromů se berou v potaz vztahy:

- GINI index ($p(j|t)$ je relativní frekvence třídy j v uzlu t .)

$$GINI(t) = 1 - \sum_j [p(j|t)]^2 \quad (9)$$

Popřípadě (CART, SLIQ, SPRINT) (n je počet všech záznamů v rodičovském uzlu p , n_i je počet záznamů spadající pro podčást stávajícího uzlu i):

$$GINI_{split} = \sum_{i=1}^k \frac{n_i}{n} GINI(i) \quad (10)$$

- Entropie (míra neuspořádanosti), ($p(j|t)$ je relativní frekvence třídy j v uzlu t .)

$$\text{Entropy}(t) = -\sum_j p(j|t) \log p(j|t) \quad (11)$$

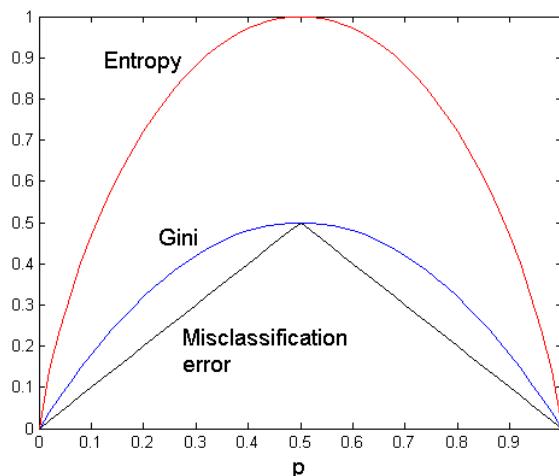
Respektive informačního zisku (ID3, C4.5):

$$GAIN_{split} = \text{Entropy}(p) - \left(\sum_{i=1}^k \frac{n_i}{n} \text{Entropy}(i) \right) \quad (12)$$

Nevýhodu je, že mají tendenci vytvářet komplexní stromy s velkým množstvím členění (**prořezávání stromu (pruning)**).

- Klasifikační chybě ($p(j|t)$) je relativní frekvence třídy j v uzlu t .)

$$\text{Error}(t) = 1 - \max_i P(i|t) \quad (13)$$



Obrázek 13: Srovnání hodnot pro rozdelení prvku. Entropie má tendenci generovat komplexnější stromy, zatímco klasifikační chyba naopak jednodušší stromy.

Parametry:

Minimální velikost rozdelení – minimální velikost uzlu, který může být ještě rozdelen.

Minimální velikost listu stromu – minimální velikost všech listů stromu.

Minimální zisk – minimální hodnota zisku, aby bylo povoleno rozdelení.

Maximální hloubka – maximální povolená hloubka stromu.

Důvěryhodnost – úroveň důvěryhodnosti použitý pro rozhodnutí o prořezávání stromu (pruning).

12.7.1.1 Klasifikační pravidla

Klasifikační pravidla jsou jiným zápisem rozhodovacích stromů. Zápis je členěn nikoli do stromu, ale do tabulky, kde zápis jednotlivých hodnot je ve formě:

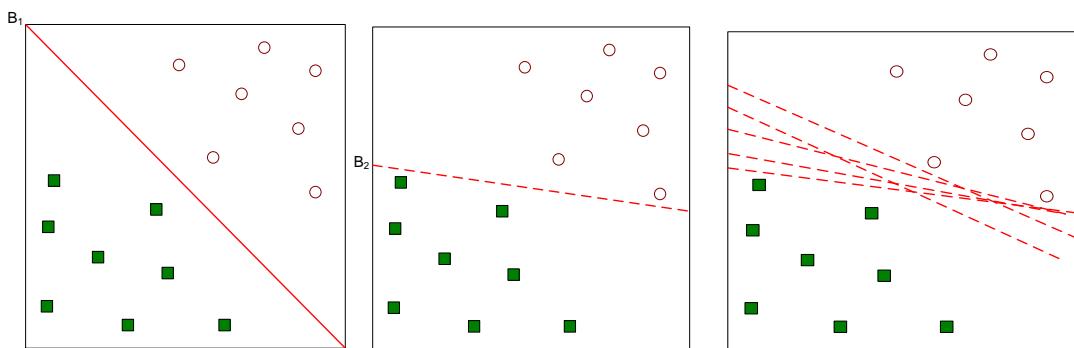
(podmínka) → třída

Vcelku jednoduchým přepisem pravidel z rozhodovacího stromu lze dosáhnout klasifikačních pravidel.

