MVŠO

ALGORITMIZACE

STUDIJNÍ OPORA PRO KOMBINOVANÉ STUDIUM

ALGORITMI-ZACE

Ing. Jiří BLAHUTA, Ph.D

Moravská vysoká škola Olomouc, o.p.s., 2017

Autor: Ing. Jiří BLAHUTA, Ph.D.

Olomouc 2017

Obsah

Úvod

Schopnost algoritmického myšlení patří mezi základní dovednosti programátora a je užitečná i pro řadu dalších odborných činností na vysokoškolské úrovni. Cílem předmětu je naučit studenty algoritmickému myšlení a principy řešení úloh na počítači. Studenti jsou seznámeni mimo jiné se základními algoritmy a datovými strukturami a naučí se je používat při řešení různých problémů.

Cílem předmětu není do hloubky se zabývat konkrétním programovacím jazykem, ale principem návrhu, analýzy a optimalizace algoritmů jako celku, nezávisle na konkrétní platformě či programovacím jazyku. Jednotlivé kapitoly jsou doplněny názornými příklady, vypracovanými v univerzálním pseudokódu k pochopení principu daného problému. Dobrý programátor se neučí pouze syntaxi konkrétního jazyka, ale umí správně uchopit problém, analyzovat ho a poté teprve přejít k samotnému programování. Primární cílem tohoto předmětu a tohoto textu tak není zabývat se jedním jazykem, ale aby studenti pochopili smysl algoritmizace v širším pojetí, jako spojení logiky, matematiky a informatiky.

V první části budou studenti seznámeni se základními pojmy v oblasti algoritmizace a programování, způsob návrhu vývojových diagramů a seznámeni s rozdělením programovacích jazyků, které se v současné době používají.

Další kapitoly navazují na teoretické poznatky a prohlubují znalosti. V kapitolách jsou uvedeny příklady k pochopení algoritmu a jeho naprogramování. V textu se studenti dále seznamují s cykly, podmínkami, větvení, datovými typy a strukturami i problematikou třídících algoritmů. V rámci předmětu se rovněž seznámí i se základy časové a prostorové složitosti, P a NP složitostí. Představeny jsou též vybrané algoritmy pro binární stromy, hashovací funkce i modulární programování. V závěru je kapitola věnovaná základům paralelních algoritmů s ukázkami v

jazyce C#. Poslední kapitola stručně seznamuje s moderními programovacími jazyky a technologie pro vývoj webových aplikací.

Náplň předmětu sleduje aktuální vývoj trendů v programování, tak i zachovává osvědčenou posloupnost probíraných témat k pochopení algoritmizace jako celku. Programování není o učení se nazpaměť pro jeden programovací jazyk, ale porozumět problému, který se má vyřešit. Dobrý vývojář, který tyto obecné znalosti algoritmizace dobře ovládá, je umí následně aplikovat v různých programovacích jazycích, poněvadž umí jednotlivé kroky analyzovat. Použití konkrétních programovacího jazyka je tak již otázkou pochopení příkazů, funkcí, objektů apod., ale analýza problému je stejná, jestli zvolíme C, C++, C#, Javu či jiný programovací jazyk. Bude se lišit paradigma či zvolené vývojové prostředí, ale programátor ví, jak problém řešit. Jeho naprogramování je pro něj až posledním krokem (samozřejmě poté ladění a optimalizace). Proto je dobré, aby studenti pochopili smysl algoritmizace jako uceleného vývoje, nikoli jako jen samotné „kódění“.

.

Kapitola 1

Pojem algoritmus

Po prostudování kapitoly budete umět:

- definovat pojem algoritmus a programování;
- popsat požadované vlastnosti algoritmů;
- vyjmenovat základní programovací jazyky a jejich rozdělení.

Klíčová slova:

Algoritmus, vlastnosti algoritmu, programování, programovací jazyk, algoritmizace.

1.1 Algoritmus a jeho vlastnosti

V předmětu Algoritmizace budete postupně poznávat principy návrhu algoritmů a jejich analýzy. Nejde o programování v konkrétním jazyce, nýbrž o chápání algoritmů jako celku, jako postupu, jak úlohu naprogramovat.

Programátorovi nestačí perfektní znalost programovacího jazyka, ale především musí vědět, jak vůbec danou úlohu řešit. Umět ji rozdělit na dílčí úlohy a poté sestavit celé řešení. Nezáleží na tom, zda řešíme součet dvou čísel nebo programujeme rozsáhlý ERP systém. Stále potřebujeme znalost postupu řešení a jednotlivých kroků. O tom je princip algoritmizace. Spoustu praktických řešených úloh algoritmů lze nalézt např. v (Vrbík, 2002).

