1
Problém, algoritmus, program

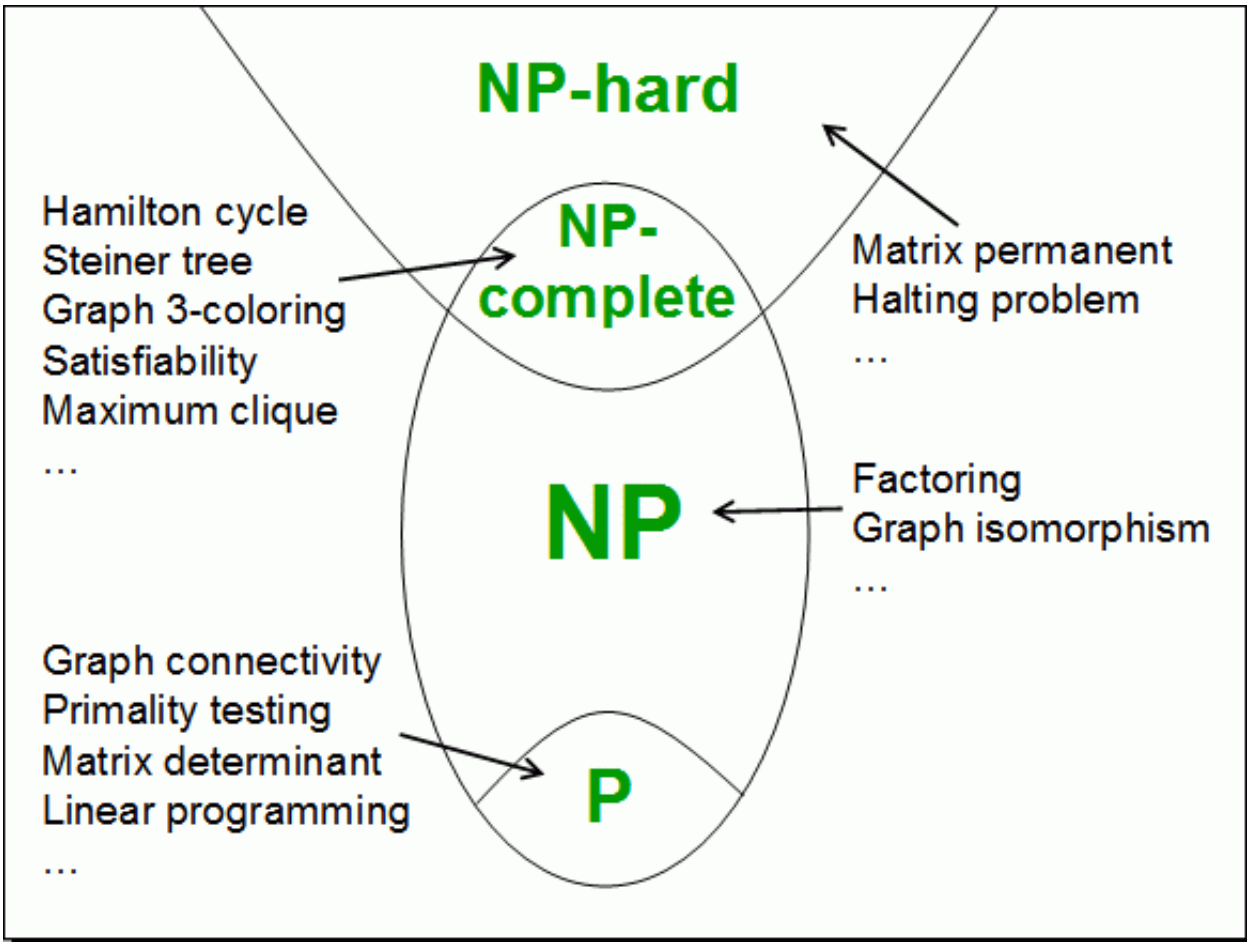
Problem

Problém: Otázka či stav, pro jejíž zodpovězení hledáme řešení, případně existenci řešení. Takovýto stav je nežádoucí a je tedy na nás jej změnit (problém vyřešit)

Postup: Zjistíme, že problém nastal (nelze situaci zvládnout známými postupy), zadefinujeme problém (vstupy a výstupy), nalezneme způsob žešení, zjistíme, zda je způsob řešení efektivní a případně optimalizujeme způsob žešení.

Rozhodovací problémy lze dělit do tříd složitosti

- P (polynomiální) - problém je řešitelný pomocí (deterministického) Turingova stroje (http://cs.wikipedia.org/wiki/Turing%C5%AFv_stroj) v polynomiálním čase (např problém řazení), lze dokázat, že všechny problémy P jsou podmnožinou problémů skupiny NP (NP problém s jednou větví, NP problém bez hádání či nápovědy)
- NP (nedeterministicky polynomiální) - Polynomiální čas, ale nedeterministický Turingův stroj (takový, který může rozvětvit program v libovolném kroku na více větví, ve kterých hledá řešení současně (zcela paralelně), případně lze mluvit o stroji, který "uhodne" kterou větví se vydat, případně mu něco "napoví" kudy se má vydat, aby řešení bylo správné. Také lze mluvit o problémech, jejichž řešení lze ověřit v polynomiálním čase (zpětný postup, zjištění, zda je nalezené řešení opdpovídající problému), níkoliv jej získat - pokud bychom všechny větve umístili řekněme do zásobníku (zásobníkový Touringův stroj), čas pro nalezení řešení by nutně nebyl polynomiální. Je otázkou, zda lze pro NP problémy najít P řešení.
- NP-těžké - Problémy, na které lze v polynomiálním čase převést všechny problémy NP, nemusí však nutně v NP samy o sobě být (nemusí být ani rozhodovací)
- NP-úplné - NP-těžké problémy, které jsou ve třídě NP, jde tedy o nejtěžší NP úlohy



Vztah mezi P a NP je jedním ze sedmi problémů tisíciletí, které vypsal Clayův matematický ústav 24. května 2000, za rozhodnutí vztahu nabízí 1 000 000 dolarů.

Pro zasmání: http://www.abclinuxu.cz/clanky/komiks-xkcd-287-np-uplnost

Algoritmus

Algoritmus: Obecný postup pro nalezení řešení, ověřený že funguje. Jde o konečnou sekvenci známých operací (některé definice říkají, že tyto operace musí být elementýrní) které vedou k vyřešení problému.

- Každý bod algoritmu musí být jednoznačně určen a musí být proveditelný (pochopitelný strojem)
- Algoritmus musí mít konečný počet kroků, tedy musí skončit po konečném počtu operací (narozdíl od výpočetní metody)
- Algoritmus má 0 - N vstupů
- Algoritmus má 1 - N výstupů

Program

Výpočetní metoda: "Algoritmus", který není konečný (event-driven systémy jako třeba automat na kávu, snímače JIS karet, digitální hodinky...)

Program: Výpočetní metoda (interaktivní aplikace) či algoritmus (dávkové aplikace) zapsaný v souladu s pravidly programovacího jazyka tak, aby bylo možné algoritmus přenést a použít na počítači či jiném stroji. Programovací jazyk je nástroj programátora pro snazší programování, převede se na strojový jazyk, kterému počítač rozumí. Součástí programu může být i popis použitých datových struktur, se kterými algoritmus či výpočetní metoda pracuje.

2
Vykonání programu

Vykonávání programu je hlavní činnost počítače. Hlavními částmi, které se podílí na výpočtu, jsou procesor a hlavní paměť. V hlavní paměti je uložen program ve strojovém kódu jako posloupnost instrukcí a dat (von Neumannova koncepce). V procesoru (CPU) se vykonávají instrukce, sekvenčně, pokud není pořadí změněno instrukcemi skoku. Programy lze tedy větvit na základě podmínek a pořadí vykonávání instrukcí je závislé na jejich splnění.

Drtivá většina programů napsána ve vyšších jazycích, je proto nutné je upravit tak, aby jim počítač rozuměl, tedy jednotlivé příkazy převést na instrukce procesoru. K tomu existují 2 postupy:

Kompilace: Program je přeložen najednou do strojové podoby, která je platformově závislá. Operační systém tento program nahraje do operační paměti a spustí. O překlad se stará překladač nebo kompilátor, který přeloží příkazy vstupního souboru na instrukce. Jelikož je v konvenčních pratformách použito jednoho adresního prostoru pro jednu aplikaci, některé prog. jazyky také vyžadují také linker, který převádí adresy jednotlivých funkcí a instrukcí (obsažených v jednotlivých modulech programu) z abstraktní podoby na konečnou absolutní. Kompilovaný program je platformově závislý a vyžaduje specifický překladač pro danou platformu, jeho běh je však nativní a zpravidla rychlý, a je možno využít knihoven dané platformy (například API uživatelského rozhraní WinForms). Příkladem je C, C++, Pascal

Interpretace: Program je interpretován přímo ze zdrojového kódu, příkazy jsou z kódu přečteny, interpretovány ("tlumočeny") a okamžitě provedeny. Lze interpretofat příkazy jednotlivě, nebo interpretovat celý kód najednou při spuštění. Tento přístup vyžaduje interpreter, který je shopen pro danou platformu příkazy tlumočit. Příklady: PHP, BASIC, Perl, Python

Hybridní vykonávání: Některé jazyky využívají kombinace předchozího přístupu, tedy kompilaci zdrojového kódu do univerzálního platormově nezávislého jazyka, který je poté interpretován. Příkladem je Java (překlad do byte-code (class) souborů, které jsou spouštěny na virtuálním stroji) nebo C# (překlad do nezávislého Common Intermediate Language, který je poté přeložen do nativní podoby a spuštěn, je li potřeba (JITerem))

3
Objekt, třída

Třída

Třída: Elementární stavební prvek objektově orientovaného programování. Obsahuje předpis, podle kterého lze vytvořit instanci třídy - objekt. Obsahuje členské atributy a metody.

