Vypracované okruhy z PPA2

2008

Poměr použitých zdrojů:

Přednášky PPA2 (p. Šafařík) - 0,00000000001%

Wikipedia a Google (internet) - 99,99999999999%

1. Problém, algoritmus, program

Problém: V reálném světě potřebujeme řešit otázky, které mají různý charakter. Některé z nich vyžadují nalezení řešení nebo rozhodnutí, např. zda nějaké řešení vůbec mají. Takovéto otázky obecně nazveme *problémy*.

Algoritmus: Jedním ze způsobů řešení takovýchto otázek je *algoritmus*. Jedná se o konečnou množinu příkazů, které vedou k řešení určitého problému. Jeden *příkaz* algoritmu vede k provedení množiny operací, které jsou vykonatelné mechanicky (tj. bez dalšího přemýšlení) a tedy mohou být vykonávány strojem.

Algoritmus má tyto vlastnosti:

1. Konečnost: Algoritmus musí skončit vždy po provedení konečného (někdy velmi velkého) počtu kroků. Pokud nemá tuto vlastnost nazveme takový předpis *výpočetní metoda*. Příkladem jsou reaktivní systémy, které opakovaně reagují se svým okolím (automat na kávu běží ve smyčce).
2. Jednoznačnost: Každý krok algoritmu musí být jednoznačně určen.
3. Vstup: Každý algoritmus má žádný nebo více vstupů.
4. Výstup: Každý algoritmus má jeden nebo více výstupů.

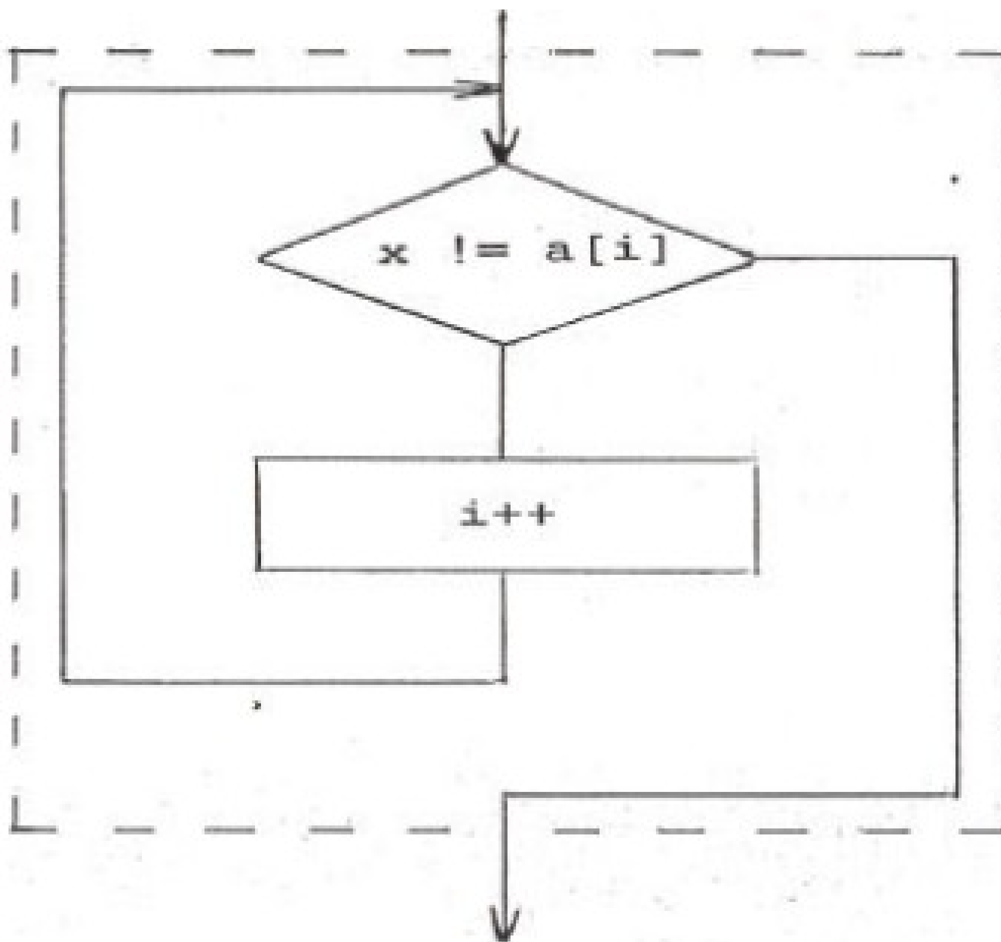
Program: Po vzniku počítačů bylo nutné algoritmus zapsat tak, aby jeho příkazy odpovídaly operacím počítače. Takový zápis byl ovšem pro člověka nesrozumitelný, proto byly formálně definovány (vyšší) *programovací jazyky*. Zápis, který vyhovuje pravidlům programovacího jazyka nazveme *program*. Obecně se jedná o výpočetní metodu, pokud splňuje kritérium konečnosti, jde o algoritmus.

2. Vykonání programu

Programovací jazyk na to, aby mohl být zápisem obecného algoritmu nebo výpočetní metody, musí obsahovat prostředky pro vyjádření příkazů, kterým říkáme řídící struktury a určují posloupnost vykonávaných kroků. (obecně tady jde tedy o cykly, větvení atd.)

Obecně, kterým krokem budeme pokračovat může záviset na vyhodnocení nějaké podmínky. Graficky toto rozhodování vyjadřujeme kosočtvercem se dvěma výstupními šipkami. Jedna pro případ, že podmínka je splněna a druhá pro její nesplnění. Řízení vykonávání následujících výpočetních kroků v programu můžeme graficky vyjádřit následovně:

```
while (true) {
        if (x != a[i])
         i++;
        else break;
}
```



Rozhodovací blok vnáší dvojrozměrnost. Tradičně na vyjádření, kterým krokem se má pokračovat, se v programovacích jazycích používal příkaz go to.

Pokud v grafu programu LS, jeho podgraf ohraničený čárkovaně vyjadřující opakování budeme považovat za blok, dostáváme jednorozměrnou posloupnost bloku. Taková struktura je přirozenější lidskému porozumění než obecné dvourozměrné struktury s množstvím rozhodovacích bloků s odevzdáváním řízení na libovolná místa, čemu se říkalo *spaghetti code*.

Tvorba programu, které jsou srozumitelné, využívajíc sekvence příkazů, příkazy alternativy a příkazy opakování se nazývá *strukturované programování*. (takže for, while, if atd.)

3. Objekt, třída:

Objekt: Objekt je entita skládající se z proměnných, zvaných *členské proměnné (member variables)*, a příslušejících funkcí, zvaných *metody (methods)*. V členských proměnných je obsažen stav objektu, tj. vše, co si objekt "pamatuje", a metody vyjadřují jeho chování, tedy vše, co objekt "umí".

Třída: V programovacím jazyce je třída (class) přesně popsána. Z hlediska syntaxe je třída obyčejnou deklarací datové struktury rozšířenou o metody. Třída představuje objektový typ, objekt se též nazývá *instancí třídy*. Třída se tedy skládá z kolekce datových položek a kolekce metod, které vykonávají akce nad těmito po položkami.

Příklad definice třídy v Javě:

```
class Auto0 {
  float obsahNadrze;
  float spotreba;
  void ujelo(float vzdalenost) {
    obsahNadrze = obsahNadrze - vzdalenost / 100.0F * spotreba;
  }
}
```

Pomocí operátoru **new**, za kterým následuje jméno třídy, vytvoříme instanci této třídy, tedy dynamicky se vytvoří objekt této třídy a vrátí se reference na nově vytvořený objekt. Hodnota reference je uchována v referenční proměnné:

```
Auto0 a;
```

a můžeme jí přiřadit hodnotu vrácenou operátorem *new*

```
a = new Auto0();
```

Objekt, který takto vytváří instanci jiné třídy se nazývá klientský program.

Metody, které mají stejný název jako třída, jsou *konstruktory*. Umožňují inicializovat datové členy. Konstruktor je možné *přetížit*, což umožňuje inicializovat objekt různými způsoby. Přetížené metody mají stejné jméno, ale liší se počtem, typem nebo pořadím formálních parametrů.

4. Spojové datové struktury:

Součást reálného světa není často možné popsat jedním parametrem, ale pomocí více parametrů. Tato abstrakce má obecný název *záznam* a říkáme, že jednotlivá data uchováváme v *položkách záznamu*. Např. informace o studentovi:

jméno

příjmení

pohlaví

…

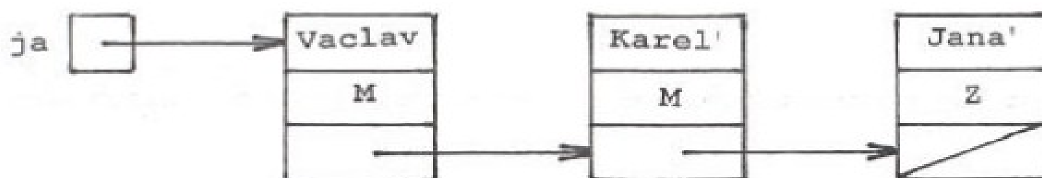
Tímto způsobem je tedy vyjádřena množina hodnot, které může záznam o studentovi nabývat, jako kartézský součin množin hodnot datových typů jednotlivých položek záznamu a jde tedy o nová datový typ. Jazyky C nebo Pascal umožňují přímo definovat záznam (record), abstrakci záznamu v Javě definujeme pomocí tříd, kde položky záznamu jsou reprezentovány členskými proměnnými.