Definujme tedy pojem ***algoritmus***. Existuje více definicí. Jednoduše lze říci, že algoritmus je postup s přesně popsanými kroky, který vede k danému cíli, tedy řešení úlohy. Formálně lze algoritmus definovat např. následovně: Jednoznačně stanovená posloupnost operací, které řeší daný problém.

S mnohými algoritmy jste se určitě setkali už mnohokrát, aniž si to možná uvědomujete. Zejména v matematice je spousta přesně daných postupů vedoucích k řešení určité úlohy. Například postup řešení kvadratické rovnice, Eratosthenovo síto k hledání prvočísel do n nebo Euklidův algoritmus pro zjištění největšího společného dělitele (NSD). De facto lze říci, že většina netriviálních matematických operací jsou algoritmy (řešení soustav lineárních rovnic pomocí GEM, výpočet determinantu matice 3×3 dle Sarrusova pravidla či např. Shorův algoritmus k nalezení periody posloupnosti)

Algoritmus musí být konečný a jednoznačný. U každého algoritmu musí být jasně definován vstup a výstup. Formálně definujeme tyto vlastnosti, které každý algoritmus musí splňovat:

- **konečnost:** Algoritmus neskončí v nekonečném cyklu
- **resultativnost:** Po konečném počtu kroků vrátí výstup (může být i chyba)
- **správnost:** Výstup musí být správný
- **determinovanost:** V každém kroku musí být zcela jasný způsob pokračování algoritmu (vč. větvení)
- **hromadnost (univerzálnost):** Algoritmus musí být popsán obecně, nikoli pro konkrétní případy (např. algoritmus pro součin čísel musí být obecný, nikoli pouze pro čísla 5 a 8)
- **opakovatelnost:** Při stejném vstupu stejný algoritmus musí dát stejný výsledek

Velmi častým pojmem je ***redukce***. To znamená, redukujeme (převedeme) úlohu, kterou neumíme vyřešit na jinou, pro kterou existuje jednodušší algoritmus. Například výpočet mediánu v obecném nesetříděném poli lze redukovat, tedy převést, na setřídění pole a výběr prostředního prvku. Více o polích a dalších datových strukturách v kapitole 6.

Můžeme rozdělit algoritmy ještě do několika zvláštních skupin:

- **rekurzivní algoritmy** – v rámci cyklu volají samy sebe, o rekurzi viz kap. 9
- **pravděpodobnostní algoritmy** – některé kroky algoritmu jsou prováděny náhodně
- **paralelní algoritmy** – využívají vícejádrové architektury současných procesorů, některé kroky se vykonávají paralelně, více v kap. 12
- **heuristické algoritmy** – není cílem exaktní výstup, užití pro časově složité úlohy, například problém obchodního cestujícího (TSP), více o složitosti viz kap. 5
- **genetické algoritmy** – využití napodobení evolučních procesorů genetiky a principů genetické operací

Uveďme si na začátek 3 jednoduché matematické úlohy a sestavíme postup, který později definujeme exaktně vývojovým diagramem (viz kap. 2).

Úloha 1: Součet tří čísel *a*, *b*, *c*.

Vstup: 3 čísla a, b, *c*

Výstup: součet *a*+*b*+*c*

1.Načtení čísel *a*, *b*, *c*

2.Provedení operace součtu *a*+*b*+*c*

3.Zobrazení výsledku

Úloha 2: Faktoriál čísla *n*

Vstup: Číslo *n*

Výstup: Faktoriál *n*!

1. Načtení čísla *n*
2. Pokud *n*<0, zobraz chybu
3. Pokud *n*=0, pak *n*!=1
4. Pokud *n*>0, spočti *n*(*n*-1)(*n*-2)…(1)
5. Zobrazení výsledku

Nejprve musíme vědět, co je *faktoriál*. Jde o součin od čísla *n* po 1, tedy např. 5!=5×4×3×2×1=120. Na příkladu vidíme jednoduchou podmínku, tzv. větvení algoritmu. Jestliže je zadané číslo menší než nula, zobrazí se chyba, jelikož faktoriál je definován jen z kladného čísla. Je-li *n*=0, pak je vždy 0!=1. Pro *n*>0 proběhne výpočet součinu.