Zapouzdření: Třída zapouzduje atributy a funkce jako "její". Na venek komunikuje pomocí rozhraní, nebo odhaluje některé atributy či metody pomocí přístupových práv, většinou

- Public - veřejné, k třídě či jejím public atributům a metodám lze přistupovat odkudkoliv
- Protected - třídě či jejím protected atributům a metodám lze přistupovat pouze z ní (this v případě objektu, statický přístup) nebo z oděděných tříd
- Private - třídě či jejím private atributům a metodám lze přistupovat pouze z ní

Některé jazyky podporují i další práva, např Internal v jazyce C#, který dovoluje přistupovat k třídě z její assembly (aplikace, ddl knihovna,...)

Typy tříd:

- "Klasická" třída - dovoluje instancování na objekty, každá instance dostane přidělenu vlastní paměť
- Statická třída a členové - klíčové slovo static - Třída, která není instancovatelná, a její metody jsou přístupné pouze statickým způsobem, tedy Třída.Metoda() (nikoliv instanceTřídy.Metoda()). Klasická třída může obsahovat statické členy, které jsou pro včechny případné instance společné (například static parametr počet instancí, který se v konstruktoru zvýší). V jazycích, které statické třídy nepodporují se používá návrhový vzor singleton.
- Abstraktní třída - abstract - obsahuje abstraktní metody, není instancovatelná. Je nutné od ní oddědit a abstraktní metody doimplementovat (např abstraktní třída Čtečka s metodou Čti(), od které dědí FlashČtečka, DiskČtečka, CDROMČtečka....(To není zrovna dobrý příklad, jelikož čtení provádí OS, ale jako idea je to OK))
- Koncová třída (final v Javě, sealed v C#, v C++ není) - třída, kterou nelze dědit

Objekt

Objekt: Instance třídy, obraz třídy v paměti, referenční datový typ. Přebírá atributy a metody své třídy. Je nutné jej vytvořit pomocí operátoru new, případně uvolnit pomocí většinou delete (pokud není garbage collector).

Třída vs Struktura: Třída a struktura (struct, record,...) mohou mít společné vlastnosti. Rozdíly se projevují podle jazyka, většinou je však platný ten rozdíl, že členské prvky struktury jsou public zatímco členské prvky třídy jsou private. V C# je další rozdíl v uložení (třída v paměti, struktura na zásobníku). Java struktury nemá, používá se final class

Konstruktor a destruktor: Metoda, která inicializuje, respektive uvolňuje třídu do/z paměti. Většinou nastavuje atributy třídy na výchozí hodnoty.

Gettery a Settery: Veřejné metody, které nastavují privátní atributy objektu či třídy. Používáme je, abychom zamezili entitám využívající atributy třídy nastavit tyto atributy na nesmyslné hodnoty.

4
Spojové datové struktury

Pokud potřebujeme ukládat data do paměti a nevíme předem, kolik jich bude (kolik prvků bude nutné uložit), nebo mění-li se v runtimu jejich počet, používáme spojové datové struktury. Každý prvek - záznam - struktury je reprezentován např. jako objekt, který ukazuje na svého následovníka (ukazatel, pointer), případně na další objekty (předchozí, začátek...). Ukazatel je vlastně adresa objektu, takže pomocí něho můžeme k jiným objektům přistupovat.

Vyhledávání prvků struktury je většinou iterační, posunujeme se z jednoho objektu na druhou (na rozdíl od pole, kde přistupujeme přímo, indexem). Díky tomu je vyhledávání ve spojových datových strukturách pomalejší než v polích (kde je O(1)), ale zato není potřeba předem alokovat velké množství paměti, kterou si spojová datová struktura bere až za běhu programu (na rozdíl od statickéého pole, kde se alokuje při spuštění).

Jednotlivé záznamy mohou obsahovat metody pro operace se strukturou (zápis, vložení, čtení, posun po prvcích) a, pokud vyžadujeme dědičnost, je možné implementovat spojové struktuře rozhraní, které potřebné operace implementuje.

Příkladem jsou:

- fronty
- zásobníky
- seznamy (jednosměrné, obousměrné)
- stromy
- skip-list
- ...

5
Správnost programů

Testováním programu, tj. ověřením, že pro daný vstup získáme správný výsledek, nemůžeme obecně prokázat správnost programu. Množství vstupů je většinou tak veliké, že testování není možné ani za použití velice rychlých počítačů. Na druhé straně testování může alespoň prokázat chybu v programu. Protože empirickým testováním nemůžeme zaručit správnost programu, musíme ho nahradit analytickým přístupem pro porozumění správnosti programu. Sekvence příkazů (přiřazení, alternativy a cyklu) umožňuje dokázání správnosti programu tak, že můžeme napsat tvrzení o hodnotách proměnných před vykonáním příkazu, které nazýváme předpoklad a tvrzení po jeho vykonání, které nazýváme důsledek.

Příkazy sekvenčně

Například pokud máme dva příkazy sekvenčně za sebou, tak vyjdeme z požadovaného důsledku druhého příkazu. Na základě důsledku stanovíme požadovaný předpoklad pro druhý příkaz, což je také požadovaný důsledek pro první příkaz atd. až získáme hodnoty možných vstupů.

Příkazy alternativy

Příkazy alternativy zachovávají sekvenční vykonávání příkazu.

Příkazy cyklu

Pro pochopení příkazů opakování zavedeme pojem invariant cyklu. Pro invariant cyklu musíme obecně ukázat tři vlastnosti:

- **Inicializace** - je splněn před vykonáním prvního cyklu.
- **Udržování** - je-li splněn pře vykonáním cyklu, zůstává splněn i po něm.
- **Skončení** - když příkaz cyklu skončí, invariant nám ukáže správnost algoritmu (dokážeme-li, že inicializace platí, a cyklus je udržován pro všechny iterace).

Přiklad na řazeni pole vkládáním:

- **Inicializace**: Po začátečním přiřazení i = 1 je pole s prvky s indexy 0, ..., i-1 osahující jeden prvek triviálně seřazeno.
- **Udržování**: Cyklus while posouvá již seřazené prvky a[i-1], a[i-2], ... o jednu pozici doprava dokud se nenajde správné místo pro a[i]. Výsledkem je, že prvky pole a[0] ... a[i] jsou seřazeny. Invariant je tedy splněn i pro následující opakování, kdy i je v hlavičce cyklu inkrementováno.
- **Skončení**: Cyklus for skončí když i=n, kde n je počet prvků pole. Dosazením do textu invariantu cyklu dostáváme, že prvky s indexy 0 ... n-1 obsahují seřazené původní prvky pole, co je to co jsme požadovali.

6
Analýza programů

Analýza programu se zabývá nároky programu na zdroje, tedy čas, paměť, šířka pásma... Máme-li problém, lze pro jeho řešení zpravidla použít více algoritmů, a každý algoritmus může mít několik implementací. Analýza programů se zabývá právě vztahem implementace řešení původního problému vzhledem k požadovanému zdroji.

Obsah

- 1 Časová náročnost
- 2 Asymptotická složitost
- 3 Očekávaná složitost
- 4 Klasifikace složitosí

Časová náročnost

Nejčastěji nás zajímá časová náročnost programu. Analyzujeme, jakou dobu bude implementovanému algoritmu trvat výpočet. Tuto dobu ovlivňuje způsob implementace (počet nutně provedených operací) a velikost vstupních dat (počet prvků, které algoritmus zpracovává, případně jejich uspořádání). Definujeme elementární čas pro všechny operace $c_{operace}$ a jejich skádáním počítáme celkový čas potřebný pro provedení algoritmu.

Příklad: Naplnění již alokovaného pole náhodnými čísly (c#)

```
for (int i = 0, i < pole.Length; i++)
{
  pole[i] = rand.Next();
}
```

- máme pole délky n = pole.Length:
- alokace proměnné trvá c_{aloc}
- operace přiřazení trvá $c_{přiřazení}$
- operace porovnání trvá $c_{porovnání}$
- operaci inkrementace lze definovat jako $c_{inkrement} = c_{přiřazení} + c_{součet}$
- řekněme, že generování náhodného čísla trvá konstantní čas c_{rand}

- Inicializace cyklu for: alokace + přiřazení = $c_{aloc} + c_{přiřazení}$
- Běh cyklu: porovnání (i < length) + přiřazení (do pole) + náh. číslo + inkrementace (i++) = $c_{porovnání} + c_{přiřazení} + c_{rand} + c_{inkrement}$

- Cyklus for proběkne n-krát, tedy: n * ($c_{porovnání} + c_{přiřazení} + c_{rand} + c_{inkrement}$)
- Celkem: $c_{aloc} + c_{přiřazení} + n * (c_{porovnání} + c_{přiřazení} + c_{rand} + c_{inkrement})$

Asymptotická složitost

Ve většině případů neznáme čas potřebný pro konstantní operace, ale víme, že je velmi malý. Dobu výpočtu předchozího příkladu můžeme tedy přepsat na **a** + **n** * **b**, kde **a** a **b** jsou doby, které mají určitou délku, která nás však přísliš nezajímá.

Více než doba výpočtu nás tedy zajímá počet operací, které je třeba vykonat (stejně budou operace na každém systému vykonávány jinou dobu (PC vs. mobilní telefon např.)). Složitostí programu myslíme právě počet operací potřebný pro výpočet vzhledem k počtu prvků na vstupu algoritmu, omezený shora a/nebo zdola. Tato složitost se pro libovolný počet prvků asymptoticky blíží k určité funkci. Nás zajímá, ke které. Rozlišujeme také složitost algoritmu a složitost problému (ta by měla patřit do otázky PPA1).

Theta notace Θ(f(x)) vyjadřuje A.S., omezenou složitostní funkcí $c_{1,2} * f(x)$ shora, resp zdola. Hledáme tedy 2 konstatnty, pro které je analyzovaná složitost omezena.

Tato notace říká, že algoritmus nebude asymptoticky složitější než $c_1 * f(x)$, a nebude rychlejší než $c_2 * f(x)$. To znamená, že problém lze řešit algoritmem, který nebude asymptoticky složitější než $c_1 * f(x)$, ale zároveň nikdy nebude lepší než $c_2 * f(x)$.