Klientský program pracuje s daty jenom pomocí metod příslušných tříd. Schéma této koncepce vychází z následující abstrakce:

- **rozhraní** definuje požadované metody
- **implemetnace** metod rozhraní
- **klientský program** využívá metod rozhraní pro práci na vyšší úrovní abstrakce

Je-li v Javě vytvořeno pole, jeho velikost již nelze měnit. Někdy nemůžeme dopředu odhadnout

počet datových jednotek, které budeme ukládat, protože tyto dynamicky vznikají a zanikají, využívá se proto mechanismus spojování a takové struktury se nazývají *spojové*. Každá datová jednotka obsahuje položku zvanou ukazatel, která ukazuje na jinou datovou jednotku.



V Javě lze spojové struktury reprezentovat tak, že každý objekt obsahuje referenční členskou proměnnou, ve které bude uložen odkaz na jiný objekt. Referenčních proměnných může být obecně více.

5. Správnost programů:

???

6.Analýza programů:

Vývoj počítačů vytvořil potřebu rozvoje teorie, která věnuje pozornost problematice:

- analýzy algoritmu (programu)
- složitosti algoritmu
- hledání optimálních algoritmů

Analýza algoritmu = určení požadovaných prostředků na jeho vykonání, jakými mohou být čas, šířka pásma, paměťové nároky atd. Nejčastěji nás zajímá *čas výpočtu*.

Uvažujme výpočet algoritmu na PC s jedním CPU, standardním instrukčním souborem a obvyklým zobrazením čísel. Čas výpočtu obecně roste s velikostí vstupních dat, například počtem prvků, které se mají seřadit. Čas výpočtu algoritmu potom bude dán počtem vykonaných elementárních kroků.

$c_{operace}$=elementární počet kroků k vykonání určité operace

je-li provedena n-krát, pak přispěje k času vykonání $nc_{operace}$

Splnitelnost výrokových formulí:

```
if(booleovský výraz) příkaz;
```

Nechť se booleovský výraz skládá z:

- n proměnných logického typu $v_1,v_2,\dots v_n$
- m logických operátorů and, or, …
- závorek

Jestliže přiřazení hodnoty logické proměnné trvá c_h a čas každé logické operace c_o, pak vyhodnocení výrokové formule trvá

$$T(m,n) = nc_h + mc_o$$

Výroková formule je splnitelná tehdy, existuje-li přiřazenu hodnot *true* a *false* logickým proměnným $v_1...v_n$ tak, že hodnota výrokové formule (log. Výrazu) bude *true*.

Problém: zjistit, zda je výroková formule splnitelná

Algoritmus: pro všechna možná přiřazení hodnot proměnným $v_1...v_n$ vyhodnotit výraz

Počet takových přiřazení je 2^n , pak celková doba výpočtu bude:

$$T(m,n) = 2^n(nc_h + mc_o)$$

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 zplodit pomalé tím pádem nefunkční monstrum.

Ne vždy je ale možno určit přesnou složitost algoritmu, nebo složitost algoritmu nezávisí jenom na velikosti, nebo množství dat, ale přímo na jejich hodnotách.

Asymptotická složitost vyjadřuje limitní růst času výpočtu, pokud velikost problému neomezeně roste.

Asymptotické omezení shora i sdola (Theta notace):

Funkce g(n) je $\Theta(f(n))$ existují-li kladné konstanty c_1, c_2 a n_0 takové, že $0 \le c_1 f(n) \le g(n) \le c_2 f(n)$ pro všechna $n > n_0$.

T notace omezuje funkci shora i zdola.

Př.:

$g(n) = an^3+bn^2$, čas výpočtu takového algoritmu je tedy $\Theta(n^3)$ (n^2 přispívá málo, může se vynechat)

Výpočet:

pak tedy $c_1n^3 \le an^3+bn^2 \le c_2n^3$ a,b>0, teď vydělíme n^3 a získáme

$c_1 \le a+b/n \le c_2$

$c_1 \le a$, $c_2 \ge a+b/n_0$, kde $n_0 > 0$

Asymptotické omezení shora (Omikron notace):

Funkce f(n) je O(f(n)) existují-li kladné konstanty c, n_0 takové, že $0 \le g(n) \le cf(n)$ pro všechna $n > n_0$.

$g(n) = an^3+bn^2$ je tedy $O(n^3)$, ale i $O(n^4)$, …

Asymptotické omezení sdola (Omega notace):

Funkce f(n) je $\Omega(f(n))$ existují-li kladné konstanty c, n_0 takové, že $0 \le cf(n) \le g(n)$ pro všechna $n > n_0$

Polynomiální algoritmy:

Čas, který potřebují ke zpracování vstupních dat algoritmy považované za rychlé, bývá zpravidla shora omezen funkcemi typu n, n.log (n), n^2, ... Jako horní hranici lze vždy použít polynom , proto

se těmto algoritmům říká polynomiální nebo také prakticky použitelné.

Exponenciální algoritmy :

Pomalé (prakticky nepoužitelné) algoritmy užívají metodu „hrubé síly“ (brute force), kterou probírají všechny možnosti.

7. Rekurze:

O rekurzi mluvíme je-li něco částečně složeno nebo definováno pomocí sebe samého. Rekurze představuje opakované vnořené volání stejné funkce (podprogramu), v takovém případě se hovoří o rekurzivní funkci. Nedílnou součástí rekurzivní funkce musí být ukončující podmínka určující, kdy se má vnořování zastavit.

Rekurze vs. Iterace

Rekurze často poskytuje elegantnější a jednodušší řešení problému než iterace. Rekurzivní řešení je obvykle méně efektivní než iterační (vzhledem k paměťovým a časovým nárokům). Rekurze je mocným nástrojem pro řešení problémů, které jsou rekurzivně definovány. Rekurze může být ekonomickým řešením problému, pokud není hloubka rekurzivního volání příliš velká.

Rekurzivní funkce:

Faktoriál :

a) 0! = 1,

b) n! = n.(n-1)!, pro n >0.

Čas výpočtu rekurzivního algoritmu je O(n).

Rekurze je elegantním vyjádřením výpočtu, ale rekurzivní volání metody je časově i paměťově náročné. Její přímočaré použití muže být velice nevhodné.

Rekurzivní průchod seznamem:

Rekurzivní definice množiny hodnot datového typu umožňuje rekurzivní vyjádření práce s těmito hodnotami. Častým úkolem je průchod všemi prvky datové struktury. V případě seznamu, jeho přímý průchod s tiskem hodnot procházených prvku můžeme napsat:

```
void pruchod(Prvek x) {
   if (x == null)
       return;
   x.tiskPrvku( );
   pruchod(x.dalsi);
}
```

Obecně, je-li posledním krokem metody *M(x)*, volání *M(y)*, můžeme volání *M(y)* nahradit přiřazením *x = y* a skokem na začátek metody *M*.

8. Abstraktní datové typy:

Různé definice ADT:

Kromě základních datových typů existují i komplexnější *abstraktní* datové typy. Ty nejsou určeny potřebami hardware, ale potřebami programátorů. Umožňují abstrakci dat (zjednodušený pohled na data), což je nezbytné pro psaní větších programů, protože lidský mozek je schopen řešit úlohy jen do určité míry složitosti. Jednoduchým příkladem abstrakce dat budiž datový typ bod, který se skládá ze dvou čísel (souřadnic) a který může programátor používat při programování grafiky.

Zdůrazněme, že jde o princip, který je nezávislý na konkrétním jazyce i když jsme ho demonstrovali na jazyce, který používáme. *ADT je matematický model* společně s operacemi nad tímto modelem. Používání ADT nám umožňuje abstrahovat od konkrétní reprezentace dat a implementace operací nad těmito daty.

ADT je matematická struktura složená ze dvou částí - třídy a operace nad prvky.

ADT je v informatice výraz pro typy dat, které jsou nezávislé na na vlastní implementaci. Hlavním cílem je zjednodušit a zpřehlednit program, který provádí operace s daným datovým typem. ADT umožňuje vytvářet i složitější datové typy, např. operace s ADT typu *zásobník*, *fronta* a *pole*.

ADT vyžadují oddělení jednotlivých vrstev abstrakce, co znamená přistupování k datům ADT jenom přes rozhraní, které tvoří jeho operace.

ADT je datový typ, který je přístupný jenom přes rozhraní. Program, který používá ADT nazýváme klientem a program, který je realizací ADT nazýváme implementací.

Nejdůležitější vlastnosti abstraktního typu dat jsou:

- Všeobecnost implementace: jednou navržený ADT může být zabudován a bez problémů používán v jakémkoliv programu.
- Přesný popis: propojení mezi implementací a rozhraním musí být jednoznačné a úplné.
- Jednoduchost: při používání se uživatel nemusí starat o vnitřní realizaci a správu ADT v paměti.
- Zapouzdření: rozhraní by mělo být pojato jako uzavřená část. Uživatel by měl vědět přesně co ADT dělá, ale ne jak to dělá.
- Integrita: uživatel nemůže zasahovat do vnitřní struktury dat. Tím se výrazně sníží riziko nechtěného smazání nebo změna již uložených dat.
- Modularita: „stavebnicový" princip programování je přehledný a umožňuje snadnou výměnu části kódu. Při hledání chyb mohou být jednotlivé moduly považovány za kompaktní celky. Při zlepšování ADT není nutné zasahovat do celého programu.