Úloha 3: Řešení (nalezení kořenu) kvadratické rovnice v R

Vstup: Obecná kvadratická rovnice

Výstup: Kořen(y) v R

1. Není-li rovnice v anulovaném tvaru, upravit podle ekvivalentních úprav
2. Výpočet diskriminantu $D=b+4ac$
3. Je-li $D<0$, zobraz chybu, v množině R nemá řešení
4. Je-li D=0, rovnice má jeden dvojnásobný kořen $x=-b/2a$
5. Je-li $D>0$, pak vypočti kořeny $x_1=(-b+\text{sqrt}(D))/2a$ a $x_2=[-b+\text{sqrt}(D)]/2a$
6. Zobrazení výsledku

Zde opět musíme mít základní znalosti matematiky. Vědět, že anulovaný tvar kvadratické rovnice je $ax^2+bx+c=0$ a jak se počítá diskriminant. Zápis sqrt(*D*) představuje druhou odmocninu z *D*. Z programátorského hlediska může být krok 1 náročný (zlomky apod.).

Uveďme si ještě příklad nematematické úlohy. Na vstupu je slovo a úkolem je zjistit, zda je slovo palindrom (tedy čte se stejně zleva doprava i opačně). Příkladem může být slovo KUK, POP nebo OKO.

Úloha 4: Je slovo palindrom? Výstupem je ANO nebo NE.

Vstup: Slovo

Výstup: ANO/NE

1. Načtení slova
2. Srovnání 1. písmena a posledního, 2. písmena a předposledního, atd.
3. Pokud se písmena shodují, výstup bude ANO
4. Pokud se písmena neshodují, ukončit program a výstup NE
5. Zobrazení výsledku

Zde je důležitý jeden krok – pokud algoritmus narazí na neshodu písmen, nepokračuje dále a rovnou se vypíše NE. Není třeba tak procházet celé slovo, ale pouze dokud nedojde k neshodě. Více o podmínkách a cyklech v kap. 2 a 3.

Je-li algoritmus zapsán ve formě, které rozumí počítač, pak mluvíme o *programu*. Program vzniká v ***programovacím jazyce*** (forma zápisu algoritmu pro počítač) a tuto činnost nazýváme programování. *Programování* tak je zjednodušeně řečeno zápis algoritmů v „řeči“ počítače.

1.2 Programovací jazyky

Každý algoritmus je pouze teoretickým řešením postupu. Druhou částí je praktické programování. Programovat, tedy „napsat“ algoritmus v programovacím jazyce je praktickým výstupem úlohy k programování. Existuje několik desítek programovacích jazyků, z nichž cca do 10 je majoritně používáno v praxi. Každý má své přednosti a nevýhody dle programované úlohy. Můžeme je rozdělit dle několika kritérií:

- historické a současné
- procedurální a objektové (OOP paradigma)
- nižší (assembler) a vyšší (např. C++ nebo Java)
- interpretované a kompilované
- pro programování desktopových (např. C) a webových aplikací (např. PHP)

K historickým programovacím jazykům, které se dne nepoužívají, patří např. BASIC, Fortran, Ada, Cobol, Pascal. Přesto se Pascal často na školách učí z důvodu jeho relativní jednoduchosti na pochopení principů programování.

Mezi současně používané jazyky patří zejména C, C++, C#, Java, PHP. Zvláštní skupinu jazyků tvoří *funkcionální jazyky*, mezi něž patří F# z rodiny .NET a speciální využití mají též jazyky jako Ruby, Python, Perl, Haskell.

Cílem tohoto předmětu však není studium žádného konkrétního programovacího jazyka do hloubky, nýbrž pochopení algoritmů samotných, jejich návrh a analýza. Proto v tomto textu budou příklady vysvětlovány pomocí tzv. ***pseudokódu***, což je de facto univerzálně pochopitelný zápis daného řešení. Pseudokód je dobrý na pochopení principu a ten pak modifikujeme pro konkrétní programovací jazyk.

V drtivé většině se setkáte s programováním ve vyšších programovacích jazycích. Nižší programovací jazyky pracují na principu zápisu instrukcí pro procesor a adresami paměti. Vyšší programovací jazyky jsou určeny pro běžné programování aplikací. Zápis v programovacím jazyce se přeloží do strojového kódu (kterému rozumí pouze počítač) a program tak může pracovat. Nemusíme tak vůbec znát instrukce procesoru a logickou strukturu operační paměti. Předmět Algoritmizace se zabývá obecným pochopením návrhu algoritmů, nikoli jejich programováním v konkrétním programovacím jazyce.