Omikron notace O(f(x)) vyjadřuje A.S. omezenou funkcí c * f(x) pouze shora. Jinými slovy, jde o maximální možnou složitost algoritmu. Jde o první podmínku theta notace.

Omega notace Ω(f(x)) vyjadřuje A.S. omezenou funkcí c * f(x) pouze zdola. Říká, že algoritmus pro alespoň jeden vstup bude této složitosti. Druhá podmínka theta notace.

Pokud platí omikron a omega notace pro stejnou f(x), mluvíme o theta notaci.

Konstanty v notaci zanedbáváme, jelikož vzhledem k většímu n nehrají roli. Sčítáním složitostí tedy nedochází k jejich zhoršení, jsou-li stejného řádu (např $n^2 + n^2 = 2n^2 => n^2$, 200 operací je asymptoticky stejné jako 100, ale $n^2 + n^3 => n^3$, 100 * 100 operací je vzhledem k 100 * 100 * 100 zanedbatelné).

Předchozí příklad je tedy složitosti Θ(n) , jelikož je vždy lineární vzhledem k délce pole (netrvá kratší, ani delsí dobu/počet operací)

Očekávaná složitost

Většina algoritmů je asymptoticky omezena pro extrémní případny na vstupu (např. Quicksort je Ω(n) (pole již seřazeno) a O(n2) (pole je seřazeno opačně)). Chování algoritmu má však očekávanou složitost, která nastává pro většinu případů (jak víme O(n logn) pro QuickSort). Využívá se také Theta/Omikron/Omega notací.

Je-li algoritmus závislý na vstupních datech (resp. na jejich uspořádání), a potřebujeme, aby složitost byla pokud možno vždy očekávaná, lze využít náhodného přeuspořádání dat na vstupu (za předpokladu, že je to možné - např. pro řazení ano, pro výpis znaků na obrazovku evidentně ne). Toto přeuspořádání může být konstatní

složitosti (např 100x přeházíme vzájemně prvky pole s náhodnými indexy pokud je pole rozumné odpovídající délky), případně lineární, složitost algoritmu tedy neovlivňuje (za předpokladu, že algoritmus samotný není očekávané sublineární časové složitosti).

Klasifikace složitosí

Polynomiální - složitost O(n * c) kde c je konstanta. Polynomiální algoritmy jsou použitelné. Ideální jsou algoritmy rychlejší (lineární, logaritmické,...), ne vždy je však možné takový algoritmus vymyslet.

Exponenciální - složitost $O(c^n)$. Tyto algoritmy nejsou příliš dobré, a je vhodné se zamyslet a pokusit se vymyslet jiný, polynomiálně složitý algoritmus. Zpravidla algoritmy hrubé síly jsou této složitosti, jelikož procházejí všechny kombinace vstupu a hledají řešení. Pro malé n jsou však použitelné díky menší režii a přípravě, paměťovým nárokům atd.

Postupy výše lze aplikovat i na složitosti paměťové, šířky pásma, ...

7
Rekurze

Rekurzí rozumíme sebevolání, tedy kód, který volá sám sebe či svojí část (přímá rekurze), případně volá kód, který posléze volá kód volající (nepřímá rekurze).

Rekurze je vhodná pro řešení rekurzivních problémů (například výpočet faktoriálu nebo Fibonacciho posloupnosti, fraktály), je však paměťově velmi náročný (každé rekurzivní volání vyžaduje vlastní paměťový prostor), doporučuje se tedy pouze pro vhodné problémy (rekurzivní výpis řetězce na obrazovku je nesmysl).

Rekurze musí mít následující podmínky:

- musí být konečná, tedy musí existovat vstup, pro který rekurzivní funkce nevolá sama sebe, tedy musí být testována tato konečnost před voláním sebe sama
- problém se musí rekurzí zjednodušovat, aby rekurze ke koncové podmínce dospěla (případně můžeme definovat maximální počet vnoření jako pojistku)

Typy rekurze

- **Přímá** - metoda volá sama sebe
- **Nepřímá** - metoda volá metodu, která volá zpětně ji

- **Lineární** - metoda volá pouze jednu svoji kopii (faktorial)
- **Stromová** - metoda volá více svých kopií (Fibonacci, průchod adresářovou strukturou, často Divide & Conquer algoritmy)

Rekurze však často umožňuje zjednodušení kódu a ušetření času při programování. Často lze rekurzivní problémy řešit také čistě iteračním algoritmem, nebo použitím zásobníku.

Příklad: Procházení adresářové struktury. Vytvoříme metodu ProjdiAdresář(string cesta);

Iteračně (nesmysl?):

1. Vytvoříme ukazatel na položku adresářové struktury ukazující "před?" výchozí adresář a uložíme tuto hodnotu do vedlejší proměnné
2. Posuneme ukazatel na další položku adresáře
3. Je-li pod ukazatelem označená položka, nebo prázdno, vrátíme se na předchozí položku a zpět na 2, nebo ukončíme, pokud jsme na uložené položce "před?"
4. Je-li pod ukazatelem adresář, označíme jej jako zpracovaný (i když je otázka jak) a posuneme ukazatel na jeho první položku
5. Je-li pod ukazatelem soubor, [otevře, vypíše název, zpracuje, přidá do vlastní datové struktury,...], a označíme soubor jako zpracovaný
6. Zpět na bod 2

Zásobníkem:

1. Načte obsah adresáře 'cesta' do zásobníku
2. Prochází položky zásobníku:
    1. Je - li položka soubor, [otevře, vypíše název, zpracuje, přidá do vlastní datové struktury,...]
    2. Je - li položka adresář, načte jeho obsah do zásobníku
3. Opakuje body 2 a 3 dokud není zásobník prázdný

Rekurzivně:

1. Načíst obsah adresáře 'cesta' a prochází jednotlivé položky
2. Narazí - li na soubor, [otevře, vypíše název, zpracuje, přidá do vlastní datové struktury,...]
3. Narazí - li na adresář, volá sama sebe na jeho cestu

Vidíme, že pro rekurzi neimplementujeme vlastní zásobník (viz však níže), a kód je přehlednější (Rekurze cca 5 řádek, iteračně mnohem více)

Pár slov k implementaci rekurze: V imperativním programování je většinou rekurze vnitřně řešena jak jinak než na zásobníku, jelikož je stav volající funkce uložen na vnitřní zásobník. Technicky vzato je tedy rekurze nahrazena zásobníkovým způsobem. Pri chybě rekurze (nesplnění koncové podmínky) většina jazyků vyhodí Stack Overflow (přetečení zásobníku) případně Access violation (Chyba přístupu do chráněné paměti), záleží na způsobu vnitřní implementace zásobníku.

// http://www.abclinuxu.cz/clanky/ruzne/komiks-xkcd-244-stolni-rpg pro ty, to ví co je Dungeons & Dragons

Příklady

Výpočet faktoriálu:

```
int faktorial(int f) {
  if (f <= 1)
    return 1;
  else
    return f * faktorial(f - 1);
}
```