Pokud je ADT programován objektově, jsou většinou tyto vlastnosti splněny.

Na abstraktním datovém typu rozlišujeme tři druhy operací: *konstruktor*, *selektor* a *modifikátor*. Operace, která ze zadaných parametrů vytváří novou hodnotu abstraktního datového typu, se nazývá konstruktor. Úkolem konstruktoru je sestavení platné vnitřní reprezentace hodnoty na základě dodaných parametrů. Operace označovaná jako selektor slouží k získání hodnot, které tvoří složky nebo vlastnosti konkrétní hodnoty abstraktního datového typu, a konečně operace typu modifikátor provádí změnu hodnoty datového typu.

V ADT tedy nelze napsat `Auto.spotreba = 10;` ale musí být vytvořena metoda (klasické getry a setry), která proměnnou `spotreba` nastaví, tím je tedy myšleno již zmiňované rozhraní ADT:

```
void zmenaSpotreby (float novaSpotreba) {
  spotreba = novaSpotreba;
}
```

Vyžadujeme ovšem také, aby některá data nebyla klientovi přístupná vůbec, popřípadě aby byla přístupná pouze pomocí metody. viz výše. To nám umožňují jazyky které umožňují změnu přístupových práv (private, protected, public). Pak tedy definujeme:

```
private float spotreba;
```

a tato proměnná bude přístupná pouze uvnitř třídy.

Metody, které musí ADT implementovat lze v Javě určit pomocí *interface*, kde najdeme signatury (hlacičky) příslušných veřejných metod.

9. Zásobník, fronta, seznam:

Jednosměrný seznam:

Pod pojmem *seznam* rozumíme takovou dynamickou strukturu, která umožňuje vložit nebo odebrat libovolný prvek. Kromě ukazatele na začátek a na konec dynamické struktury si můžeme zavádět další ukazatele podle toho, jak budeme chtít se strukturou pracovat.

Operace potřebné pro manipulaci se seznamem:

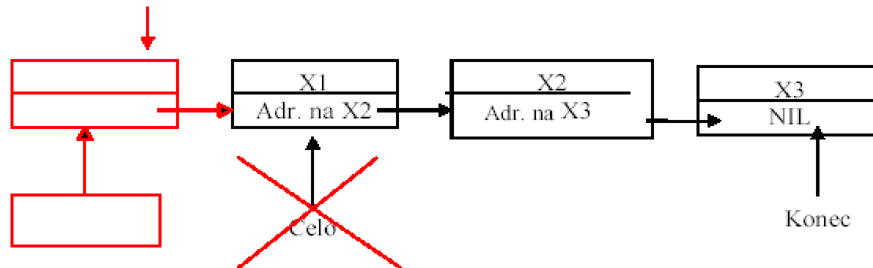
- vytvoření prázdného seznamu
- přidání nového prvku na konec seznamu
- odebrání prvku z čela seznamu
- test prázdnosti seznamu
- odstranění libovolného prvku ze seznamu
- vložení nového prvku na libovolné místo v seznamu
- spojení dvou seznamů v jednu dynamickou strukturu

```
// rozhrani ADT seznam
Seznam()                 // vytvorení prázdného seznamu
boolean jePrazdny( )     // test je-li seznam prázdný
void tiskSeznamu( )      // tisk prvku seznamu
void naZacatek( )        // nastavení okamžité pozice na
                        // první prvek
boolean jePosledni( )   // test je-li okamžitá pozice
                        // nastavena na poslední prvek
void naDalsiPrvek( )    // nastavení okamžité pozice na
                        // pozici následujícího prvku
```

```
int ctiKlic( )             // precti klíc prvku na okamžité
                           // pozici
void vloz(int i)           // vlož prvek
int vyber( )              // vyber prvek
```

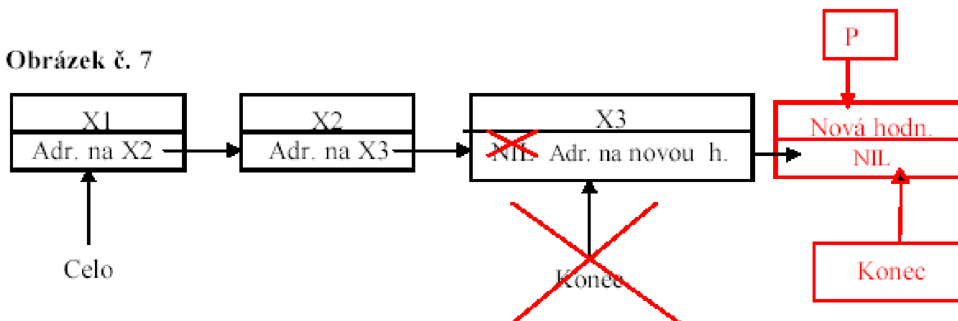
Přidání prvku na začátek struktury:

Na obrázku č. 8 je seznam po přidání nové hodnoty na začátek struktury:



Přidání prvku na konec struktury:

Na obrázku č. 7 je seznam po doplnění nové hodnoty na konec struktury:



Obousměrný seznam:

Při ukládání dat formou obousměrně zřetězeného seznamu je součástí záznamu o daném prvku kromě adresy následníka navíc i adresa jeho předchůdce.

Takto sestavená dynamická struktura:

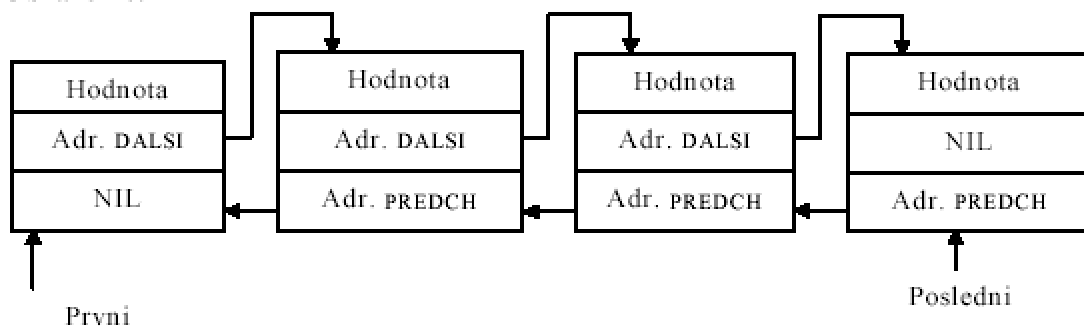
- zvětšuje nároky na paměť tím, že pro uložení každé hodnoty musíme rezervovat místo pro danou hodnotu a pak místo pro dvě adresy – adresu prvku, který následuje a adresu prvku, který předchází
- výhodou je ale pak to, že vždy známe nejen adresu následníka, ale i adresu předchůdce. Při práci se strukturou pak odpadá nutnost zavádění a umísťování pomocných ukazatelů za účelem

zjištění adresy předchůdce.

Na obrázku č. 15 je ukázka obousměrně zřetězeného seznamu, ke kterému je přístup přes dva ukazatele – *Prvni* obsahuje adresu prvního prvku a *Posledni* obsahuje adresu posledního prvku.

Struktura je sestavena tak, že:

- adresní část pro předcházející prvek obsahuje u prvního prvku hodnotu NULL
- adresní část pro následující prvek obsahuje u posledního prvku hodnotu NULL



Operace potřebné pro manipulaci s obousměrným seznamem:

- Vložení prvního prvku do seznamu
- vložení nového prvku na konec seznamu
- vložení nového prvku na začátek seznamu
- vložení nového prvku na libovolné místo v seznamu
- odebrání prvního prvku seznamu
- odebrání posledního prvku seznamu
- odstranění libovolného prvku ze seznamu
- spojení dvou seznamů v jednu dynamickou strukturu

Implementace spojového seznamu pomocí polí :

Pro jednoduchost uvažujme lineární spojový seznam. Na implementaci jeho prvku s položkami klíč a další použijeme dvě stejnojmenná pole a prvek seznamu bude reprezentován dvojicí jejich prvků se stejným indexem. Je-li typ klice char, tyto pole budou

```
char[] klic = new char[max];
int[] dalsi = new int[max];
```

V prvcích pole další je ukazatel na následující prvek seznamu, tj. index prvku polí klíč a další , ve kterých je uložen. Je-li prvek seznamu poslední bude další obsahovat hodnotu -1.

Příklad

int jmeno = 3;

klic[3] = 'P'; dalsi[3] = [4];

klic[4] = 'A'; dalsi[4] = [7];

klic[7] = ‘V’; dalsi[7] = [2];

klic[2] = ‘E’; dalsi[2] = [6];

klic[6] = ‘L’; dalsi[6] = [-1];

Fronta:

Fronta slouží k ukládání dat, které vystupují ze struktury ve stejném pořadí, jako vstupují. Odborně se nazývá FIFO - First In First Out (někdy také jako tunel). Lze si ji představit jako frontu lidí, kteří postupují k pokladně. Jak se řadí do fronty, tak i odcházejí od pokladny.