12.7.2 Algoritmy podpůrných vektorů

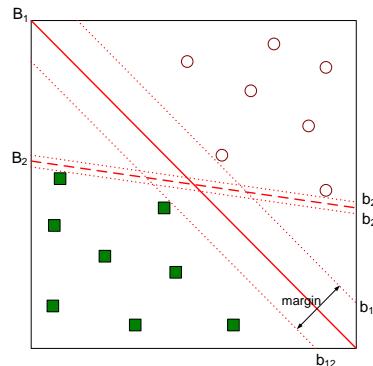
Algoritmy podpůrných vektorů (**Support Vector Machine**, SVM) patří k poměrně novým metodám strojového učení, které tvoří kategorii tzv. jádrových algoritmů (kernel machines). Tyto metody vychází z matematické teorie pro nalezení lineární hranice a zároveň jsou schopny reprezentovat vysoce složité nelineární funkce. Základním principem je převod daného původního prostoru do jiného, vícedimensionálního, kde již lze od sebe oddělit třídy lineárně.

Princip této myšlenky je znázorněn na Obrázek 14. Otázkou je, jak nejlépe zvolit oddělovací hranici těchto prostorů tak, aby byly vedeny efektivně z hlediska kategorizace budoucích dat, které při trénování nebyly použity. Samotná optimalizace se opírá o matematický aparát, který je nad rámec tohoto textu.



Obrázek 14: Při klasifikaci je možné vytvořit libovolný počet oddělovacích polorovin. Jen jedna bude mít na nové případy nejlepší výsledky.

SVM řeší tento problém nalezením poloroviny, která se snaží o maximalizaci okrajů vůči podpůrným vektorům a potom tedy rozdělení pomocí B_1 je lepší než pomocí B_2 .



Obrázek 15: Příklad vhodně a nevhodně zvoleného rozdělení hyper-roviny.

Snahou je nalézt takové řešení, které

$$\vec{w} \cdot \vec{x} + b = 0 \quad \dots \text{na přímce } B_1$$

$$\vec{w} \cdot \vec{x} + b = -1 \quad \dots \text{na podpůrném vektoru dole } b_{12}$$

$$\vec{w} \cdot \vec{x} + b = +1 \quad \dots \text{na podpůrném vektoru dole } b_{11}$$

Klasifikace je pak prováděna na základě vztahu:

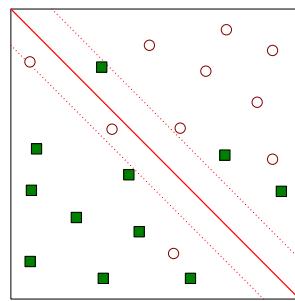
$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b \geq 1 \\ -1 & \text{if } \vec{w} \cdot \vec{x} + b \leq -1 \end{cases} \quad (14)$$

Okraje (Margin) je možné vyjádřit pomocí:

$$\text{Margin} = \frac{2}{\|\vec{w}\|^2} \quad (15)$$

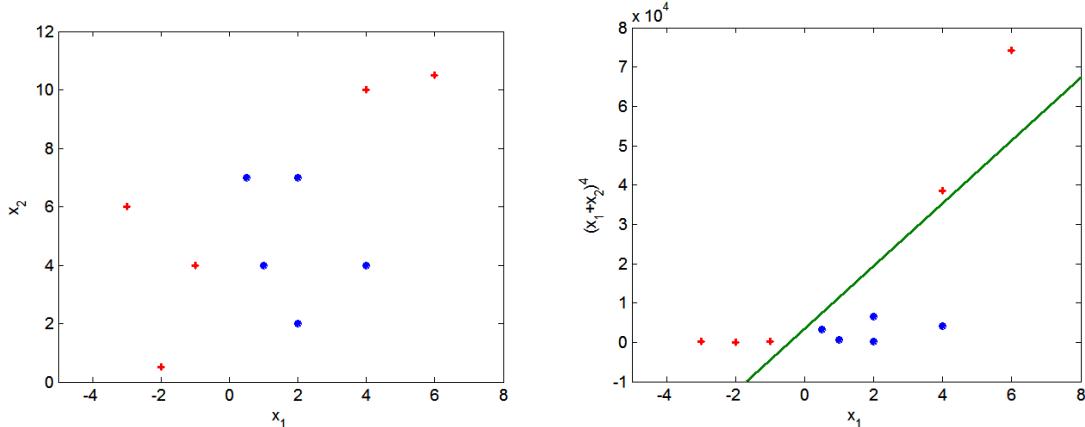
V případě, že problém není lineárně separovatelný (většina případů), zavádí se tzv. laxní proměnné:

$$f(\vec{x}_i) = \begin{cases} 1 & \text{if } \vec{w} \bullet \vec{x}_i + b \geq 1 - \xi_i \\ -1 & \text{if } \vec{w} \bullet \vec{x}_i + b \leq -1 + \xi_i \end{cases} \quad (16)$$



Obrázek 16: Příklad ideálního rozdělení hyper-roviny s použitím podpůrných vektorů.