Fibonacciho posloupnost:

```
int fibonacci(int n) {
  if (n <= 2)
   return 1;
 else
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

8
Abstraktní datové typy

ADT je matematický model dat a operace nad těmito daty. Využívá základních datových typů pro konstrukci komplexnějších ADT, které využíváme pro snadnější organizaci dat a často pro optimalizaci algoritmů.

Abstraktní datový typ

- je zcela oddělený blok, s klientským programem komunikuje pomocí rozhraní
- je nezávislý na implementaci, na venek se projevuje jen veřejnými metodami
- vše ostatní, kromě ovládacích metod, je uživateli skryto (samotná data a přístup k nim např.) (integrita)
- je (klientsky) nezávislým blokem, může jej využívat kdokoliv, kdo zná rozhraní (pokud je ADT tedy třeba v DLL knihovně, můžeme jej používat i z jiných jazyků) (modularita)
- je pevně a jasně definován

ADT definuje a podporuje tyto operace

- **kostruktor** - vytvoří istanci ADT
- **selektor** - vyhledá žádaný prvek (indexer, At(), Pop()...)
- **modifikátor** - upraví data ve struktuře (indexer, Push(), Add()...)

Těmito operacemi jsou data jako taková uživateli skryta. Dynamická množina - množina dat, které předchozí operace podporuje, základ většiny ADT

Příklady ADT:

- Seznam
- Fronta
- Zásobník
- Trie
- Binární vyhledávací strom
- Hashmapa
- ...

ADT je často využíváno návrhovými vzory. V některých jazycích lze ADT vytvořit jako generické, tedy vhodné pro uchování libovolného datového typu (např List<int>). Tyto generické ADT jsou také často v jazyce již implementovány jako knihovny (C# v Generics, C++ v Standard Template Library). U jazyků, které toto nepodporují lze využít Objekt, a ten přetypovávat, což není ideální, ale funguje. U jazyků nemajících objekt je to horší, musíme přijít s vlastní specifickou implementací.

9
Zásobník, fronta, seznam

Všechny tyto ADT implementují IsEmpty (prázdná struktura) a Size (pošet prvků ve struktuře)

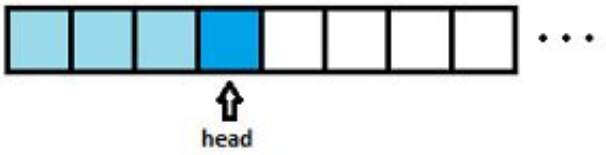
Zásobník

Interface: operace Push, Pop, Peek

- **Push** vkládá data do zásobníku za sebe
- **Pop** vybírá nejpozději vložená data
- **Peek** vrátí nejpozději vložená data bez výběru prvku ze struktury

= LIFO pamět (Last In First Out)

Všechny operace jsou O(1)



Head: vrchol zásobníku

Implementace objektově:

- Prvky (objekty) obsahují data a ukazují na nižší prvky v zásobníku (na prvky pod sebou)
- **Konstruktor:** Vytvoříme ukazatel Head, který nastavíme na null
- **Modifikátor:** Push vytvoří pro data nový prvek, který ukazuje na prvek Head (Head bude předchozí), a změní ukazatel Head na vytvořený prvek
- **Selektor:** Peek vrátí data prvku Head, pokud Head != null. Pop vrátí stejná data jako Peek, ale před návratem změní ukazatel Head na předchozí prvek (Head = Head.Previous)

Implementace na poli:

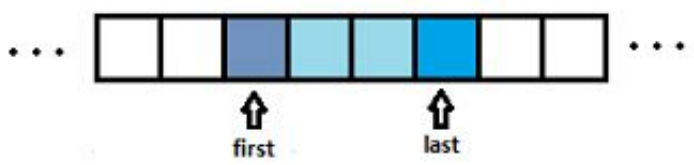
- Máme pole typu stejného jako data
- **Konstruktor:** Alokace pole a indexu Head, nastavení Head na -1
- **Modifikátor:** Push zvýší index Head o 1 a zapíše data do pole na tento index. Pokud je index mimo pole, zvětšení pole (např. 2x), a po té zápis
- **Selektor:** Peek vrátí prvek pole pod indexem Head, pokud Head >= 0. Pop jako peek, ale před návratem provede Head--, pokud Head >= 0.

Fronta

Interface: operace Push, Pop, Front

- **Push** vkládá data do fronty za sebe
- **Pop** vybírá nejdříve vložená data
- **Peek** vrátí nejdříve vložená data bez výběru prvku ze struktury

= FIFO pamět (First In First Out)



First: začátek fronty; Last: konec fronty

Implementace objektově:

- Prvky obsahují data a ukazují na následující prvky fronty
- **Konstruktor:** Vytvoříme pouze ukazatele First a Last, ukazující na null.
- **Modifikátor:** Push vytvoří pro data nový prvek, upraví prvek Last (pokud != null) tak, aby ukazoval na tento prvek, a změní ukazatel Last na tento prvek. Je-li ukazatel First nastaven na null, nastaví tento také na vytvořený prvek.
- **Selektor:** Front vrátí data prvku First, pokud First != null. Pop vrátí stejná data jako Front, ale před návratem změní ukazatel First na následující prvek, pokud First != null (First = First.Next)

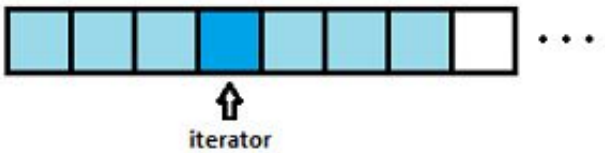
Implementace na poli (cyklicky): Máme pole typu stejného jako data a indexy First a Last. Indexy neukazují přímo do pole, nýbrž je proveden přepočet (Index mod Delka pole) abychom využili celé pole. Tato fronta má omezenou délku.

- **Konstruktor:** Alokace pole a indexů, nastavení Last na -1 a First na -1
- **Modifikátor:** Pokud ((Last + 1) mod Delka) != (First mod Delka), Push provede Last++ a zapíše data do pole na tento index, jinak chyba (plná fronta). Pokud First = -1, pak First = 0 (první prvek).
- **Selektor:** Front vrátí prvek pole pod indexem First, pokud First <= Last. Pop jako Front, ale před návratem provede First++.

Speciálním případem je například prioritní fronta, která řadí prvky ve frontě podle priority

Seznam

Interface - Iterator nad seznamem: Add, Remove, Get; First, Next, IsLast, případně další (Previous, Last, IndexOf, ...)



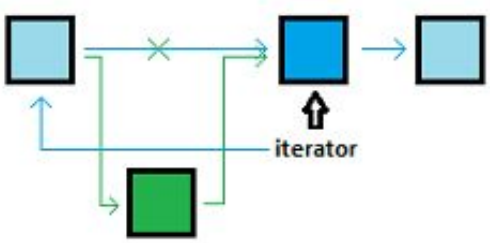
- **Add** vkládá data na pozici iterátoru. Složitost dle implementace.
- **Remove** maže data na pozici iterátoru

- **Get** vrátí data na pozici iteratoru
- **First** vrátí iterátor na začátek seznamu
- **Next** posune iterátor dále
- **IsLast** indikuje koncová prvek
- (Previous v obousměrných seznamech posune iterator na předchozí prvek)
- (Last posune iterator na poslední prvek)
- (IndexOf vyhledá prvek ve struktuře)

= struktura, do které je možné libovolně zapisovat a číst (na libovolné pozici)

Iterator: ukazatel do struktury

Implementace objektově - nástřel:



Prvky ukazují na následující prvky seznamu. Iterátor je dobré implementovat tak, že ve skutečnosti ukazuje na předchozí prvek (Pokud tedy seznam není obousměrný, pak máme metodu Previous a je to jedno). Metoda add vloží prvek za prvek, na který ukazuje iterátor tak, že pouze upraví ukazatele prvků:

Remove je opačný proces, Get vrátí data (Tedy Iterator.Next.Data), První prvek seznamu si uložíme, abychom se mohli rychle vracet, stejně jako poslední prvek. Next provede Iterator.Next = Iterator.Next.Next. Kontrolujeme, zda nejsme mimo seznam.

- Obousměrný seznam: Prvky mají i ukazatele na předchozí prvky, a je tedy možné se po nich pohybovat na obě strany.
- Kruhový seznam: Poslední prvek neukazuje na null, ale na první prvek. Tyto seznamy jsou použitelné ve speciálních případech (například nastavení segmentů sedmisegmentovky pro digitální hodiny)

Implementace na poli (nástřel): Pole má nevýhodu, že je nutné přesunout prvky doprava od iterétoru, pokud vkládáme prvky. Při mazání lze prvky posunout doleva, nebo do pole uložit hodnotu, která se zcela jistě nebude v poli nebude vyskytovat, nebo vytvořit stínové pole, ve kterém si smazané prvky označíme, a při posunu iterátoru je poté vynecháme. Tento přístup je dobrý pokud pole často upravujeme.

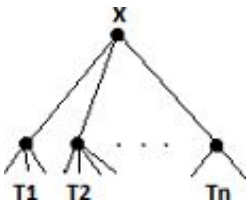
10
Strom, průchody stromem, binární vyhledávací stromy

Strom je ADT - matematicky jde o typ grafu, který neobsahuje cykly.

Definice

- **Vrchol (uzel)** - jeden prvek stromu. Má jednoho předchůdce a 0 - N následovníků.
- **Hrana (větev)** - spojení dvou vrcholů
- **List** - nrchol bez následovníků
- **Kořen** - vrchol, který nemá žádné předchůdce. Z kořene je pak možné se dostat do jakéhokoliv uzlu stromu. Každý strom má jen jeden kořen.
- **Podstrom** - // dopsat
- **Cesta stromem** - cesta od libovolného vrcholu ke kořeni stromu. Tato cesta je ve stromu unikátní.
- **Hloubka vrcholu** - délka cesty od kořene do daného vrcholu.
- **Výška stromu** - nejdelší možná cesta ve stromu.

// přidat obrázek s popisem částí stromu



Implementace: Objektově - každá prvek má ukazatele Parent a např. pole ukazatelů Childern, výodné, jednoduché Polem - složité pro obecné stromy, je nutné ukládat indexy kořenů podstromů, počty podstromů těchto kořenů a data vrcholů jako taková. Speciálně lze implementovat stromy s konstatntním počtem podstromů v každém vrcholu (BVS, Octree, ...), kde je implementace jednodušší

Adresářová struktura, sportovní výsledky, reprezentace stromů v přírodě :), ...

Průchod stromem

Do šířky - Projdeme vrcholy po patrech, tedy kořen, postupně kořeny jeho podstromů, kořeny všech těchto podstromů atd. Jelikož ve stromech obecně neznáme sousedy, musíme se často vracet. Měřením cesty vrcholů však můžeme prohledávání do šířky převést na prohledávání do hloubky, i když ne zrovna elegantní (vyhledáme všechny vrcholy o hloubkách 0, pak 1, pak 2, ...)

Do hloubky - Procházíme vrcholy do hloubky, tedy dojdeme-li do vrcholu, projdeme jeho podstromy a poté se teprve vracíme

Procházení stromu může být průchod do hloubky dvojího druhu, podle priority operací:

- **Preorder** - Nejprve vyzvedneme data vrcholu a poté projdeme postupně všechny podstromy

```
preorder(node) {
  print node.value
  if node.left ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
}
```

- **Postorder** - Nejprve projdeme všechny podstromy, a poté vyzvedneme data vrcholu