Základní operace s frontou:

- vytvoření prázdné fronty
- vložení prvku na konec fronty (přidání, zápis)
- odebrání prvku na začátku fronty (čtení)
- test prázdnosti fronty (abychom nemuseli číst, když tam nic není)

Implementace fronty pomocí pole:

Pole při tomto řešení má omezenou délku (definujeme proměnné na začátku programu ve formě např. pole:arrays[1..1000] of integer, takové pole může mít maximálně 1000 míst pro uložení dat ve frontě). Tím, jak data do pole zapisujeme a čteme, posouváme právě zapsaná data polem až ke konci. Tam se zastaví. Jednoduchým řešením by bylo při každém přečtení všechny položky v poli posunout na začátek, to ovšem není nejlepší, především u velkých polí s množstvím dat. Proto se používá tzv. *kruhová fronta*, neboli, když dojde zápis na konec pole, zapisuje opět na volné místo na začátku. Celé to funguje jako had, který na konci zalézá do díry a opět vylézá na začátku. Nesmíme nikdy zapomenout na důležitost kontroly plnosti fronty, resp. jestli místo, ze kterého čteme nebo do kterého zapisujeme, je plné (obsahuje data), jinak následuje podtečení nebo přetečení fronty.

zacatek – index odkud vybereme prvek

konec –index kam uložíme prvek

V případě, že fronta je prázdná nebo plná jsou si tyto indexy rovny.

Čas každé z operací nad frontou implementovanou pomocí pole je O(1).

Implementace fronty pomocí spojového seznamu:

Tento způsob vytvoření fronty je poněkud složitější, ale odpadá zde problém s velikostí pole. Místo pole se používá ukazatel do operační paměti. Tímto lze zapisovat data tak dlouho, dokud máme dostatek operační paměti. Samotné vytvoření je založeno na klasické implementaci pomocí ukazatele.

zacatek – ukazatel odkud vybereme prvek

konec – ukazatel kam uložíme prvek

Čas každé z operací nad frontou implementovanou pomocí spojového seznamu je O(1).

Zásobník:

Základní operace se zásobníkem:

- vytvoření prázdného zásobníku
- vložení prvku na vrchol - push (přidání, zápis)
- odebrání prvku z vrcholu – pop (čtení)
- test prázdnosti zásobníku (abychom nemuseli číst, když tam nic není)

Implementace zásobníku pomocí pole:

Stejně jako u fronty se jedná o nejjednodušší formu zásobníku. Na rozdíl o fronty je zásobník celkově snadnější, protože nemusíme počítat s tím, že data budeme „točit“ z konce na začátek (viz výše, kruhová fronta). Budeme opět potřebovat pole, do něhož zapíšeme data postupně, ale číst je budeme od vrcholu zásobníku.

Čas každé z operací nad zásobníkem implementovaným pomocí pole je O(1).

Když vykonáme operaci pop nad prázdným zásobníkem říkáme, že došlo k podtečení *underflow* zásobníku. Obecně jde o chybový stav. V Javě lze na něj reagovat použitím mechanizmu výjimek. Když vykonáme operaci *push*, kdy v*rchol = maxN* říkáme, že došlo k přetečení *overflow* zásobníku. Jde o to, že jsme pro aplikaci dostatečně neodhadli potřebnou velikost zásobníku, na což lze opět reagovat pomocí mechanizmu výjimky.

Pokud narazíme na omezení velikostí pole, zle vytvořit tzv. *dynamické pole*, kdy pokud chceme provést operaci *push* nad plním zásobníkem, nejdříve pro něj vytvoříme pole nové o dvojnásobné velikosti, zkopírujeme staré pole do nového a pak teprve provedeme *push*.

Implementace zásobníku pomocí spojového seznamu:

Podobně jako u fronty můžeme zapisovat přímo do operační paměti za běhu aplikace. Nemusíme hlídat velikost pole, ale musíme hlídat volnou paměť. Princip leží na připojování a odpojovaní položek spojového seznamu.

Čas každé z operací nad zásobníkem implementovaným pomocí pole je O(1).

10. Stromy, průchody stromem, binární vyhledávací stromy:

Prvky v dynamické množině organizované jako strom nazýváme vrcholy. Některé vrcholy jsou spojeny a toto spojení nazýváme hranou. Strom potom můžeme rekurzivně definovat následovně:

a) Jeden vrchol je strom. Tento vrchol se nazývá kořen stromu.

b) Nechť x je vrchol a jsou stromy T1, T2, ..., Tn. Strom je vrchol x spojený s kořeny stromu T1, T2, ..., Tn.

V tomto stromě je x kořenem stromu a stromy T1, T2, ..., Tn se nazývají podstromy. Jejich kořeny jsou přímými následovníky vrcholu x a vrchol x je jejich přímým předchůdcem.

Vrchol, který nemá přímé následovníky se nazývá listem. Vrchol, který není listem se nazývá vnitřním vrcholem. Někdy je vhodné zahrnout mezi stromy i prázdnou množinu vrcholu.

Cesta je posloupnost vrcholu, ve které po sobe jdoucí vrcholy jsou spojeny hranou. Délka cesty je počet hran cesty. Délku cesty z vrcholu k sobe samému potom můžeme definovat jako nulovou. Ke každému vrcholu je z kořene právě jedna cesta. Hloubka vrcholu ve strome (úroveň, na které se nachází) je definována jako délka této cesty. Úroveň (hloubka) kořene stromu je tedy nulová.

Výška stromu je maximální hloubka vrcholu stromu.

Množina přímých následovníků muže být uspořádaná, například při grafickém zobrazení zleva doprava. Důležitou třídou takových stromu jsou binární stromy.

Def: Binární strom je prázdný strom anebo vrchol, který má levý a pravý podstrom, které jsou binární stromy.

Možná implementace vrcholu:

```
class Vrchol {
```

```
  int klic;
  Vrchol levy;
  Vrchol pravy;
}
```

3 možné druhy procházení (začneme kořenem):

Přímý pruchod (preorder) navštívíme vrchol, potom levý a pravý podstrom

Vnitřní průchod (inorder) navštívíme levý podstrom, vrchol a pravý podstrom .

Zpětný průchod (postorder) navštívíme levý a pravý podstrom a potom vrchol.

Rekurzivní průchod stromem:

```
void pruchodR(Vrchol v) {
    if (v == null)
       return;
    v.tiskVrcholu();
    pruchodR(v.levy);
    pruchodR(v.pravy);
}
Vrchol koren;
pruchodR (koren);
```

Uvedená metoda implementuje průchod *preorder*, posunutím řádku s tiskem mezi rekurzivní volání získáme implementaci průchodu *inorder* a jeho posunutím za obě rekurzivní volání získáme implementaci průchodu postorder.

Nerekurzivní průchod pomocí zásobníku:

```
void pruchod(Vrchol v) {
  VZasobnik z = new VZasobnik();
  z.push(v);
  while (!z.jePrazdny()) {
     v = z.pop();
     v.tiskVrcholu();
     if (v.pravy != null) z.push(v.pravy);
     if (v.levy != null) z.push(v.levy);
  }
}
```

Binární vyhledávací stromy (BVS) :

Vkládání prvku do lineárních implementací uspořádaných množin a hledání prvku v takových implementacích neuspořádaných množin je časově náročné. Pro BVS jsou v průměrném případě obě tyto operace efektivní.

Nechť x je vrchol stromu.

Je-li y vrchol v levém podstromu, potom *y.klíc* < *x.klíc*.

Je-li y vrchol v pravém podstromu, potom *y.klíc* > *x.klíc*.

Hledání ve stromu:

```
private String hledejR(DVrchol v, int klic) {
  if (v == null)
    return null;
  if (klic == v.klic)
    return v.data;
  if (klic < v.klic)
    return hledejR(v.levy, klic);
  else
    return hledejR(v.pravy, klic);
}
String hledej(int klic) {
  return hledejR(koren, klic);
}
```

Vložení prvku:

```
private DVrchol vlozR(DVrchol v, int klic,
                        String data) {
  if (v == null)
    return new DVrchol(klic, data);
    if (klic < v.klic)
      v.levy = vlozR(v.levy, klic, data);
    else
      v.pravy = vlozR(v.pravy, klic, data);
    return v;
  }
void vloz(int klic, String data) {
  koren = vlozR(koren, klic, data);
}
```

Obě metody při každém rekurzivním volání sestoupí o jednu úroveň níž a tedy složitost uvedených algoritmu je O(h), kde h je výška stromu.