V těch případech, kdy oddělovací pravidlo nemá nelineární charakter, je prostor převeden do vyšší dimenze, kde je možné oddělení:



Obrázek 17: Příklad rozdělení hyper-roviny s použitím nové dimenze.

Ve výše popsaném případě vycházíme z **lineárního jádra** SVM algoritmu. Další varianty jsou:

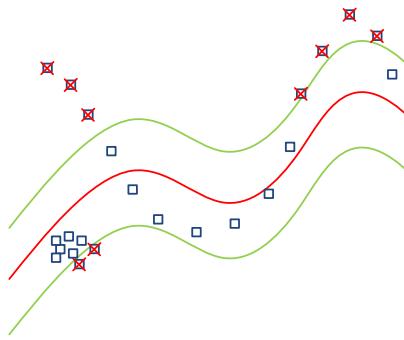
- Radiální
- Bodové
- Neuronové
- Anova
- Gausovské
- a další...

Parametrů pro optimalizaci klasifikace je celá řada a mnoho z nich závisí na typu zvoleného jádra. Ve všech systémech podpůrných vektorů se však setkáme s parametrem C (složitost) a

ξ . Výkonnost SVM zobecnění (přesnost odhadu) výrazně závisí na vhodném nastavení parametrů C , ξ a parametrů jádra. Výsledná složitost navrženého modelu bohužel závisí na všech těchto parametrech současně. Výběr hodnot těchto parametrů bývá zpravidla ponechán na uživateli klasifikátoru a měl by mimo jiné odrážet i distribuci vstupních trénovacích dat.

Parametr C určuje kompromis mezi složitostí modelu (jeho hladkosti) a mírou odchylek větších než ξ , které jsou tolerovány optimalizační rovnici. Například, jeli C příliš velké (nekonečno), potom je snahou optimalizace minimalizovat riziko na základě zkušenosti (počet použitých podpůrných vektorů roste), bez ohledu na složitost takového modelu v optimalizační rovnici.

Parametr ξ ovlivňuje šířku tzv. ξ -necitlivé oblasti, která se používá k nastavení vzorků trénovacích dat. Z tohoto důvodu hodnota ξ může ovlivnit počet podpůrných vektorů pro konstrukci klasifikátoru / regresivní funkce. Větší ξ znamená, že bude vybrán menší počet podpůrných vektorů. Na druhou stranu, větší hodnoty ξ ústí ve více vyhlazené předpovědi. Z tohoto pohledu oba parametry, C i ξ ovlivňují složitost modelu (ale odlišně).



Obrázek 18: Systém podpůrných vektorů. Zeleně je vyznačena ξ -necitlivá oblast. Modré body představují vzorky trénovacích dat. Přeškrtnuté body jsou takové, které se neberou v úvahu při optimalizačním procesu.

12.7.3 k nejbližším sousedů (k-NN, K Nearest-Neighbor)

Základní myšlenka vychází z předpokladu – chodí-li jako kachna, kváká-li jako kachna, pak je to nejspíše kachna.

Pro klasifikaci potom potřebuje následující:

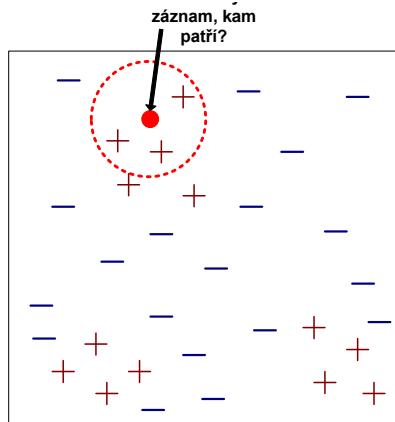
- Množina známých (již klasifikovaných) vzorků
- Definice metriky, se kterou lze počítat vzdálenost mezi vzorky:
 - Euklidovská

$$d(p, q) = \sqrt{\sum_i (p_i - q_i)^2}$$

- Camberrovská
- Chebíkovská
- Korelační
- Kosínova
- Manhatonovská

- A mnoho dalších
- Číslo k

Pro výpočet se určí k nejbližších sousedů a na základě tříd jim přiřazených se zvolí třída neznámého vzorku (nejčastěji dle převládajícího hlasu – dobré volit liché číslo k). Trénovací vzorky je dobré nejprve normalizovat.