```
postorder(node) {
  if node.left ≠ null then postorder(node.left)
  if node.right ≠ null then postorder(node.right)
  print node.value
}
```

Procházení binárního stromu má navíc ještě jeden způsob:

- **Inorder** - Projdeme levý podstrom, poté vyzvedneme data vrcholu, a poté pravý podstrom. Při výpisu BVS touto metodou dojdeme k seřazenému poli.

Binární vyhledávací strom (BVS)

BVS je binární strom (strom, který má maximálně 2 podstromy v kažném vrholu) vytvořený tak, že kořen levého podstromu každého vrcholu má nižší hodnotu než tento vrchol a ten má nižší hodnotu než kořen pravého podstromu.

Pri přidáváni vrcholu začínáme u kořene, a rozhodujeme se, kterou hranou jít dále, tedy je li vkládaný prvek menší , nebo větší než tento kořen. Takto postupujeme i vybraným podstromem až najdeme místo, kam vrchol umístit. Umístěný vrchol je vždy listem stromu.

Mazání je složitější. Nejprve vrchol najdeme ve stromu a podle toho jaký je:

- Pokud je listem, je to triviální
- Pokud vrchol má pouze jeden podstrom, vrchol smažeme a na jeho místo umístíme kořen jeho (jediného) podstromu
- Pokud má dva, vydáme se doprava a poté jdeme stále doleva dokud nenarazíme na list, který smažeme, ale jeho hodnotu přiřadíme původně mazanému vrcholu.

Tento strom je nevyvážený, tedy může nastat situace, kdy bude jeden podstrom kořenu mít výrazně větší výšku než podstrom druhý. První vložený prvek bude vždy kořen stromu. Při načítání prvků do stromu je tedy vhodné alespoň pro tento prvek nalézt medián.

Přidání prvku do pole a mazání je maximálně O(n) (pro strom, který byl vytvořen ze seřazené posloupnosti - je to tedy pouze lineární spojový seznam), očekávaná složitost operací je však O(log2 n) jelikož je strom v každém vrcholu rozdělen na 2 skupiny hodnot.

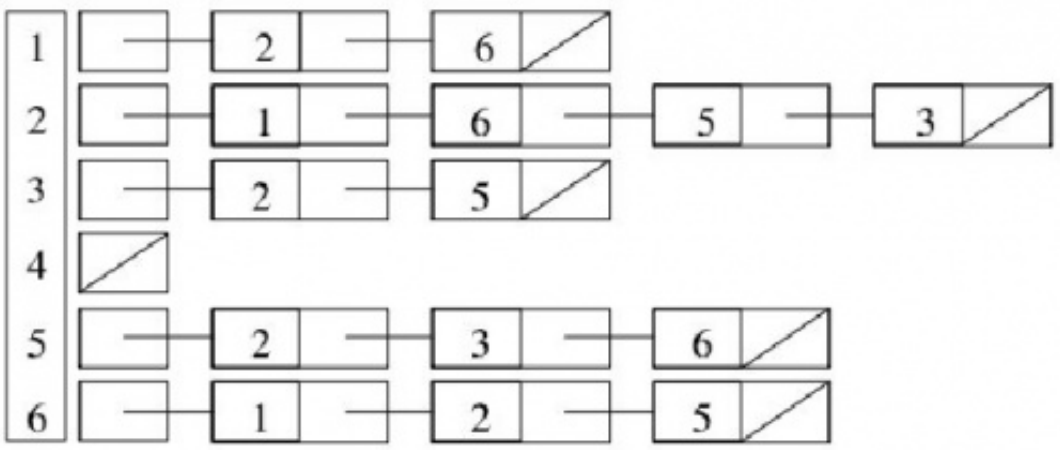
11
Grafy a jejich implementace

Graf je reprezentován množinou vrcholů (uzlů) V, které mohou být spojeny určitým počtem hran H. Graf je poté definován jako G(V,H)

- **Neorientovaný graf**: Hrany grafu nejsou orientované, tedy pokud vede hrana z uzlu 1 do uzlu 5, vede i z uzlu 5 do uzlu 1 a platí tedy (u1, u2) = (u2, u1)
- **Orientovaný graf**: Hrany grafu jsou orientované (jednosměrné), tedy pokud vede hrana z uzlu 1 do uzlu 5, nevede automaticky zpět z uzlu 5 do uzlu 1 a platí tedy (u1, u2) != (u2, u1)

- **Ohodnocený graf**: Každá hrana má přižezenou svoji hodnotu (číslo). Tyto hodnoty pak udávají například délku hrany, propustnost nebo třeba cenu za přechod po dané hraně.
- **Neohodnocený graf**: Všechhny hrany grafu jsou si rovny (v podstatě jsou všechny ohodnoceny stejným číslem), takže nemá smysl uchocácat informaci o jejich ohodnocení.

Implementace grafu pomocí spojového seznamu (seznam sousednosti)



Máme seznam vrcholů a pro každý z nich máme seznam vrcholů, se kterými sousedí. Lze tak implementovat všechny 3 typy grafů (v obousměrných budou sousednosti v seznamech obou vrcholů sdílejících hranu, ohodnocení vrcholů je uloženo v seznamu vrcholů, ohodnocení hran je v seznamu sousednosti)

Problém metody je v složitosti hledání existence hrany, jelikož musíme procházet seznamy sousednost (Pro velké grafy pomalé, na druhou stranu pro řídké grafy méně pamětově náročné)

Implementace grafu pomocí matice sousednosti

Stejný graf jako v předchozím případě

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

Vytvoříme matici v*v prvků, kde v je počet vrcholů a hrany reprezentujeme jedničkami, zbytek matice jsou nuly.

Vidíme, že neorientovaný graf produkuje symetrickou matici, je tedy možné horní polovinu zanedbat a ušetřit tak paměť. Věškeré operace jsou O(1) (pokud použijeme 2rozměrné pole), paměťově je však tato implementace náročnější, jelikož alokujeme paměť i pro místa, kde hrany nejsou. Tato metoda je tedy vhodná pro časté vyhledávání ve větších grafech.

Ohodnocení lze provést uložením hodnoty místo jedniček, na diagonále lze ohodnotit vrcholy.

// co třeba hledání kostry grafu a podobný vylomeniny?

12
Prohledávání grafů

Barvení grafu

Pro přehlednost při procházení grafem jednostlivé vrcholy obarvujeme

- **bílá** - tento vrchol algoritmus ještě nezpracoval
- **šedá** - vrchol byl navštíven, ale ještě se k němu budeme vracet
- **černá** - vrchol je zpracován, už se k němu vracet nebudeme

Na začátku jsou všechny vrcholy samozřejmě bílé a postupně během průchodu je teprve bravujeme.

Prohledávací algoritmus končí v případech, kdy:

- nalezneme požadovaný vrchol
- nebo jsou všechny vrcholy grafu obarvené na černo

Prohledávání do šírky

(breadth-first search, BFS)

Realizace pomocí fronty

Vybereme počátecní vrchol, obarvíme na šedo a projdeme všechny jeho sousedy. Nenalezneme-li řešení, obarvíme počáteční vrchol na černo a pokračujeme výběrem každého z jeho sousedů a opet projdeme jejich sousedy, které obarvujeme. Sousedy zatím nevybíráme, zpracováváme nejprve vrcholy právě vybraného vrcholu tak dlouho, dokud tento není obarven na černo.

Lze také k vrcholům ukládat předchudce a pocet hran od pocátecního vrcholu, címž získáme (nejkratší) cestu k počátečnímu vrcholu a její délku. Prohledáním celého grafu získáme BFS strom. Implementace pomocí fronty, složitost O(|H| + |V|).

Prohledávání do hloubky

(depth-first search, DFS)

Realizace pomocí zásobníku

Z počátecního vrcholu jdeme do jeho prvního souseda, z něj do jeho prvního souseda atd., které obarvujeme na šedo. Vybíráme ze sousedů vrcholy, které jsou bílé, které okamžite barvíme na šedo. Pokud projdeme všechny vrcholy, obarvíme vrchol na černo. a vracíme se na šedý vrchol. Pokračujeme do nalezení hledaného vrcholu nebo prohledání všech.

Prohledáním celého grafu získáme DFS strom. Implementace rekurzí nebo zásobníkem, složitost O(|H|+|V |).

Hrany orientovaného grafu: Stromové hrany (patrící do stromu, jejich směr je od výchozího vrcholu), Zpětné hrany (opačná hrana ke hraně stromové, vytváří malý cyklus, směr je do výchozího bodu), dopředné (nemusí ležet ve strome, ale její vyrcholy v něm leží, jakésy zkratky stromem) a križující (opacné k dopredným, nebo hrany mezi stromy lesa). Zpětné a křižující hrany indikují cykly v grafu.

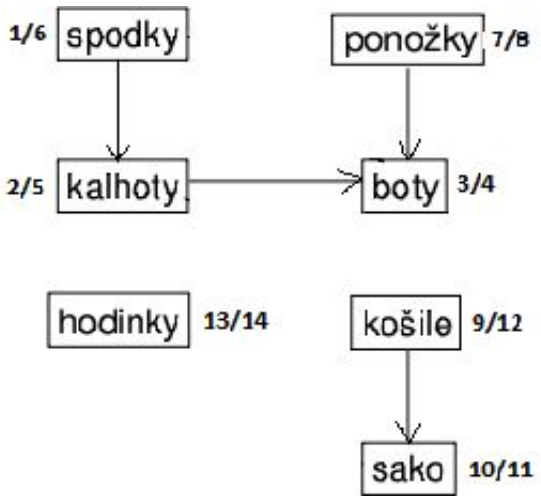
Je-li graf rozdelen do nekolika izolovaných částí, zbudou po prvním prohledávání bílé vrcholy. Mužeme vybrat jeden jako další pocátecní a znovu z neho spustit vyhledávání, prípadne postup opakovat, pokud zustanou opet bílé vrcholy. Získáme tím les stromů.

Jsou-li grafy orientovány, je vhodné začínat na vrcholu, do kterého nevede hrana, aby nedošlo ke zbytečnému roztržení grafu na komponenty

13
Topologické řazení

Máme množinu prvků, ve které je definována množina dvojic, které jsou uspořádánvy v určitém pořadí. Tuto množinu lze zakreslit jako orientovaný graf, kdy každá hrana reprezentuje pořadí prvků.

Jako příklad může posloužit proces oblékání.



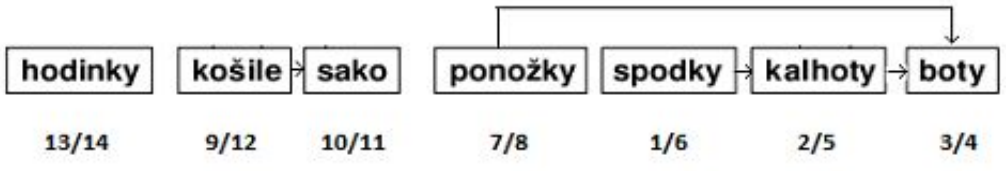
Dvojice jsou (musíme obléci):

- Ponožky dříve než boty
- Spodky dříve než kalhoty
- Kalhoty dříve než boty
- Košili dříve než sako
- a hodinky nezávisle na zbytku

V grafu vidíme, že obsahuje 3 komponenty, a že neobsahuje cykly. Pokud orientovaný graf neobsahuje cykly, nazývá se orientovaný acyklický graf (anglicky DAG, directed acyclyc graph). V takovém grafu lze nalézt uspořádání takové, že prvky budou chronologicky za sebou podle toho, v jakém pořadí je třeba je projít (vykonat jejich akce, obléci daný kus oblečení), např. takto