11. Grafy a jejich implementace:

G = (V, H)

V – množina vrcholů (uzlů) |V| - počet vrcholů

H – množina hran |H| - počet hran

hrana je dvojice (u,v) u,v∈V

Platí-li (u,v) ≠ (v,u) je hrana orientovaná, tj. má začáteční a koncový vrchol

V(G) – množina vrcholů grafu G

H(G) – množina hran grafu G

Reprezentace grafu :

Dva standardní způsoby v informatice:

- soubor (collection ne file) seznamu sousednosti
- matice sousednosti

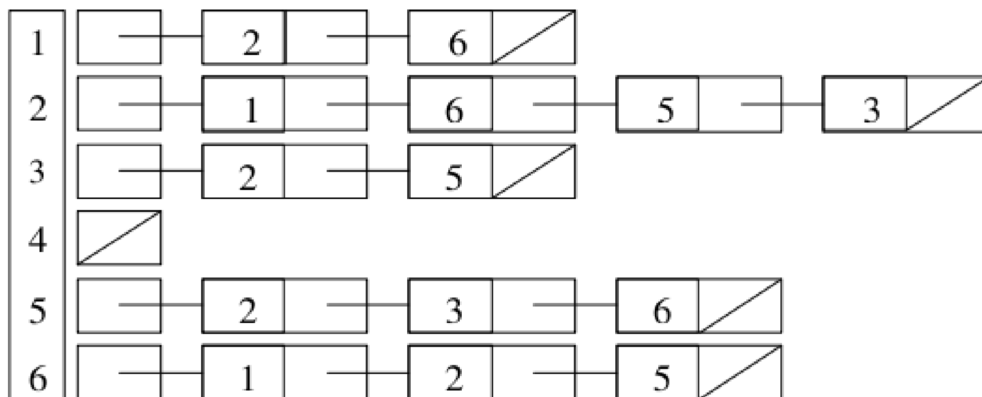
Oba způsoby jsou použitelné pro orientované i neorientované grafy.

Seznam sousednosti :

Pro každý vrchol je vytvořen seznam sousedů. Sousedící vrcholy jsou obecně uloženy v seznamech v libovolném pořadí.

Reprezentace neorientovaného grafu:

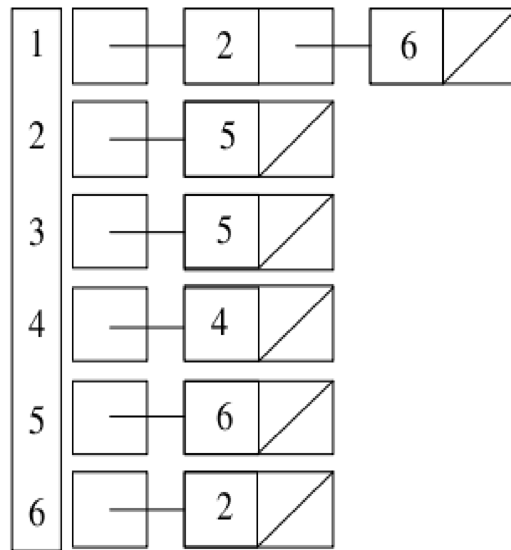
např. 1 je spojeno s 2 a 6; 2 je spojeno s 1, 6, 5 a 3 atd.



Celková délka seznamu sousednosti pro neorientovaný graf je 2|H|.

Reprezentace orientovaného grafu:

např. 1 ukazuje šipkou na 2 a 6 atd.




```
class Vrchol {
  ...
  Soused sousedi;
  Vrchol () {
  ...
  sousedi = null;
  }
}
```

Reprezentace seznamem sousednosti je vhodná i pro ohodnocené grafy.

Ohodnocení grafu G se nazývá funkce

w: H(G) -> R (zobrazení z množiny hran grafu G do reál. čísel)

Je-li (u,v)∈H(G), w(u,v) se uloží ve vrcholu v seznamu sousedu vrcholu u.

Potenciální nevýhodou je zjišťování existence hrany (u,v), což znamená hledat vrchol v v seznamu sousedu vrcholu u.

Matice sousednosti :

Označme vrcholy čísly 1, ..., |V|.

Matice sousednosti je matice S = (s_{ij}), i,j = 1, ..., |V|, přičemž je-li (i,j)∈H, s_{ij} = 1 jinak s_{ij} = 0

Reprezentace neorientovaného grafu :

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	1	1
3	0	1	0	0	1	0
4	0	0	0	0	0	0

5	0	1	1	0	0	1
6	1	1	0	0	1	0

Reprezentace orientovaného grafu :

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1
6	0	1	0	0	0	0

Graf je při této reprezentaci implementován dvourozměrným polem i*nt[][] s = new int [|V |] [|V |];*

Pro neorientovaný graf je $S = S^T$, kde S^T je transponovaná matice, což umožňuje snížit nároky na paměť téměř na polovinu.

Maticí sousednosti lze implementovat i ohodnocený graf. Pro ohodnocený graf je $s_{uv} = w(u,v)$, je-li $(u,v) \in H(G)$ (neboli na pozici v matici je přímo hodnota ohodnocení hrany), s_{uv} je hodnota mimo hodnot možných ohodnocení, je-li $(u,v) \notin H(G)$ (neboli na pozici v matici je např. nekonečno).

Je-li $|H| \ll |V|^2$ graf se nazývá řídký obvykle je vhodnější použít seznam sousednosti.

Je-li $|H| \sim |V|^2$ graf se nazývá hustý obvykle je vhodnější použít matici sousednosti.

Matici sousednosti je vhodnější použít také, je-li nutno rychle zjistit existenci hrany.

12. Prohledávání grafu:

PROHLEDÁVÁNÍ DO ŠÍRKY

BREATH-FIRST SEARCH

Z vybraného vrcholu *s* nalezneme všechny vrcholy ve vzdálenosti *k* od vrcholu *s* předtím, než nalezneme vrcholy ve vzdálenosti *k+1*. Nalezený vrchol obarvíme šedě a uložíme ho do fronty, přičemž začneme vrcholem *s*. Po nalezení všech sousedu vrchol obarvíme černě. Není -li fronta prázdná, pokračujeme dalším vrcholem z fronty.

Vlastnosti BFS algoritmu :

- nalezneme všechny dosažitelné vrcholy z vybraného vrcholu *s*
- vypočteme vzdálenost (počet hran) k objeveným vrcholům od *s*
- vytvoříme BFS strom všech dosažitelných vrcholu, kterého kořen je *s*
- v grafu G(V,H) definujeme délku nejkratší cesty mezi vrcholy $s,v \in V$ jako minimální počet hran všech cest z vrcholu *s* do vrcholu *v*

***P*ROHLEDÁVÁNÍ DO HLOUBKY**

DEPTH-FIRST SEARCH

Hledáme, když je to možné, napřed do „hloubky“, tj. do větší vzdálenosti a nalezené vrcholy obarvíme šedě . Po průchodu všemi hranami z vyšetřovaného vrcholu, vrchol obarvíme černě a vrátíme se k jeho předchůdci nebo skončíme.

DFS je pro orientované i neorientované grafy. DFS využívají jiné algoritmy. Jestliže po vykonání DFS zůstaly neobjevené vrcholy, jeden vybereme a postup opakujeme , vznikne tak les DFS stromu.

13. Topologické řazení:

Máme množinu prvků, ve které je definované uspořádaní jen pro některé dvojice – např. prvky jsou činnosti v čase. Uspořádaná dvojice prvků (u,v) potom tvoří hranu orientovaného acyklického grafu (directed acyclic graph – DAG) .

Algoritmus topologického řazení:

Úkol: Lineárně seřadit vrcholy (činnosti) tak, aby vzájemné poradí dvojic zůstalo zachováno.

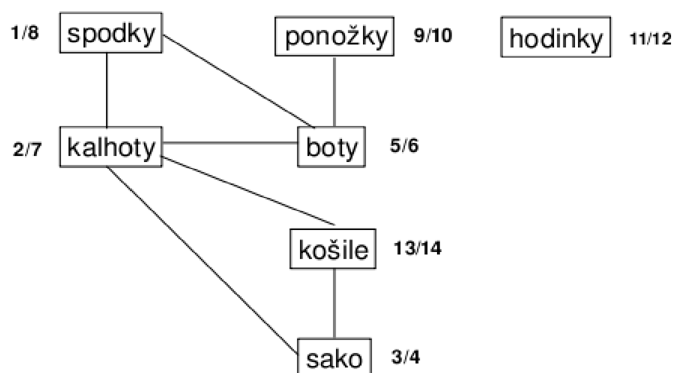
Algoritmus:

• Zavolej DSF a vytvoř les DSF stromů.

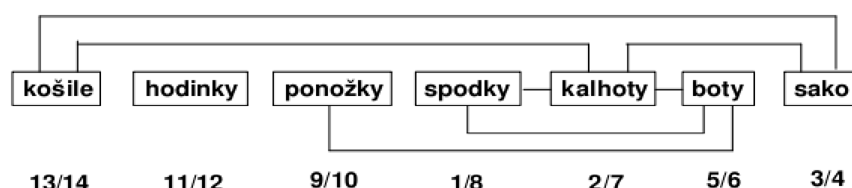
• Když je vrchol ukončen, přidej ho do na začátek seznamu.

• Vrať seznam vrcholů .

Po vykonání DFS získáme napríklad hodnoty casu *objevení/dokoncení* jednotlivých vrcholu na následujícím obrázku.