Obrázek 19: Princip k-NN, kde na základě okolí bodu je stanovena výsledná klasifikace.

k-NN patří do množiny líných klasifikátorů (lazy learners) – nevytváří explicitně modely. Je poměrně snadné vložit / odstranit prvek do trénovací množiny, samotná klasifikace ale stojí poměrně značné výpočetní prostředky.

Parametry:

k – počet sousedů, kteří ovlivňují výslednou klasifikaci.

metrika – způsob, kterým se počítá výsledná klasifikace modelu (například Euklidovská, Manhatonovská atp.).

12.7.4 Bayesovské sítě

Bayesovské sítě vychází z teorie teorie pravděpodobnosti. Demonstrujme jeho princip na příkladu:

Příklad:

Známo:

- Doktor ví, že meningitida způsobuje zduření lymfatických uzlin v 50% případů
- Pravděpodobnost, že někdo má meningitidu je 1/50 000
- Pravděpodobnost, že nějaký pacient má zduřelé lymf. Uzliny je 1/20

Příchozí pacient má zduření lymfat. uzlin. Má meningitidu?

Odpověď vychází z teorie podmíněné pravděpodobnosti a Bayesovského teorému. Tedy:

$$P(M | S) = \frac{P(S | M)P(M)}{P(S)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002$$

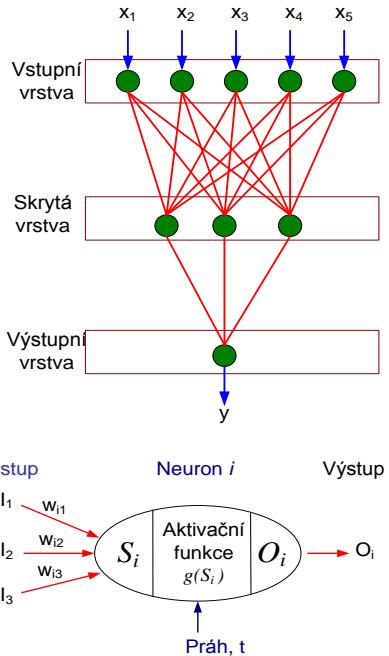
Pro spojité data je nutné

- Diskretizovat,
- anebo rozdělit,

- anebo předpovědět pravděpodobnost rozdělení.

V současnosti existují zpravidla lepší varianty nežli Bayesovské sítě.

12.7.5 Neuronové sítě (Vícevrstvá perceptronová síť)



Obrázek 20: Struktura neuronových sítí

Učení neuronové sítě potom závisí na přizpůsobení vah jednotlivých neuronů, resp. aktivační funkce a prahování. Nejčastějším algoritmem je backpropagation.

Parametry:

Skryté vrstvy – popisuje počet a velikost skrytých vrstev.

Počet cyklů – počet trénovacích cyklů pro algoritmus backpropagation.

Rychlosť učení – Jak rychle se mění váhy při každém cyklu.

Hybnost – přidává k aktuálnímu výsledku váhy zlomek z předchozí hodnoty váhy (je prevencí proti uvíznutí v lokálním minimu).

Epsilon – hodnota chyby, při které má být učení ukončeno.

Výhodí nastavení v prostředí RapidMiner

- 1 skrytá vrstva, sigmoidní typ, počet neuronů skryté vrstvy se počítá automaticky při hodnotě -1 (počet atributů + počet tříd) / 2 + 1)

12.8 Číselná predikce

SVM typu nu-SVR

Neuronové sítě

12.9 Shluky

- k-means
- k-medoids
- Shlukování shora dolů
- Náhodné shlukování

12.10 Transformace

I přes skutečnost, že v rámci převážné literatury je problematice transformace věnováno relativně málo prostoru, zpravidla se jedná o nejzdlouhavější a nejpracnější část práce. Pokud by byla tato část zahrnuta do doložení znalostí, zahrnovala by téměř všechny vědní obory. Často je totiž nutné datum perfektně rozumět (odborníci pro danou oblast jsou často označováni jako doménoví experti) a vědět, co je zapotřebí zkoumat.