Ve většině případů se programuje ve speciálních editorech, tzv. vývojové prostředí (Integrated Development Environment; IDE). Program můžeme psát v programovacím jazyce i v jednoduchém textovém editoru jako je Poznámkový blok, ovšem IDE poskytuje zejména přímou možnost kompilace a spuštění programu (jeho „přeložení“ do jazyka procesoru). Mezi známá vývojová prostředí patří například NetBeans, Eclipse IDE nebo Microsoft Visual Studio. Mezi obyčejným textovým editorem

a plnohodnotným IDE jsou ještě tzv. Programátorské editory, které sice nemají přímou možnost kompilace, ale mají především funkci zvýraznění syntaxe. To znamená, že například pro jazyk C# nebo Java se hlídá syntaxe a snadno odhalíme (a opravíme) syntaktické chyby (viz kap. 1.3). Mezi takové editory patří například Notepad++, PSPad nebo Bluefish. Umožňují zvýraznění syntaxe pro několik desítek programovacích i skriptovacích jazyků. V současné době roste obliba online IDE a editorů, například Ideone[1] podporující více než 60 programovacích jazyků.

1.3 Způsoby zápisů algoritmů

Existují de facto 4 způsoby, jakými lze algoritmus popsat:

- **slovně:** Vyjádříme slovně postup řešení a jednotlivé kroky (použili jsme v kap. 1)
- **graficky:** Použití vývojových diagramů a struktogramů (viz kap. 2)
- **matematicky:** algoritmus popíšeme jednoznačnou matematickou konstrukcí (např. rovnicí nebo konstrukčním popisem geometrické úlohy)
- **programem:** kroky algoritmu jsou popsány instrukcemi procesoru, resp. převedeny z vyššího programovacího jazyka, tedy algoritmus programujeme

Předmět Algoritmizace má být vodítkem, jakým způsobem algoritmy vyjadřovat, analyzovat, sestavit a řešit v pseudokódu. Jeho výhodou je univerzálnost a nezávislost na programovacím jazyce. Díky pseudokódu se mohou dohodnout na postupu řešení i například týmy z různých zemí, kteří by nevyjádřili slovní popis.

Ještě zmíníme o způsobech, jakým algoritmus navrhujeme.

- **shora dolů:** problém rozdělíme na několik podúloh, které řešíme a spojením dostaneme celý algoritmus (typická ukázka je metoda rozděl a panuj, viz kap. 6)
- **zdola nahoru:** z triviálních úloh skládáme vyšší úlohy a spojením dostaneme celý algoritmus
- **kombinace obou metod:** návrh shora dolů v některém kroku doplníme o část „zdola nahoru“

V praxi vždy záleží především na komplexnosti a povaze řešeného algoritmus, který postup bude nejlepší aplikovat.

[1] https://ideone.com/

1.4 Syntaktické a sémantické chyby

Nikdo a nic není bez chyby. Samozřejmě to platí i v této oblasti. Chybu lze udělat při návrhu algoritmu či v programování. Pokud navrhujeme algoritmus, mnohé chyby ani nepodchytíme a teprve při programování vyvstanou. Obecně rozdělujeme chyby na syntaktické a sémantické. Základní rozdíl je, že dojde-li k syntaktické chybě, překladač pro programovací jazyk nebude pokračovat, kdežto u sémantické chyby ano.

Syntaktická chyba je taková chyba, která odporuje syntaxi programovacího jazyka, v němž algoritmus programujeme. Může jít o nepovolené pojmenování proměnné, neexistující klíčové slovo, neukončení podmínky, apod. Zkrátka cokoli, co nedovolí překladači, aby kód správně přeložil do instrukcí procesoru. Protože při programování v drtivé většině případů používáme vývojová prostředí, která mají přímo kompilátor v sobě, na chyby upozorní a většinou i vypíše, na kterém řádku kódu chyba je. Lze ji tak poměrně snadno najít a opravit. Například, napíšeme-li u cyklu namísto „for“ slovo „fro“, dojde k chybě, jelikož takové klíčové slovo neexistuje.

Sémantické chyby mají jiný charakter. Na rozdíl od syntaktických chyb se program přeloží a může normálně fungovat. Ovšem nikoli správně dle požadavků. Například navrhneme algoritmus, vše správně naprogramujeme, ale nedostaneme správné výsledky. Příkladem mohou být výpočty, u nichž požadujeme pouze kladná čísla, ale program umožňuje výsledky v záporných hodnotách. Znamená to, že program funguje, ale má sémantickou chybu. Takové chyby se hledají hůře, jelikož vývojové prostředí nehlásí žádnou chybu, neboť syntaxe jazyka je v pořádku. Často na tyto chyby narazíme až při reálném používání programu. Z těchto důvodů je důležitý právě správný návrh algoritmu před jeho naprogramováním a důsledně promyslet všechny možnosti a varianty, které mohou nastat. Nejen k tomu nám pomáhají vývojové diagramy, o nichž bude řeč v následující kapitole.