Tomu odpovídá následující lineární serazení vrcholu



Algoritmus: http://en.wikipedia.org/wiki/Topological_sorting

```
L ← Empty list where we put the sorted elements
Q ← Set of all nodes with no incoming edges
while Q is non-empty do
    remove a node n from Q
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into Q
if graph has edges then
    output error message (graph has a cycle)
else
    output message (proposed topologically sorted order: L)
```

14. Tabulka s přímým adresováním:

Přímé adresování je vhodné není-li velikost ¦K¦ množiny klíčů K velká. Obdobně, jak jsme předpokládali u BVS, klíče všech prvků dynamické množiny jsou různé a prvky jsou reprezentovány objekty třídy Prvek.

```
class Prvek {
   int klic;
   String data;
   ...
   Prvek(int klic, String data) {
       this.klic = klic;
       this.data = data;
   }
}
```

Rozhraní ADT tabulky s přímým adresováním bude tvořeno následujícími operacemi :

```
class Tabulka {
   // rozhrani ADT
  Tabulka()
   Prvek hledej(int)
  vloz(Prvek)
  vyber(Prvek)
}
```

Tabulka bude reprezentována polem *t* velikosti ¦K¦ , jehož prvky budou obsahovat reference na prvky tabulky, resp. null nejsou-li využity.

```
class Tabulka {
  Prvek[] t = new Prvek[¦K¦];
```

```
  Tabulka() {
     for(int i = 0; i < t.length; i++) {
        t[i] = null;
  }
}
```

Implementace dalších operací je zřejmá:

```
   Prvek hledej(int k) {
      return t[k];
  }
   void vloz (Prvek x) {
      t[x.klic] = x;
  }
  void vyber (Prvek x) {
      t[x.klic] = null;
  }
```

15. Rozptylové tabulky s vnějším řetězením :

Uvažujme situaci, že aktuální množina A prvků identifikovaných klíče je daleko menší než množina všech hodnot klíče, tedy ¦A¦ << ¦K¦. Pokud prvky množiny resp. ukazatele na ně budeme kvůli rychlosti přístupu chtít uchovávat v poli, abychom nemrhali pamětí, toto pole by melo mít rozměr úměrný ¦A¦ . Zatím ponechme stranou jeho velikost ve vztahu k poctu prvku aktuální množiny ¦A¦ a označme jeho velikost a tím i velikost tabulky m, přičemž m << ¦K¦ . V tabulce budou nyní prvky množiny identifikované svým klíčem zpřístupňovány pomocí indexu s hodnotami 0 až m-1. Potřebujeme tedy hodnotu klíče prvku transformovat na hodnotu indexu, jinými slovy musíme definovat funkci

h: K → {0,1, ... m -1}

kterou budeme nazývat rozptylovací funkcí (hash function).

Vhodné řešení je modulo (K%m).

Dvě kritéria na hash funkci:

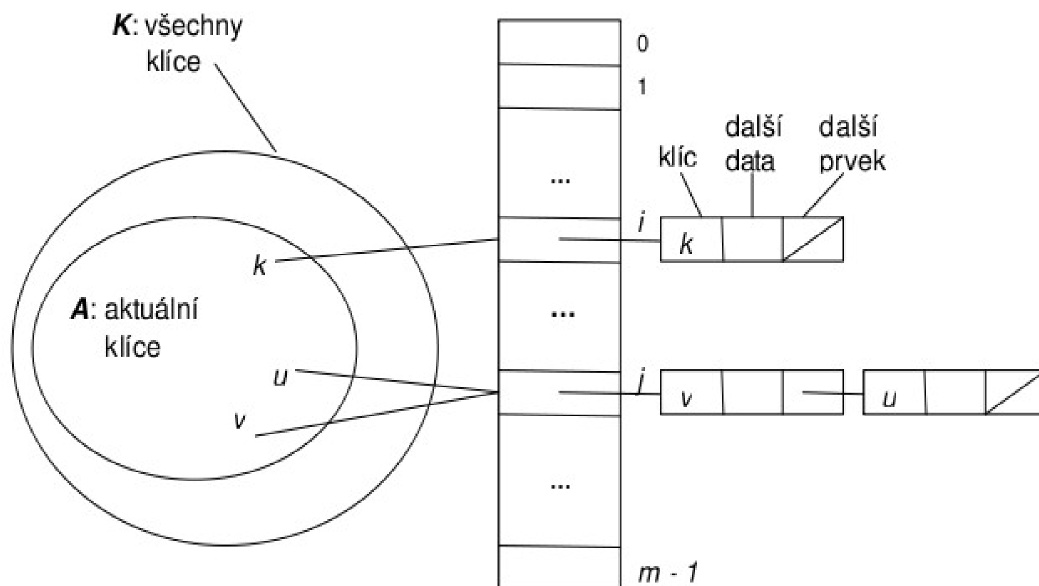
a) rychlý výpočet

b) minimální množství kolizí

Vzhledem k tomu, že m << ¦K¦ , tedy počet všech klíčů je daleko vetší než počet hodnot indexu, na které je zobrazujeme, nevyhnutelně musí rozptylová funkce zobrazit obecně dva a více různých klíčů na stejný index, tj u≠v a u,v∈K bude h(u) = h(v). Tuto skutečnost nazveme kolizí.

Vnější řetězení (separate chaining):

Kolize jsou při této metodě řešeny tak, že se vytvoří seznam prvků, jejichž klíče jsou zobrazeny na stejnou pozici v tabulce, což je znázorněno na následujícím obrázku.



Metody jsou stejné jako u seznamu, ovšem vše probíha přes hash funkci h().

```
Prvek hledej(int k) {
  hledej prvek s klícem k v seznamu t[h(k)]
}
void vloz (Prvek x) {
  vlož x na zacátek seznamu t[h(x.klic)]
}
void vyber (Prvek x) {
  vyber x ze seznamu t[h(x.klic)]
     }
```

16. Prioritní fronta:

Rozhraní ADT prioritní fronta obsahující prvky, které jsou objekty třídy Prvek je definováno následovně:

```
class PF {
    // ADT rozhrani
     PF();     // vytvorení prázdné prioritní fronty
     boolean jePrazdna();      // test, je-li prázdná
     void vloz(Prvek); // vložení prvku
```

```
    Prvek vybermax(); // výber nejvetšího prvku
}
```

Implementace pomocí pole:

Operace odebrání maxima se vykoná v konstantním čase. Vkládaní, je založeno na myšlence vložení prvku na konec seřazené fronty a jeho postupnou výměnnou s prvky před ním, pokud prvek před ním je větší, se nalezne pozice kam vkládaný prvek patří. Stejná myšlenka byla použita v algoritmu řazení vkládáním, kde se postupně každý prvek vložil na správné místo do podposloupnosti uspořádaných prvku před ním. V nejhorším případe nutno projít všechny prvky v prioritní frontě. Alternativně můžeme prioritní frontu implementovat neuspořádaným polem, kdy největší prvek budeme hledat až při jeho vybíraní.

Implementace pomocí spojového seznamu:

Zatímco při implementaci pomocí pole v obou alternativách jsme vkládali a vybírali prvek z konce pole, při implementaci neuspořádaným obousměrným spojovým seznamem, prvek vkládáme na jeho začátek a největší prvek po jeho nalezení přímo odebereme. Implementace pomocí uspořádaného spojového seznamu by odebírala prvek ze začátku seznamu a před jeho vložením by musela nalézt místo pro vložení.

Uchovávaní prvku prioritní fronty jako neuspořádané posloupnosti je příkladem tzv. trpělivého (lazy) přístupu k řešení problému implementace jejích operací, kdy to co v rámci dané operace nemusíme udělat odložíme na později. Na druhé straně, uchovávaní prvku prioritní fronty jako uspořádané posloupnosti je příkladem tzv. netrpělivého (eager) přístupu k řešení problému, kdy v rámci operace vykonáme co nejvíce práce potřebné pro efektivní implementaci jiných operací.

	vlož	vyber největší prvek	najdi největší prvek
uspořádané pole	N	1	1
neuspořádané pole	1	N	N
uspořádaný seznam	N	1	1
neuspořádaný seznam	1	N	N

Implementace pomocí stromu:

Efektivním uložením prvku dynamické množiny zohledňující velikost klíčů byl BVS, kde operace vložení a nalezení prvku se zadaným klíčem byly O(h), kde h je výška stromu. V případe ADT prioritní fronta je operace výběru prvku specifikována jako výběr prvku s největším nebo s nejmenším klíčem.

Prvek v BVS s největším klíčem nalezneme tak, že budeme z kořene sledovat pořád ukazatele na pravý podstrom až k listu, kde ukazatel je null. Vzhledem na vlastnost BVS, nemá-li vrchol x pravý podstrom, protože v levém podstromu jsou prvky s menším klíčem než x.klic, je x prvkem

s největším klíčem. Má-li x pravý podstrom, potom, protože v levém podstromu jsou klíče menší než x.klic, prvek s největším klíčem musí být v pravém podstromu s kořenem x.pravy.

```
int maxKlic() {
   DVrchol x = koren;
   while (x.pravy != null)
```

```
    x = x.pravy;
  return x.klic;
}
```

17. Halda:

Halda je stromová datová struktura splňující tzv. vlastnost haldy: pokud je B potomek A, pak x(A) >= x(B). To znamená, že v kořenu stromu je vždy prvek s nejvyšším klíčem (klíč udává funkce *x*). Tento stav nazveme *max heap*. Analogicky existuje *min heap*, kde je kořenem nejmenší prvek. Haldu lze efektivně reprezentovat v poli: na první pozici pole je uložen vrchol haldy (kořen stromu) a pro každý uzel k platí, že je v poli uložen na pozici k, jeho levý potomek je uložen na pozici 2k a pravý potomek na pozici (2k+1) a předchůdce na pozici k/2.