V této kapitole jsou popsány jen metody obecné pro všechny skupiny. Mějte ovšem na paměti, že špičková analýza by se bez porozumění dané problematiky neměla obejít.

12.11 Výběr atributů (příznaků)

- Hrubou silou
- Dopředný výběr příznaků
- Zpětný výběr příznaků
- Genetické algoritmy
- Optimalizace pomocí teorie hejn

12.12 Normalizace

- Z-transformace
- Rozsahová transformace
- Proporční transformace

SVM, rozhodovací stromy, neuronové sítě

12.13 Kombinování několika modelů

- Bagging
- Bagging s váhováním
- Boosting
- Voting
- Hierarchická klasifikace
- AdaBoost
- MetaCost

- Stacking

12.14 Další informace

12.14.1 Učení z rozsáhlých objemů dat

12.14.2 Dolování znalostí z textu a webu

Korpus – množina dokumentů.

Dokument – seznam slov jdoucí za sebou

Slovo / token – jedno konkrétní slovo.

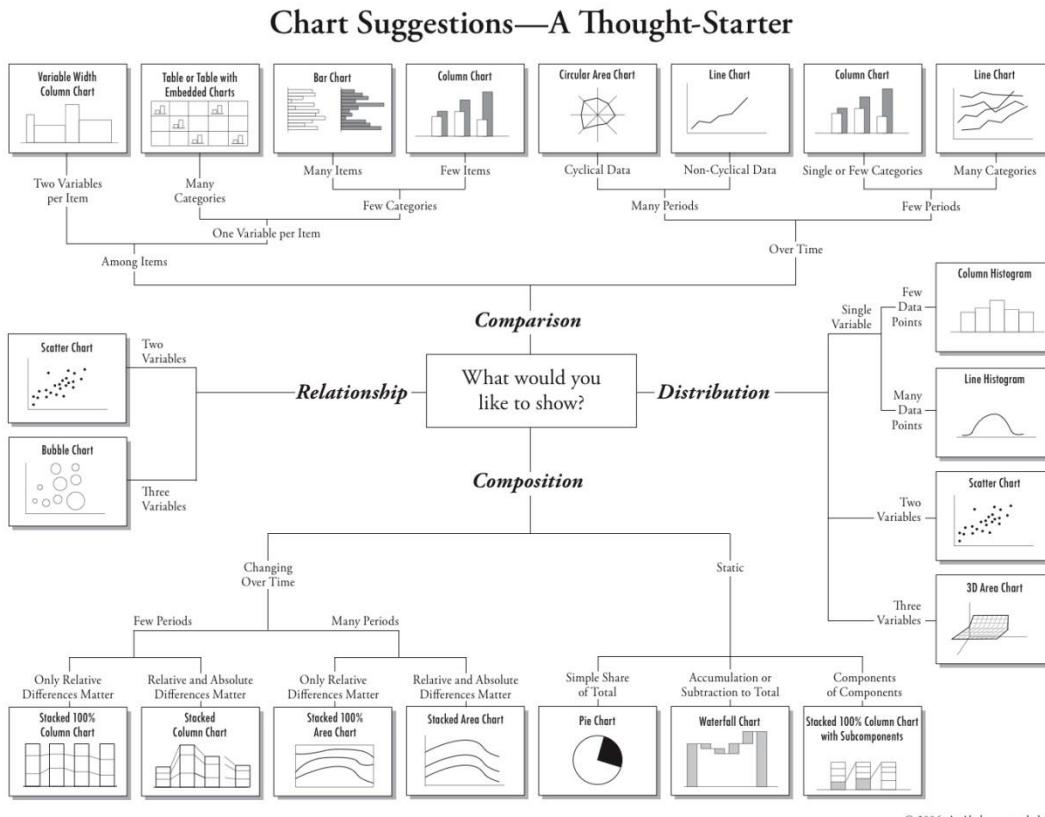
TF-IDF (term frequency-inverse document frequency)

Frekvence výskytu $\text{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$, kde $n_{i,j}$ je počet výskytů slova t_i v dokumentu d_j a jmenovatelem je sumou počtu všech výskytů všech slov v dokumentu. Inverzní frekvence dokumentu je: $\text{idf}_i = \log\left(\frac{|\mathcal{D}|}{|\{d: t_i \in d\}|}\right)$, kde čitatel $|\mathcal{D}|$ je počet dokumentů celkem, jmenovatelem $|\{d: t_i \in d\}|$ je počet dokumentů, ve kterém se vyskytuje slovo t_i . Jelikož tu hrozí dělení nulou, je v takovém případě lepší přičíst hodnotu 1.