Jedním z algoritmů je Topologické řazení pomocí DSF:

Projdeme graf pomocí DFS, a při každé operaci (každém obarvení: nalezení/přesun na vrchol (šedá) a dokončení vyhledávání okolních vrcholů (černá)) pamatujeme čas, ve kterém jsme operaci provedli. První číslo udává "čas" nalezení vrcholu, druhé "čas" zpracování vrcholu. Pokud je vrchol dokončen, vložíme jeho prvek jej do zásobníku Výběrem všech prvků zásobníku získáme pořadí, ve kterém lze prvky [vykonat, použít, vypsat] tak, že jsou splněny jejich závislosti na poředí vykonání



Na obrázku je vidět, že seřazení může být různé pro různé výchozí body hledání (v našem případě spodky, košile, sako). Je úplně jedno, v jakém vrcholu začneme graf prohledávat, protože díky ukládání do zásobníku se ve výsledku vždyy seřadí tak, jak potřebujeme. (př: když začneme od kalhot, tak se na dno zásobníku uloží boty a nad ně kalhoty. Tady jsme skončili, takže pokračujeme v prohledávání od jiného nenavštíveného uzlu - třeba od spodků. Ty se tedy uloží nad boty a kalhoty, takže když vybíráme ze zásobníku, dostaneme správně spodky->kalhoty->boty. Pokud bychom místo spodků pokračovali třeba ponožkami, nic se nestane, jen bude do této podloupnoosti vložen ještě prvek ponožky: spodky->ponožky->kalhoty->boty, ale posloupnost bude stále zachována taková, aby položky následovaly tak jak mají.)

14

Tabulka s přímým adresováním

Máme skupinu záznamů, které jsou jednoznačně identifikovány klíčem a mají přiřazenou hodnotu (například slova ve slovníku, nebo seznam studentů podle osobního čísla). Takováto struktura se nazývá ADT Tabulka.

Složitost operací lze podle způsobu implementace rozdělit následovně:

Tabulka obsahuje mnoho hodnot vzhledem k počtu klíčů

(např. seznam otázek k SZZ z předmětu PPA2). Zde je vhodné použít pole, otázky jsou číslovány, a tyto čísla použít jako index do pole otázek. Každá otázka pak má Předmět, zadání, ... Hledání v takové tabulce je pak O(1). Do této skupiny by mohly patřit i problémy, kdy je možné prvky v lineárním čase indexovat tak, že je tento index získán v konstantním čase (například den v roce pro adresování stránky v deníku)

Tabulka obsahuje mnoho klíčů a méně hodnot

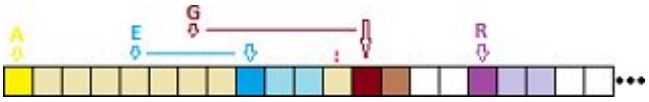
(Například počty výskytů slov této otázky. Čeština má cca 300 000 slov, ale v textu jich unikátních bude hrstka). Pro takové problémy se implementace pomocí pole nehodí, protože plýtváme pamětí (vetšina pole bude hodnota 0). Pro adresování tedy použijeme například spojový seznam. Vložení klíče je O(1), ale hledání klíče je O(n), protože musíme projít všechny položky a pro každou se podívat, jaký klíč obsahuje. Toto lze zrychlit použitím například BVS, potom bude hledání O(log n) a zásid také O(log n), kde n je počet klíčů stromu. Pokud bychom měli příklad se slovy v článku, bylo by rychlejší použít například strukturu Trie, která má vyhledávání a zápis závislý na délce slova (za předpokladu, že Trie použije v každém uzlu tabulku s přímým adresováním podle znaku. Použijeme trochu více paměti, ale pořád méně než při ad 1)

Tabulka obsahuje mnoho hodnot pro menší množství klíčů

Tato možnost se přímým adresováním řešit nedá, klíče nejsou unikátní. Je nutné použít rozptylové tabulky

15
Rozptylové tabulky s s vnějším řetězením

HASH Pokud máme množinu záznamů, ve které mohou v klíčích existovat duplicity pro různé hodnoty (například tabulka slov podle prvího písmena). Pro tento přístup je nemožné používat přímé adresování, jelikož by docházelo ke kolizím.



Můžeme ale v poli využít prázdná místa a do nich duplicitní hodnoty ukládat. Vyhladávání a ukládání potom obecně nebude v konstantním čase. Můžeme tedy například uložit duplicitní hodnotu do následujícího prázdného místa v tabulce. Problémy jsou však následující:

Můžeme zabrat místo pro klíč, který má na toto místo právoplatný nárok (kolize tedy nastanou nejen pro hodnoty se stejným klíčem, ale i obecně pro libovolný klíč) Odsuneme-li i tento klíč, dojde k vytváření shluků (a nabalování dalších klíču na tyto shluky, na obr. skluk od písmene A) Dochází také k fragmentaci, je tedy nutné hledat až do prvního prázdného klíče, což je při shlukování a vyšší saturaci tabulky pomalé

Tento přístup tedy použijeme, pokud vímě, že jsou klíče hodnot rovnoměrně rozmístěny, a že hodnot není více než klíčů, a nazývá se vnitřní zřetězení.

Vnější zřetězení znamená, že ke kažnému klíči přiřadíme seznam hodnot. Nedochází tedy ke kolizím, ale vyhledání hodnoty je O(m), kde m je počet hodnot ke každému klíči, tedy nejhůže O(n) (všechny hodnoty jsou v jednom klíči). Tento způsob však zvládá i situace, kde je více hodnot než klíčů

Rozptýlení klíčů.

Pro vnitřní zřetezení lze použít lepší vyhledáváci (a ukládací) funkci, než lineární posun, třeba kvadratickou fci. Kvadratická funkce se může protonout s jinou kv. fcí s jiným počátkem jen jednou) Tím omezíme clustery a kolize, a ukládání a hledání je rychlejší.

Pro vnější zřetezení, nebo pokud je při vnitřním zřetězení mnoho podobných klíčů, je vhodné použít orákulum, rozptylovou (hashovací) funkci. Ta má 2 vlastnosti:

Namapuje hodnoty do rozsahu počtu klíčů Ideálně zruší nebo alespoň naruší vztahy mezi podobnými klíči

Příklad: Slovník se vnitřním zřetězením. Slovo převedeme na číslo součtem ASCII hodnot znaků

Vlastnost 1: na toto číslo použijeme operaci modulo. Tato operace je pomalá (jde o dělení), použijeme tedy ideálně velikosti tabulky, které jsou mocninou 2, a po té je dělení pouhým bitovým posunem Vlastnost 2: vazby v tomto postupu existují (například anagramy se namapují do stejného pole). Použijeme tedy například následující postup: (((1. písmeno * (2k+1)) + 2. písmeno) * (2k+1)) + ... Tím zrušíme vazby mezi podobnými slovy. (2k+1) použijeme, protože jde o bitový posun a přičtení 1, což je velmi rychlé a 1 přičítáme, protože by poté modulo ztratilo význam.

Mohli bychom použít i kdyptografické hashování, jako MD5, to je však pro O(1) přístup k seznamům příliš pomalé.

16
Prioritní fronta

Speciální případ ADT podobne jako zásobník a fronta, výběr odebere prvek s nejvyšší či nejnižší prioritou (maximová resp. minimová PF) Vkládat lze prvky s libovolnou prioritou.

Implementace: polem / spoj.seznamem

Složitost podle přístupu:

Rozhraní

- vytvoření prázdný fronty
- test prázdné fronty
- vložení a výběr prvku

Lazy: Pole či seznam je neuspořádaný, vkládáme prvek vždy na konec v poli (O(1)) a na začátek v seznamu (O(1)). Při výběru musíme najít maximum, v obou případech O(n)

Eager: Pole či seznam je uspořádaný, prvek vkládáme tak, aby byla struktura seřazena. Vložení je O(n), ale výběr je O(1) (ze začátku seznamu / konce pole)

Kombinace: Implementace pomocí BVS nebo haldy - vložení i výber O(log2n)

Modifikovatelná prioritní fronta

Pomocí haldy, a

1. přidáme mapu prvek->pozice v haldě. Změnou priority a obnovením haldy je možné v O(log2n) prioritu i měnit
2. bez mapy je hledání prvku v haldě další O(log2n) čas, celkem tedy hledání O(log2n) + obnovení haldy O(log2n)

Pokud proritu příliš neměníme, způsob 1 je výhodnější, ale paměťově náročnější

17
Halda

Halda je datová struktura, která má následující vlastnosti:

- Jde o strom
- Pokud C je potomkem P, potom hodnota h(C) <= h(P)

Tato halda se nazývá MaxHeap (maximální halda), protože má v kořeni vždy větší prvek. Existuje také minHeap (minimální halda), které je seřazena opačně.

Implementace