Máme-li prioritní frontu implementovanou haldou, potom nalezení největšího prvku je O(1). Jeho výběr však vyžaduje náhradu prvku, který byl kořenem, jiným prvkem, což může vést k porušení vlastnosti hlady, pokud tento prvek má menší klíč. Na druhé straně, vložíme-li prvek, bude tento dalším listem a pokud má vetší klíč než jeho předchůdce, opět došlo k porušení vlastnosti haldy.

Uvedené situace můžeme zobecnit na dva případy, kdy se poruší vlastnost haldy a nutno ji obnovit, chceme-li využívat její vlastnost.:

V prvním případe je vlastnost haldy porušena, protože klíč v některém vrcholu se stane menším než klíče v jednom nebo obou následnících. V tomto případě ho musíme vyměnit s větším z jeho následovníků, což může vést k porušení vlastnosti o úroveň níže a postup výměny opakujeme. Tímto postupem prvek, který takto porušil vlastnost haldy putuje směrem dolů, k listům. Jeho postup skončí, když bude splněna vlastnost haldy anebo se stane listem.

Ve druhém případě je vlastnost haldy porušena, protože klíč v některém vrcholu se stane vetším než klíč v jeho předchůdci. Pro obnovení vlastnosti haldy ho musíme vyměnit s jeho předchůdcem, což v tomto případe může vést k porušení vlastnosti haldy o úroveň výše a postup opakujeme. Tímto způsobem prvek, který porušil vlastnost haldy putuje směrem nahoru, ke kořenu. Skončíme, když se obnoví vlastnost haldy anebo se stane kořenem.

18. Algoritmy řazení O(N logN):

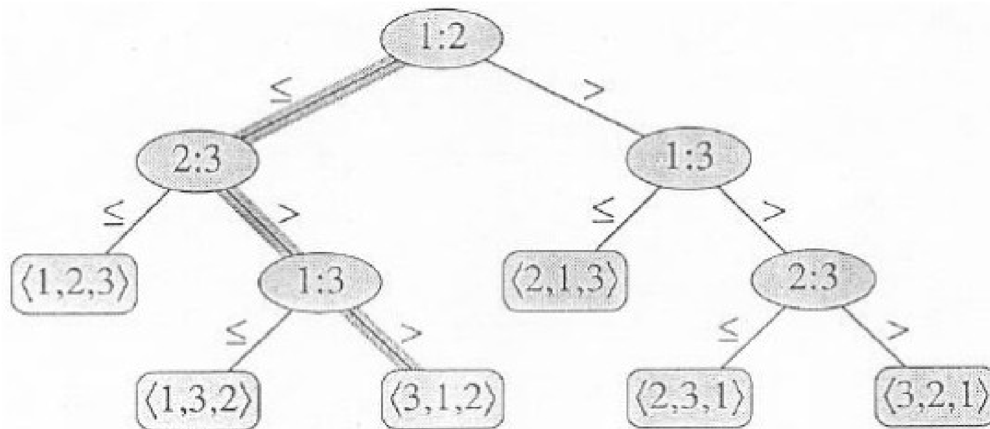
????

19. Dolní omezení pro porovnávací řazení:

Zkoumejme dolní omezení počtu porovnání T(n) pro porovnávací algoritmy řazení.

Předpokládejme, že všechny prvky posloupnosti a1, a2, ..., an jsou různé. Porovnávací řazení můžeme znázornit rozhodovacím stromem. Ve vnitřních vrcholech jsou prvky, které algoritmus porovná a v listech je permutace všech prvků původní posloupnosti, která je seřazena.

Příklad rozhodovacího stromu pro řazení vkládáním tří prvků:



Jakýkoliv správný algoritmus musí vytvořit každou z n! permutací prvků původní posloupnosti. Každá z nich musí být alespoň v jednom listě.

Nejhorším případem počtu porovnaní vykonaných algoritmem řazení je nejdelší cesta od kořene stromu k listu, je tedy roven výšce stromu T(n) = h . Nyní musíme najít dolní omezení všech rozhodovacích stromu. Nechť rozhodovací strom o výšce *h* má *l* listů, potom $l = 2^h$. Současně listů musí být alespoň tolik, kolik je permutací, tedy n! = *l*. Potom $n! = l = 2^h$ a logaritmováním

$$\log_2 (n!) = h = T(n)$$

Použitím aproximace $(n/e)^n$ pro n! je

$$\log_2(n!)=n*\log_2 n - n* \log_2 e,$$

čím dostáváme dolní omezení pro nejhorší případ $\Omega(n*\log_2 n)$.

Protože pro řazení haldou a slučováním je horní omezení času výpočtu $\Omega(n*\log_2 n)$ jsou tato řazení asymptoticky optimální.

20. Generičnost:

Genericita je možnost programovacího jazyka definovat místo typů vlastně jen „vzory typů“, kde určité typy, použité v definici typu, jsou vyvedeny vně definice jako parametry a jsou určeny později. Základním užitím genericity jsou třídy kontejnerů, které jsou určené k udržování skupin objektů určitého typu, například vzor třídy Seznam je definován vlastně jako Seznam[G], kde G je typ objektů, které mohou být do seznamu vloženy (kouzlo genericity vynikne pak v kombinaci s dědičností, kdy do seznamu mohou být vloženy nejen objekty typu G, ale i objekty všech možných podtypů typu G). Konkrétní typ, použitelný v textu programovacího jazyka, pak vzniká, když G nahradíme skutečným existujícím typem.

21: Dědičnost:

Dovoluje vybudování hierarchie tříd, které se postupně z generace na generaci rozšiřují(extends). Výhoda je že můžeme rozšiřovat už existující kód bez rizika že něco změníme v rodičovské třídě.

Obecný způsob vytvoření více specializované třídy je vytvoření podtřídy nějaké třídy, kterou potom nazveme nadtřídou (subclass, superclass). Také hovoříme o základní a odvozené třídě nebo o rodičovské třídě a třídě potomek. Tomuto procesu se říká dědičnost. Podtřída, odvozená třída, potomek dědí vlastnosti nadtřídy, základní třídy, rodiče. V Javě deklarujeme podtřídu B třídy A pomocí klíčového slova *extends*.

```
class B extends A {

...

}
```

Podtřída

- získává všechny metody a proměnné nadtřídy
- může přidat nové metody a proměnné třídy
- může předefinovat metody a proměnné rodiče

Pokud má rodičovská třída konstruktor s parametry, potomek má konstruktor, ve kterém je konstruktor rodiče volán pomocí *super()*. Pomocí *super* můžeme zpřístupnit v potomkovi proměnné i metody rodiče.

Polymorfismus:

Další důležitá vlastnost oop je polymorfismus. Třída může navenek vystupovat jako její rodičovská třída. To dovoluje mít např. pole objektů(typ Object) a do tohoto pole může dát jakýkoli objekt, protože Object je rodičovská třída všech tříd.

Předefinování metod – overriding:

Pokud třída potomka definuje třídu se stejnými vlastnostmi(specifikátor přístupu, návratová hodnota, jméno, parametry) jako její rodičovská třída, předefinuje tato metoda metodu v rodičovské třídě. K rodičovské metodě může přistupovat pomocí klíčového slova super (např. super.metoda()). Konstruktor třídy potomka v podstatě předefinuje kostruktor rodičovské třídy. Ten se ale automaticky volá jako první v konstruktoru potomka. Pokud má konstruktor přebírat nějaké parametry, musí se tam napsat manuálně(super(parametry)). Metody, které jsou definovány jako *final* nelze předefinovat.

Přetížení metod – overloading:

Uvažte následující metody:

```
public String pozdrav() {
    return pozdrav(false);
}

public String pozdrav(boolean pritel) {
    if(pritel) {
        return "ahoj";
    }
    return "dobry den";
```

}

Tyto metody se od sebe liší pouze parametry, které přebírají. Mají stejné jméno, to je ve většině programovacích jazyků zakázáno. Java je rozpozná stejně jako my podle typů parametrů, příp. jejich počtu a použije ten správný. Názvy parametrů nemají žádný vliv, stejně jako typ návratové hodnoty. V javě jdou přetěžovat i konstruktory potom se na ně odkazuje metodou this. Například jeden konstruktor může volat druhý takto: this(parametry).

22. Rozhraní:

Java poskytuje na specializaci, kromě dědičnosti, další prostředek – rozhraní. Rozhraní definuje soubor metod, bez jejich implementace. Třída, která implementuje toto rozhraní (tj. jakoby jej zdědí), musí implementovat (tj. jakoby překrýt) všechny metody rozhraní, tedy je implementovat.

Deklarace rozhraní je podobná deklaraci třídy

```
interface jmeno {
// hlavicky metod
}
```

Každá třída, která implementuje toto rozhraní, což se zapíše pomocí klíčového slova *implements*, musí obsahovat deklarace všech metod rozhraní. Za předpokladu, že rozhraní je implementováno nějakou třídou, lze k instancím takovéto třídy přistupovat pomocí referenční proměnné typu rozhraní obdobně, jako lze přistupovat k instancím podtřídy pomocí referenčních proměnných nadtřídy.

V Javě na rozdíl od jiných programovacích jazyků (například C++) nemůže třída dědit od více tříd najednou (neexistuje vícenásobná dědičnost). Každá třída však může implementovat libovolný počet rozhraní, do jisté míry tedy rozhraní vícenásobnou dědičnost nahrazují.