TF-IDF se potom počítá následovně:

$$(\text{tf} - \text{idf})_{i,j} = \text{tf}_{i,j} \times \text{idf}_i \quad (17)$$

12.14.3 Vizualizace



12.14.4 Dolování znalostí z multimediálních dat

Na naší fakultě vzniká nástroj pro extrakci příznaků z obrazových dat.

12.14.5 Mezinárodní soutěž v dolování znalostí z báze dat

- DATA MINING CUP - <http://www.data-mining-cup.de/en/>
- ACM KDD CUP - www.sigkdd.org/kddcup/
- a mnoho dalších...

12.14.6 Nástroje

- RapidMiner, <http://rapid-i.com/>
- Weka, <http://www.cs.waikato.ac.nz/ml/weka/>
- Nástroj R pro statistické zpracování, <http://www.r-project.org/>
- KNIME,

12.15 Literatura

- [1] Ian H. Witten, Eibe Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, June 2005, ISBN 0-12-088407-0

-
- [2] J. Han and M.Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann
 - [3] Valery A. Petrushin, Latifur Khan, Multimedia Data Mining and Knowledge Discovery, Springer, 2006
 - [4] Man Leung Wong, Kwong Sak Leung, Data Mining Using Grammar Based Genetic Programming, Springer, 2000
 - [5] E. Thomsen, OLAP Solutions: Building Multidimensional Information Systems, Wiley; 2 edition, April 2002
 - [6] Mike Gunderloy, SQL Server Developer's Guide to OLAP with Analysis Services, Sybex, June 2001
 - [7] M. Schrader, D. Vlamic, M. Nader, C. Claterbos, Oracle Essbase & Oracle OLAP: The Guide to Oracle's Multidimensional Solution, McGraw-Hill Osborne Media; 1 edition, October 2009
 - [8] Ronen Feldman, James Sanger, The text mining handbook:advanced approaches in analyzing unstructured data, Cambridge University Press, 2007
 - [9] NELLO CRISTIANINI AND JOHN SHawe-TAYLOR. AN INTRODUCTION TO SUPPORT VECTOR MACHINES AND OTHER KERNEL-BASED LEARNING METHODS. CAMBRIDGE UNIVERSITY PRESS, 2000. ISBN 0-521-78019-5
 - [10] HUANG T.-M., KECMAN V., KOPRIVA I. (2006), KERNEL BASED ALGORITHMS FOR MININGHUGE DATA SETS, SUPERVISED, SEMI-SUPERVISED, AND UNSUPERVISED LEARNING, SPRINGER-VERLAG, BERLIN, HEIDELBERG, 260 PP. 96 ILLUS., HARDCOVER, ISBN 3-540-31681-7
 - [11] BREIMAN, LEO; FRIEDMAN, J. H., OLSHEN, R. A., & STONE, C. J. (1984). CLASSIFICATION AND REGRESSION TREES. MONTEREY, CA: WADSWORTH & BROOKS/COLE ADVANCED BOOKS & SOFTWARE. ISBN 978-0412048418
 - [12] QUINLAN, J. R. 1986. INDUCTION OF DECISION TREES. MACH. LEARN. 1, 1 (MAR. 1986), 81-106.
 - [13] QUINLAN, J. R. C4.5: PROGRAMS FOR MACHINE LEARNING. MORGAN KAUFMANN PUBLISHERS, 1993.
 - [14] S.B. KOTSIANTIS, SUPERVISED MACHINE LEARNING: A REVIEW OF CLASSIFICATION TECHNIQUES, INFORMATICA 31(2007) 249-268, 2007
 - [15] M. MEHTA, R. AGRAWAL, AND J. RISSANEN, "SLIQ: A FAST SCALABLE CLASSIFIER FOR DATA MINING", IN PROC. EDBT, 1996, PP.18-32.
 - [16] CHANDRA, B.; VARGHESE, P.P.ON IMPROVING EFFICIENCY OF SLIQ DECISION TREE ALGORITHM.NEURAL NETWORKS, 2007. IJCNN 2007. INTERNATIONAL JOINT CONFERENCE ON 12-17 AUG. 2007 PAGE(S):66 – 71

- [17] JOHN C. SHAFER, RAKESH AGRAWAL, MANISH MEHTA: SPRINT: A SCALABLE PARALLEL CLASSIFIER FOR DATA MINING. VLDB 1996: 544-555