Nejjednodušší haldou může být seřazené pole či seznam. Nejčastěji se však používá binární halda, a pojmem halda tuto binární haldu většinou myslíme, existují však i další (binominální, Fibonacciho, měkká ...)

Binární halda má ještě jednu vlastnost, vyváženost (vlastnost tvaru, úplnost stromu). Ta říká, že je halda buď vyvážená (všechny listy jsou na stejné úrovni h) a nebo je plněna zleva do prava.

Vlastnosti

Vyhledání nejvyššího prvku v haldě je triviální (jde o kořen haldy, tedy O(1))

Vkládání prvku probíhá tak, že jej vložíme do nejnižší úrovně haldy, co nejvíce doleva. Pokud je tímto porušena vlastnost haldy, vyměníme prvek s jeho rodičem a opakujeme, dokud není vlastnost haldy zcela obnovena.

Mazání prvku je podobné. V binární haldě jej provádíme tak, že najdeme prvek, který je na nejnižší úrovni zcela vlevo (mimochodem, to se dá v poli zařídit indexem, v objektech ukazatelem), vyměníme jej s mazaným prvkem a obnovíme vlastnost haldy. Mazaný prvek z haldy po té samozřejmě vyjmeme.

Implementace na poli: Má-li prvek na indexu k v poli potomky, jsou na indexech 2k+1 a 2k+2. Splněním doplňkového pravidla haldy v poli nebudou mezery, a pozice pro vložení prvku či výměnu prvku při mazání je na konci pole

Implementace objektově Jako spojová struktura, je však nutné udržovat pointer na rodiče nejnižší úrovně (to může být problematické a pomalé)

18
Algoritmy řazení O(N logN)

// přidat ukázky

Společné pro všechny: Řadíme stromem.

Obsah

- 1 Řazení haldou (heapsort)
- 2 Shellovo řazení (shellsort)
- 3 Řazení dělením (splitsort, quicksort)
- 4 Řazení slucováním (mergesort)

Řazení haldou (heapsort)

Na začátku je neseřazené pole. Postupně obnovujeme nad tímto polem haldu pro první dva, tři, čtyři atd. prvky, až máme na poli vytvořenu haldu. Následně zaměníme nejvyšší prvek (vyjmeme ho) s posledním a obnovíme vlastnost haldy na n−1 prvcích. Obnovení haldy je O(log n) a obnovujeme po každém výběru, tedy O(n log2n) To opakujeme až zbude jednoprvková halda a celé pole je seřazené. Nestabilní (složitost závisí na vstupních datech).

V kostce: na poli uděláme haldu a max prvky vyměňujeme s posledními prvky haldy a haldu obnovujeme.

Shellovo řazení (shellsort)

Posloupnost rozdělíme na podposloupnosti s krokem h (každý h-tý počínaje prvním, počínaje druhým až počínaje h − 1-ním), které nezávisle seradíme vkládáním (insertsortem). Následne snižujeme hodnotu h a provádíme totéž až do serazení s h = 1. Prvky daleko od své výsledné pozice se k ní dostanou v menším počtu kroků, než u obyčejného insertsortu.

- Shell - počátecní h = polovina délky pole, snížení vydělením dvěma
- Knuth - posloupnost 3i + 1, počáteční h = její prvek v nejbližší třetině délky pole

Jednoduchý a efektivní algoritmus, složitá analýza, nestabilní.

V kostce: rozdělení do stromu podle kroku (třeba na osminy), insertsort větví, a pak insertsort čtvrtin, polovin, konec

Poměrně zajímavě udělaná ukázka z kartama: http://www.youtube.com/watch?v=QG8hs0wqmqk

Řazení dělením (splitsort, quicksort)

Určíme prvek, jehož hodnota bude rozdělovat pole (pivot). Procházíme z obou stran pole a menší hodnoty vpravo prohazujeme s vetšími vlevo, až se oba průchody potkají. Na tuto pozici přesuneme pivot - vlevo jsou hodnoty menší, vpravo vetší. Totéž provedeme s částí pole vlevo od pivotu i vpravo od nej. Rekurze končí, je-li předán pouze jeden prvek k seřazení. Nejčasteji se pivot určuje jako krajní prvek pole (nezasahuje pak do procesu dělení). Lze použít zásobník místo rekurze. Snadno implementovatelné, nestabilní.

V kostce: rozdělení do stromu podle pivotů (prvek velikostně cca uprostřed podstromu, ideálně medián, ale může být i náhodný), v každém podstromu vzájemně vyměníme větší a menší prvky než pivot.

Výukové videjko: http://www.youtube.com/watch?v=y_G9BkAm6B8

Řazení slucováním (mergesort)

Pole je rekurzivně rozdělováno na poloviny. Dojdeme až na úroveň dvojic prvku, které seřadíme (sloučíme). O úroven výše opět po dvojicích sloučíme (kopírujeme vždy menší prvek z obou polí a posuneme se za nej) a postupujeme až sloučíme obě poloviny původního pole. Je to nejčasteji používaná metoda a je stabilní.)

V kostce: jako když mícháme karty - 2 balíčky a děláme  ???????

. Procházíme 2 podstromy a řadíme posunem 2 indexů, jeden v každém podstromu, a vždy zapíšeme větší prvek pod jedním z indexů, který posuneme

Bezva video - http://www.youtube.com/watch?v=GCae1WNvnZM

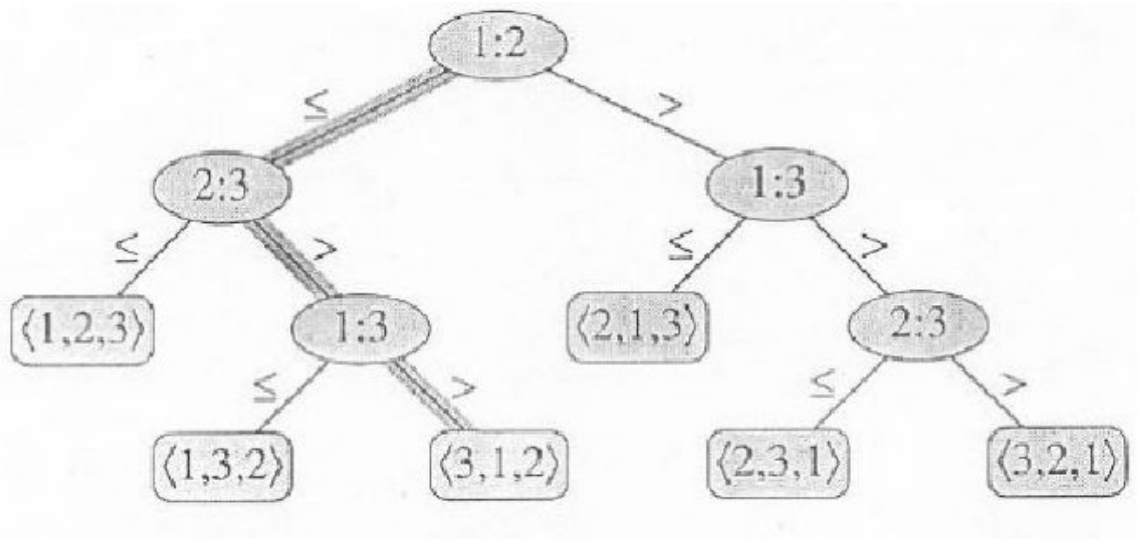
Nestabilní metody lze "zestabilnit" například zpřeházením dat na vstupu.

Radix sort =???

19
Dolní omezení pro porovnávací řazení

Celkem HC otázka, asi nejtěžší z PPA, kvůli matematice, kterou my programátoři moc nemusíme (kdo říká, že to není pravda, tak lže sám sobě, nebo neumí programovat s tím, co je k dispozici a vše si musí do podrobna analyzovat)

Zkoumejme dolni omezeni počtu porovnani T(n) pro porovnávaci algoritmy řazení.



Předpokladejme, že všechny prvky posloupnosti a1, a2, ..., an jsou různé. Porovnavací řazeni můžeme znázornit rozhodovacim stromem. Ve vnitřních vrcholech jsou prvky, ktere algoritmus porovná a v listech je permutace všech prvků původni posloupnosti, ktera je seřazena.

Příklad rozhodovaciho stromu pro řazeni vkladanim tři prvků: Jakykoliv 100% správný algoritmus (brutální síla) musi vytvořit každou z n! permutací prvků původní posloupnosti a zjistit, která je seřazená. Každou z takových permutací umístíme do stromu, ve kterém je v kořenech naznačeno porovnání, kterým se k permutacím dostaneme.

Nejhoršim připadem počtu porovnani vykonanych algoritmem řazeni je nejdelši cesta od kořene stromu k listu, je tedy rovna vyšce stromu T(n) = h .

Nyní musíme najít dolní omezení všech rozhodovacích stromů. Nechť rozhodovaci strom o výšce h má l listů, potom l = 2h (binární strom má maximálně 2h listů). Současně listů musi byt alespoň tolik, kolik je permutaci, tedy n! = l. Potom n! = l = 2h a po logaritmování (zjištění délky cesty, každá vrstva x grafu má 2x možných cest (2x listů), takže číslo vrstvy a tedy i délka ke kořeni je funkce inverzní, tedy log2x)

log2 (n!) = log2 2h = h log22 = h * 1 = h = počet porovnání, kterým se dostneme k permutaci

Použitim aproximace (n/e)n = n! je

log2(n!)=n * log2 (n/e) = log2 n- n* log2 e,

čim dostavame dolni omezeni pro nejhorši připad Ω(n*log2 n). (n * konstanta = n)

Protože pro řazeni haldou a slučovanim jsou shora omezeny časem vypočtu Ω(n*log2 n), jsou tato řazeni asymptoticky optimalni.

20
Generičnost

Genericita je možnost programovaciho jazyka definovat místo typů jen „vzory typů“, kde typy proměnných, použité v definici typu (rozuměj v typu jako ADT), jsou vyvedeny vně definice jako parametry a jsou určeny později klientskou aplikací. Základnim užitím genericity jsou třidy kontejnerů, které jsou určené k udržování skupin objektů určitého typu, například vzor třidy Seznam je definovan vlastně jako Seznam<G>, kde G je typ objektů, ktere mohou být do seznamu vloženy (kouzlo genericity vynikne pak v kombinací s dědičností, kdy do seznamu mohou být vloženy nejen objekty typu G, ale i objekty všech možných dědiců třídy (typu) G). Konkrétní typ, použitelný v textu programovacího jazyka pak vzniká, když G nahradíme skutečným existujícím typem.

Příklad:

Máme List<T> kde T je libovolný typ. Když List alokujeme, použijeme List<int> intList = new List<int>(), a tento list poté akceptuje jen čísla typu int.

Extrémní případy dovolují mnohé šílenosti, jako třeba