Algoritmizace se zabývá návrhem, analýzou a konstrukcí algoritmů. Algoritmus je přesně daný postup splňující jednoznačná kritéria. Algoritmus lze vyjádřit 4 způsoby. Pokud je vyjádřen programem, mluvíme o jeho zápisu v programovacím jazyce. Programovací jazyky dělíme dle několika kritérií, zejména na vyšší a nižší. Programování je proces zápisu algoritmu v konkrétním programovacím jazyce, v současnosti jsou používané např. C, C++, C#, Java či PHP. Součástí této kapitoly jsou 4 jednoduché algoritmy zapsané slovním způsobem pro pochopení.

1. Stručně definujte pojmy algoritmus, program a programovací jazyk.
2. Uveďte a vysvětlete alespoň 3 vlastnosti, které každý algoritmus musí splňovat.
3. Patří Assembler mezi vyšší programovací jazyky?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 2

Vývojové diagramy

Po prostudování kapitoly budete umět:

- Pochopit význam jednotlivých kroků pomocí diagramu
- Navrhnout základní kroky algoritmu pomocí vývojového diagramu
- Vysvětlit, co vývojové diagramy popisují

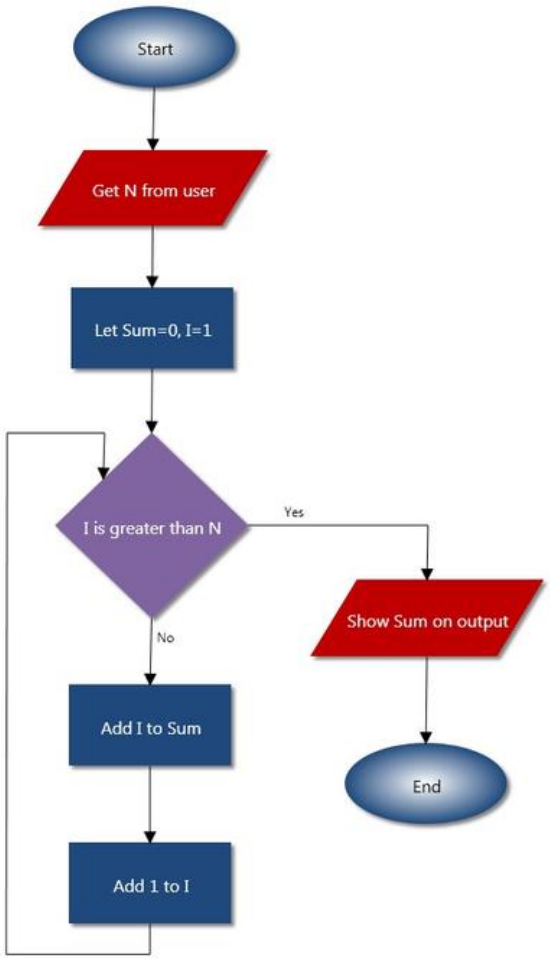
Klíčová slova:

Vývojový diagram, struktogram, flowchart, větvení

2.1 Princip vývojového diagramu

Vývojový diagram představuje grafickou, jednotnou formu popisu kroků algoritmu. Jejich výhodou je standardizace, takže každý krok má svůj grafický symbol, např. větvení. Přehledně znázorňuje kroky algoritmu, což je mnohdy efektivnější, než slovní popis nebo strohý matematický postup. Jednotlivé grafické symboly reprezentují činnost algoritmu, avšak nepopisují tok dat, pouze posloupnost kroků.

Na začátek příklad vývojového diagramu pro program, který vypíše součet čísel[2].



Obrázek 2.1 Ukázka vývojového diagramu

[2]http://tecnogomezmoreno.wikispaces.com/file/view/3.3.%20flow%20chart%20-%20Sum%20of%20N%20numbers.jpg

Na diagramu vidíme všechny základní používané symboly (tvary), které se používají pro znázornění kroků programu:

- **oválný symbol** pro začátek a konec programu (Start/End)
- **kosodélník** pro akci jako čtení vstupu nebo výstupu na obrazovku
- **obdélník** pro vnitřní akce, zde např. Add *I* to Sum (přičti *I* k součtu)
- **kosočtverec** (nebo pravidelný kosodélník) pro podmínky, z nichž se diagram větví, zde podmínka *I* is greater than *N* (*I* je větší než *N*)

Tyto základní tvary se používají nejčastěji. U podmínek se nejčastěji diagram větví do 2 částí, tedy pro případ, že podmínka platí a neplatí. Šipky označují v diagramu sled kroků, ne vždy se však používají (je dáno čtení diagramu zhora dolů, případně zleva doprava). Setkat se lze s diagramy orientovanými vertikálně i horizontálně, vertikální směr převažuje.