23. Algoritmická řešitelnost problémů:

Máme-li problém formalizován ptejme se, jest-li existuje pro každý problém formalizovaný uvedeným způsobem algoritmus.

Pro problém řazení je odpovídající rozhodovací problém následující funkce

I	S
1	ano
2	ano
...	...
1,1	ano
1,2	ano
...	...
2,1	ne

2,2 ano

... ...

... ...

Známe-li řešení uvedeného rozhodovacího problému umíme řešit i odpovídající problém, například systematickým generováním všech permutací prvku zadané posloupnosti a rozhodováním zda jsou seřazeny. Použijeme-li na řešení problému čítač instance problému bude použitím odpovídajícího kódování vyjádřena jako řetězec 0 a 1, který můžeme jednoznačně interpretovat jako celé číslo (možná velmi veliké) a instanci problému říkáme vstup programu. Řešení, výstup programu, můžeme kódovat 1=ano, 0=ne. Nyní je rozhodovací problém formalizován jako funkce f: N → {0, 1}

Příklad:

Problém: Je zadané přirozené číslo liché?

I / n ∈N	1	2	3	4	5	6	...
S / f(n)	1	0	1	0	1	0	...

Předpokládejme, že najdeme problém, pro který neumíme napsat v Javě algoritmus, jinými slovy je neřešitelný JVM. Otázka je neexistuje-li jiný formalizmus pro zápis algoritmu, který by takový problém řešil.

Church – Turingova teze tvrdí, že každý algoritmus muže být vykonán Turingovým strojem. Navíc, každý program v konvenčních programovacích jazycích muže být transformován na Turingův stroj a naopak. Konvenční programovací jazyky a také Java jsou dostatečné pro vyjádření jakéhokoliv algoritmu.

Také další formalizace pojmu algoritmu vedly ke stejnému výsledku a obecně se předpokládá platnost Church – Turingovy teze. Church – Turingova teze však není větou a nemůže být dokázána. Může být vyvrácena, bude-li objevena metoda akceptovatelná jak algoritmus, který nebude možno vykonat Turingovým strojem.

Existuje-li tedy rozhodovací problém, pro jehož řešení neexistuje program, například v Javě, potom je tento problém algoritmicky neřešitelný a naopak. Uvažujme všechny rozhodovací programy v Javě. Jejich binární tvar můžeme interpretovat jako celá čísla a vyjmenovat je v jejich poradí, tedy P1, P2, ..., Pn, ... Každý z nich může mít vstup interpretovaný jako celé číslo 1, 2, ..., k, ... a jeho výstupem je Pn(k) ∈ {0, 1}. Vznikne tedy matice:

	1	2	...	k	...
P1	P1(1)	P1(2)	...	P1(k)	...
P2	P2(1)	P2(2)	...	P2(k)	...
...	...	...	...	...	...
Pn	Pn(1)	Pn(2)	...	Pn(k)	...
...	...	...	...	...	...

Má-li každý rozhodovací problém specifikovaný nějakou funkcí f: N → {0, 1} aspoň jeden program v Javě, jenž jej vypočítá, musí v uvedené matici existovat řádek Px takový, že Px(k) = f(k), k = 1, 2, …

Metodou diagonalizace můžeme ukázat, že existuje problém, tj. funkce f: N → {0, 1}, pro kterou v uvedené matici není žádný program. Problém nazveme D a je definován následovně:

D(k) = 1 je-li Pk(k) = 0

0 je-li Pk(k) = 1 k = 1, 2, ...

Pro libovolné pořadí řádků v matici můžeme definovat odpovídající problém D. Existují tedy rozhodovací problémy, pro které neexistují v Javě programy a tedy vzhledem k Church-Turingově tezi, pro něž neexistují žádné algoritmy. Takové problémy se nazývají algoritmicky nerozhodnutelné. První algoritmicky nerozhodnutelný problém ukázal v roce 1936 Alan Turing. Jde o problém zastavení (halting problem), který pomocí programu například v Javě, můžeme formulovat následovně:

Vstup: Program P v Javě reprezentován jako celé číslo a jeho vstup k reprezentován také jako celé číslo.

Výstup: 1 když se program P zastaví , 0 když se program P nezastaví

Ptáme se zda existuje v Javě program, který vypočítá (řeší) problém zastavení.

Opět uvažujme všechny programy v Javě, J1, J2, ..., Jn, ... a vytvořme matici, jejíž prvky jsou výstupy těchto programů pro jejich vstupy anebo ?, pokud pro daný vstup program nezastaví.

Například

	1	2	3	4	...
J1	5	4	?	8	...
J2	?	?	7	3	...
J3	4	2	6	?	...
J4	5	9	?	?	...
...	...	...	...	...	...
D	6	0	7	0	...

Nechť existuje metoda pro rozhodnutí zda se program Jn pro vstup k zastaví nebo ne. Potom metodou diagonalizace můžeme vytvořit program D, kterého výstup D(k) je 0 když Jk(k) nezastaví a Jk(k) + 1, když zastaví. Tento program však musí být jedním z programu J1, J2, ..., Jn, ..., což nemůže, protože jeho výstup D(k) je různý od Jk(k). Pro problém zastavení tedy neexistuje v Javě program, tedy ani žádný algoritmus a problém zastavení je algoritmicky nerozhodnutelný.

Uvedený výsledek lze také dosáhnout následujícím postupem. Nechť existuje metoda *zastavi()* s parametry program v Javě J a jeho vstup w, oba reprezentované celými čísly, která vrátí *true* jestliže program J pro vstup w se zastaví a *false*, jestliže se nezastaví.

```
boolean zastavi(int J, int w) {
   return boolean z = program J pro vstup w se zastavi;
}
```

Můžeme napsat metodu, která bude cyklovat, když metoda *zastavi()* vrátí t*rue*.

```
boolean cyklujKdyzZastavi(int J, int w) {
   if (zastavi(J, w))
       for ( ; ;);
   return false;
}
```

Nyní zvažme program, který volá metodu *cyklujKdyzZastavi* se sebou pro sebe, tj. její parametry jsou její celočíselné reprezentace.

```
class X {
...
   void x boolean() {
       return cyklujKdyzZastavi(cyklujKdyzZastavi,
                                                cyklujKdyzZastavi)
   }
   public static void main(String args[]) {
       x;
}
```

V tomto případe když metoda *zastavi()* vrátí *false* program zastaví (a vrátí *false*) a zůstane cyklovat, když *zastavi()* pro něj vrátí *true*, což je spor a tedy algoritmus pro problém zastavení nemůže existovat.

24. Klasifikace problémů:

První rozdělení všech problému jsme již učinili a to na takové, které jsou algoritmicky řešitelné a algoritmicky neřešitelné. Další klasifikace vychází z horního omezení času výpočtu pro velikost vstupu n. Většina našich algoritmů měla časovou náročnost $O(n^k)$ pro nějakou konstantu k, tj. Polynomiální. Na druhé straně jsou řešitelné problémy, o nichž víme, že nemají algoritmus s polynomiálním časem výpočtu. Příkladem je problém Hanojských věží, kde minimální počet přesunů disků je 2n – 1. Problémy, které jsou řešitelné v polynomiálním čase považujeme za zvládnutelné, nebo-li lehké a problémy, které vyžadují superpolynomiální čas považujeme za nezvládnutelné, nebo-li těžké. Podobně jako při zkoumání řešitelnosti problému se omezíme na rozhodovací problémy. Mnoho prakticky důležitých problému jsou optimalizační problémy, kdy výstupem je hodnota, která nejlépe splňuje zadaná kritéria.

Uvedli jsme již, že pro řešení abstraktního problému na počítači jeho instance musí kódovány do binárních řetězců. Když počítač řeší nějaký abstraktní problém, ve skutečnosti řeší jeho zakódovanou instanci. Problém v tomto tvaru nazýváme konkrétní. Konkrétní problém je řešitelný v polynomiálním čase, existuje-li algoritmus, který ho řeší v čase $O(n^k)$.

Třída složitosti P je množina konkrétních problému řešitelných v polynomiálním čase.

Pro některé rozhodovací problémy neumíme nalézt řešení v polynomiálním čase, ale na druhé straně pro ně existuje polynomiální verifikační algoritmus. Verifikační algoritmus ověří řešení problému na základe poskytnutých dat, která nazveme certifikát.

Už jsem se setkali s problémem splnitelnosti výrokových formulí, kde navrhnuté řešení pro n proměnných prověřovalo 2^n možností. Pokud nám někdo řekne, že daná výroková formule je splnitelná a nabídne nám hodnoty jednotlivých proměnných na důkaz svého tvrzení, můžeme dosazením těchto hodnot a vyhodnocením výrokové formule verifikovat zda tomu tak je, jistě v polynomiálním čase.

Problémy, jenž mohou být verifikovatelné v polynomiálním čase, tvoří **třídu složitosti NP** – nedeterministicky polynomiální.

Jaký je vztah třídy P a třídy NP? Je-li nějaký problém ve třídě P výsledek verifikace získáme jeho vyřešením, což odpovídá verifikaci bez certifikátu. Je tedy $P \subseteq NP$. .

Není známo jestli P = NP.