```
List < List < List < HodnotaTabulky > > >
```

je seznam tabulek :D

Pokud definujeme vzor (template, proto se používá T), používáme místo int prostě zástupce T

Např.

```
public interface List<T> {      // vytváříme interface, který říká, že nad typem <T> bude list
  void add(T x);                // Operace přidání prvku (typu <T>)
  Iterator<T> iterator();       // Struktura má iterátor, který definujeme níže
}
```

```
public interface Iterator<T> {  // Iterátor pohybující se po prvcích typu <T>
  T next();                     // Metoda vrací další prvek v pořadí
  boolean hasNext();            // Jen aby bylo vidět, že nemusíme jen vracet <T>
}
```

Což je takhle k ničemu, protože typ T kromě hodnoty nemá nic, co bychom mohli využít. Použijeme tedy obalení do třídy, např

```
public class Wrapper {
  T value;
  Wrapper next();
}
```

a s tou pak pracujeme.

V některých jazycích lze T omezit, třeba jen na typy implementující Iterable.

21
Dědičnost

Základním stavebním blokem OOP je dědičnost. Dědění je způsob deklarace třídy tak, že využijeme jinou třídu jako předka. Naše děděná třída tedy může

Převzít metody a atributy předka Definovat nové metody a atributy Přepsat metody a atributy předka

Ve většině jazyků jsou všechny třídy implicitně děděny od třídy Object. Ve většině jazyků (snad kromě C++) zle dědit pouze od jedné třídy. Pro další dědění lze použít rozhraní.

V Javě dědíme pomocí klíčového slova extends, tedy

Class B extends A { }

V jiných jazycích (C++, C#, ...) používáme dvojtečku, přičemž první třída za dvojtečkou je obvykle dědění, ostatní jsou implementace rozhraní (i když v C++ je to jedno, tam můžeme implementovat i dědit od více tříd)

S pojmem dědění úzce souvisí i pojem Polymorfismus. Ten nám umožňuje nejen hiearchii objektů rozšiřovat, ale také ji využívat opačným směrem, tedy dědí-li třída B od třídy A, můžeme použít konstrukci (přiřazení)

B b = new B(); A a = b;

To je extrémně důležité pro ADT, kdy vytvoříme strukturu pro A, a můžeme do ní ukládat objekty A i B.

Otázka je, pokud voláme metody A.Metoda, a metoda B tuto metodu má také, co se děje? Pro snažší pochopení řekněme, že máme typ pointeru a typ objektu. V předchozí ukázce je a typu A a b typu B, ale typ pointeru a je A a typ objektu (pod tímto pointerem) a je B.

Některé jazyky podporují Virtualitu metod. Virtuální metoda je metoda, která (je li překryta) je volána vždy z typu objektu, který překrývá. Pokud překryta není (nebo pokud je volána z třídy, která tuto virtuální metodu obsahuje), volá se ona. Voláme-li tedy a.Metoda() z předchozího příkladu, zavolá se ve skutečnosti b.Metoda(), protože a ukazuje na typ b. Java má všechny metody virtuální.

Druhou možností je použít nevirtuální metody. Taková metoda nemůže být překryta, nicméně může být znovu vytvořena (předefinována řekněme) pro daná objekt. V praxi to znamená, že je volána z typu Pointeru který na objekt ukazuje, tedy voláme li a.Metoda() zavoláme opravdu a.Metoda().

Za poslední zmínku stojí Kovariance typů. Ta znamená, že je li typ kovariantní, lze vzájemně přiřazovat obalující typy, tedy lze provést např.

pokud B extends A:

List<A> a = new List**();**

tedy je li B potomkem A, je li List<B> potomkem List<A>. Tento mechanismus však ve většině jazyků nefunguje, a je nutné vytvořit List<A> a naskládat do něj objekty B, a při výběru je jeden po druhém přetypovat zpět na B, jelikož List<A> vrací objekty typu A.

22
Rozhraní

Objektově orientované jazyky poskytují pro realizaci tříd, kromě dědičnosti, další prostředek - rozhraní. Rozhraní definuje soubor metod bez jejich implementace (podobně jako kompletně virtuální třída). Analogie je v rozepsané kuchařce, kdy víme co to bude za jídlo a z čeho budeme vařit, ale postup neznáme. Třída, která implementuje toto rozhrani (tj. implementuje - definuje - všechny jeho metody), je po té v polymorfovatelná do ukazatele definovaného tímto rozhraním, což znamená, ze klient ví, že tato třída má implementované všechny metody, které potřebuje (a které jsou definované v rozhraní).

// trochu divoký vysvětlení

Deklarace rozhraní je podobná deklaraci třídy

```
interface jmeno {
  // hlavicky metod, všechny samozřejmě virtuální a abstraktní
}
```

V Javě se implementace rozhraní zapiše pomoci kličoveho slova implements

```
Trida implements Rozhrani {
  ...
}
```

V jiných jazycích (C++, C#, ...) používáme dvojtečku, přičemž první třída za dvojtečkou je obvykle dědění, ostatní jsou implementace rozhraní (i když v C++ je to jedno, tam můžeme implementovat i dědit od více tříd). Implementovat můžeme i několik rozhraní (například implements Cloneable, Runnable)

Může se stát, že 2 rozhraní mají stejné metody k implementaci. Implementovat je tedy možné

- **Implicitně** - Implementujeme metodu první nalezené metody s tímto jménem ve všech rozhraních, nebo selže, záleží na jazyku
- **Explicitně** - Implementujeme metodu tak, že řekneme kterou metodu kterého rozhraní implementujeme např:

```
public this.type Cloneable.Clone() {
  // tělo
}
```

Nejsilnější zbraní rozhraní je, že odděluje kód od definic (vhodné např. pro pluginové systémy) a je sdružuje třídy do jedné skupiny funkčnosti.

23
Algoritmická řešitelnost problémů

Problém definujeme jako binární relaci mezi množinou instancí problému I (tj. množinou všech možností vstupu) a množinou rešení S. Dve instance mohou mít stejné rešení, stejne tak jedna muže mít více rešení. Algoritmická rešitelnost zkoumá, zda pro všechny formulovatelné problémy lze nalézt algoritmus rešení.

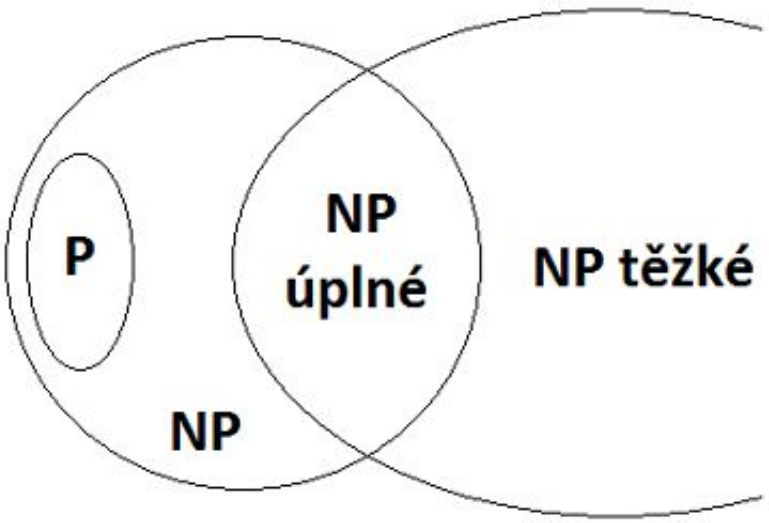
Ve 30.l. 20.st. objevil Alan Turing formální opis algoritmu - Turinguv stroj, ve spojení s Alonzem Churchem vytvorili Churchovu-Turingovu tezi a sice, že každý algoritmus (ne problém!) lze vykonat Turingovým strojem. Tezi lze vyvrátit nalezením algoritmu nevykonatelného TS. Moderní programovací jazyky pak byly navrženy tak, aby libovolný program šlo prevést na TS a naopak. Problém, který nelze vyrešit pomocí TS, tedy ani pomocí programu, je pak algoritmicky nerešitelný.

Vytvoríme-li takový rozhodovací problém (rešení je ano/ne), který pro každý rozhodovací algoritmus a alespon jeden jeho vstup dává pro tento vstup opacnou odpoved, než daný algoritmus se stejným vstupem, takový problém není rešitelný žádným z techto algoritmu a je tedy nerozhodnutelný.

První takový problém - problém zastavení - našel sám Turing: program má o všech vytvoritelných programech rozhodnout, zda pro každý ze vstupu daný program zastaví (tehdy vrací jeho výstup v podobe prirozeného císla + 1) nebo nezastaví (pak vrací 0). Takový program to ale nedokáže rozhodnout sám o sobe (pokud by byl vytvoritelný, patril by mezi zkoumané programy také a musel by ve svém výstupu vrátit svuj výstup + 1, což nejde).

24
Klasifikace problémů

Rozhodovací problémy lze dělit do tříd složitosti P (polynomiální) - problém je řešitelný pomocí (deterministického) Turingova stroje (<http://cs.wikipedia.org/wiki/Turing%C5%AFv_stroj>) v Polynomiálním čase (např problém řazení), lze dokázat, že všechny problémy P jsou podmnožinou problémů skupiny NP (NP problém s jednou větví, NP problém bez hádání či nápovědy) NP (nedeterministicky polynomiální) - Polynomiální čas, ale nedeterministický Turingův stroj (takový, který může rozvětvit program v libovolném kroku na více větví, ve kterých hledá řešení současně (zcela paralelně), případně lze mluvit o stroji, který "uhodne" kterou větví se vydat, případně mu něco "napoví" kudy se má vydat, aby řešení bylo správné. Také lze mluvit o problémech, jejichž řešení lze ověřit v polynomiálním čase (zpětný postup, zjištění, zda je nalezené řešení opdpovídající problému), nikoliv jej získat - pokud bychom všechny větve umístili řekněme do zásobníku (zásobníkový Touringův stroj), čas pro nalezení řešení by nutně nebyl polynomiální. Je otázkou, zda lze pro NP problémy najít P řešení. NP-těžké - Problémy, na které lze v polynomiálním čase převést všechny problémy NP, nemusí však nutně v NP samy o sobě být (nemusí být ani rozhodovací) NP-úplné - NP-těžké problémy, které jsou ve třídě NP, jde tedy o nejtěžší NP úlohy



A dále je tu složitost problémů, viz 6