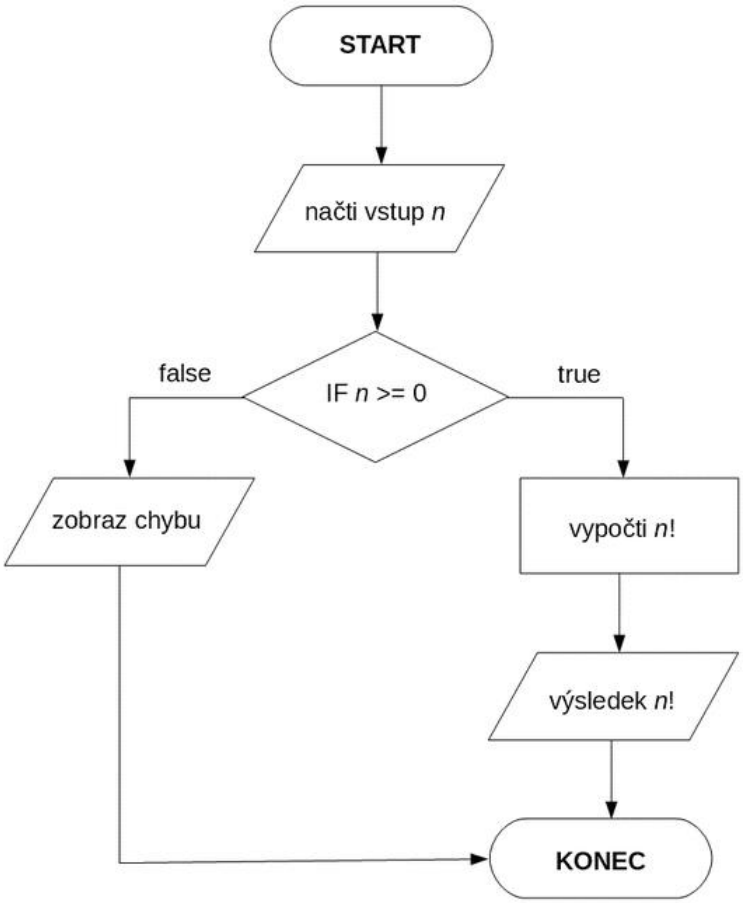
Vývojové diagramy srozumitelně a jednoznačně graficky popisují sled jednotlivých kroků programu.

Zejména v dřívějších dobách se používaly ještě tzv. ***struktogramy***. Dnes se s nimi již v algoritmizaci takřka nesetkáme, přesto měly prakticky stejný účel – rozdělit návrh programu na jednotlivé části (bloky) s popisem prováděných kroků. Obecně jsou známy pod názvem Nassi-Schieder diagram (NSD)[3]. V současnosti se NSD používají v jiných oblastech než algoritmizace, např. v psychologii.

2.2 Návrh vývojového diagramu

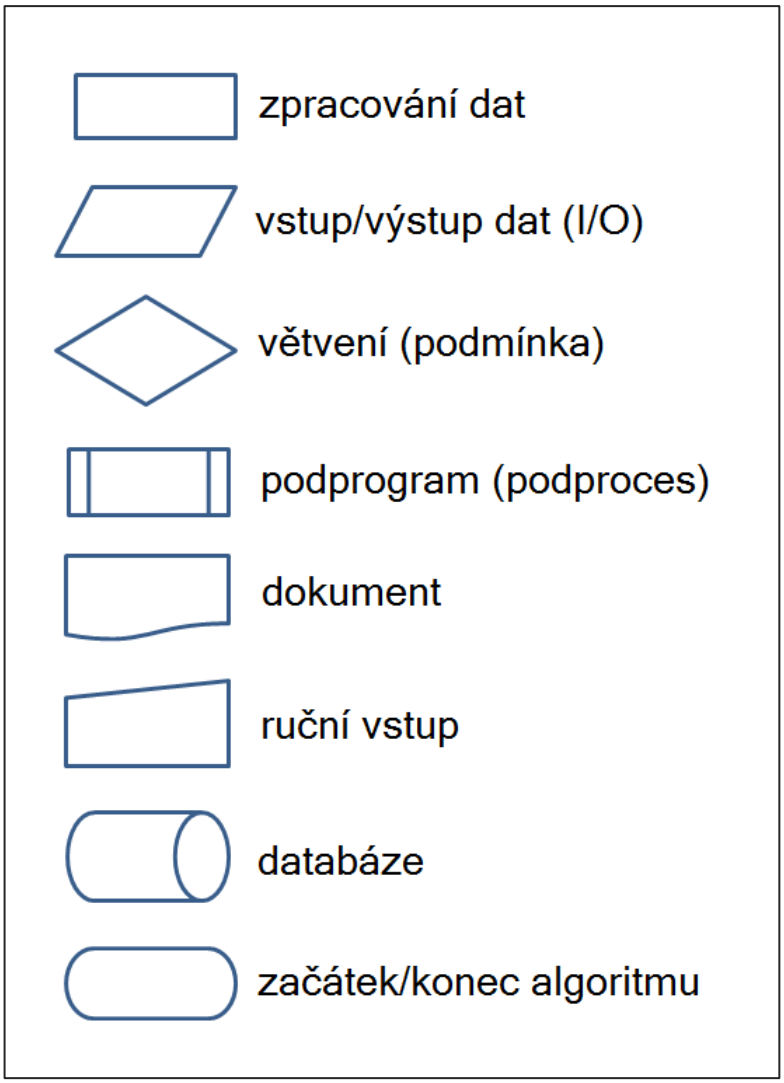
Nyní vytvoříme vývojový diagram pro algoritmus výpočtu faktoriálu zadaného čísla z kap. 1.

[3] https://www.edrawsoft.com/Nassi-Schneiderman.php



Obrázek 2.2 Vývojový diagram pro výpočet faktoriálu čísla n!

Pokud bychom navíc chtěli ošetřit i nečíselné vstupy, podmínku IF *n* >= 0 bychom mohli rozšířit na zápis IF *n* >= 0 or *n* is string, THEN zobraz chybu ELSE vypočti *n*!. Více o podmínkách v kapitole 4. Na následujícím obrázku jsou uvedeny běžně používané značky ve vývojových diagramech.



Obrázek 2.3 Základní značky vývojových diagramů

Setkat se můžete i s mnohými dalšími značkami, avšak tyto jsou obecně nejpoužívanější a nejčastější. Pro tvorbu vývojových diagramů lze použít i textový procesor, avšak specializované nástroje umějí mnohem více a komfortněji. Mezi online nástroje patří například Draw.io[4].

Mezi výhody použití vývojových diagramů patří jednoznačnost, jednoduchost grafického znázornění a univerzálnost. Značky jsou standardní a jednoduše lze popsat jednotlivé kroky algoritmu. Mezi nevýhody patří značná pracnost kreslení při složitějších algoritmech.

[4] https://www.draw.io/

Vývojové diagramy představují standardizovanou formu grafického popisu algoritmu a jeho kroků. Jednotlivé symboly reprezentují druhy kroků, například podmínky nebo I/O operace. Pomocí diagramů lze přehledně „namodelovat“ i velmi složitý algoritmus a díky šipkám vidíme přehledně sled jednotlivých kroků.

1. Co je větvení a jak se značí ve vývojovém diagramu?
2. Co nepopisují vývojové diagramy?
3. K čemu ve vývojovém diagramu slouží šipky?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 3

Datové typy, proměnné, výrazy a funkce

Po prostudování kapitoly budete umět:

- definovat a rozumět číselným a nečíselným datovým typům
- definovat pojmy proměnná, konstanta, operátor
- konstruovat výrazy, chápat prioritu operací a definovat pojem funkce

Klíčová slova:

Proměnné, výrazy, datové typy, funkce, operátory, priorita operací.

3.1 Datové typy

Datové typy určují v programu, s jakými hodnotami proměnných a konstant pracujeme. To znamená, aby číslo bylo opravdu jako číslo, text jako textový řetězec. Nejčastěji se v moderních programovacích jazycích setkáme s datovými typy **integer**, **real**, **double**, **string** a **boolean**. Datové typy integer, real a double jsou číselné (mají různý rozsah), string je textový řetězec a boolean je logická hodnota true nebo false, tedy pravda/nepravda. V některých jazycích se lze setkat též se „subtypy" pro integer – **long integer** (longint) a **short integer** (shortint). Číselné datové typy rozlišujeme celočíselné (pracující s celými čísly) a desetinné (pracující s desetinnými čísly).

Mezi textové datové typy patří **char** (z anglického charakter=znak) a **string**. Jejich hodnotou je vždy textový řetězec a to v případě, že obsahuje číslo. To bude v tomto případě reprezentováno jako text nikoli číslo. A tudíž nelze provádět matematické operace. Typ **char** reprezentuje 1 znak, kdežto **string** celý řetězec (posloupnost znaků).

Existuje mnoho číselných datových typů a každý z nich má odlišný rozsah. Mezi hlavní patří byte, **integer**, **real**, **double**, **float**. Pozor však na matematickou analogii, například datový typ **real** nejsou reálná čísla od -inf do +inf, ale jeho rozsah je omezený. Jejich rozsahy jsou následovně v otevřených intervalech:

DATOVÝ TYP	ROZSAH
Byte	(0; 255)
Shortint 2	(-128; 127)
Integer	(-32768; 32767)
Longint	(-2 147 483 648; 2 147 483 647)
Real	$(-2{,}9\times10^{-39}; 2{,}9\times10^{38})$
Double	$(-5\times10^{-324}; 1{,}7\times10^{308})$

Z tabulky je zřejmé, že pro většinu běžných operací nám bohatě postačí datový typ **real**, případně **double**. Výjimečně se setkáme s typem **extended**, který má rozsah $(-3{,}4\times10^{-4932}; 1{,}1\times10^{4932})$.

Ještě poznámka k číselným datovým typům. Setkáte se například s int a uint nebo short a ushort. Písmeno „u" na začátku značí, že tento typ neumožňuje záporná čísla a tudíž jejich rozsah v kladných hodnotách je dvojnásobný. Těmto typům se říká *unsigned* (doslova „bezznakový"), čímž se myslí absence použití unárního minus. Opakem jsou *signed*, tedy umožňují záporná čísla.

Datový typ **boolean** obsahuje pouze binární hodnoty 0 nebo 1 dle pravdivosti. Užívá se často v podmínkách (viz kap. 4), kde často požadujeme podmínku, zda je nějaký výraz pravdivý či nikoli.

Jinak řečeno, každá proměnná a konstanta musí mít uveden svůj datový typ, aby v programu bylo jasné, co vyjadřuje – číslo, text nebo logickou hodnotu.

Než přejdeme ke konkrétním popisům, pozor na záměnu pojmů datový typ a datová struktura. Datový typ určuje, co se bude zpracovávat (číslo, text, logické hodnoty), kdežto datová struktura určuje, jak budou data prezentována (řetězec, pole, atd.).

3.2 Proměnné a konstanty

Proměnná je prvek, který se v průběhu programu může měnit, oproti konstantě. Zatímco konstanta je pevně předem daná hodnota (číslo, řetězec, apod.), proměnná, jak název napovídá, se mění.

Každá proměnná musí být na začátku v programu ***deklarována***, tedy zavedena do paměti. Deklarace obsahuje datový typ a *identifikátor proměnné*, např. longint *promenna1*. Identifikátor je jednoduše název proměnné, její pojmenování. Různé programovací jazyky mívají odlišná omezení na pojmenování proměnných. Identifikátor proměnné nesmí být žádné klíčové slovo programovacího jazyka, tedy např. nelze proměnnou pojmenovat for nebo if. V různých programovacích jazycích jsou nebo nejsou názvy proměnných *case-sensitive*, tedy buď se rozlišují, nebo nerozlišují velikosti písmen. Což znamená, že někdy je proměnná **kruznice** totéž co **Kruznice** nebo **KRUZNICE**, jindy ne. Case-sensitive proměnné má např. Java či C#. Podívejme se na jednoduchou deklaraci dvou konstant *a*, *b* a proměnné *c* v pseudokódu:

```
int a=5;

int b=-21;

int c=a+b;
```

Je sice pravdou, že v tomto případě bude hodnota proměnné c vždy *c*=-16, protože čísla *a*, *b* jsou konstanty. Nicméně pokud bychom změnili hodnotu konstanty, bude proměnná nabývat jiné hodnoty.

Podobně například pro výpočet přepony pomocí Pythagorovy věty. V tomto případě budou proměnné *odvesna_a* i *odvesna_b* zadány vstupem z klávesnice, tudíž proměnná *prepona* bude vždy jiná.

```
int odvesna_a,odvesna_b;

int prepona = sqrt(a^2+b^2);
```

Konstanty jsou údaje, které se v průběhu programu nemění, mají svou deklarovanou hodnotu. Příkladem může být konkrétní číslo nebo textový řetězec apod. V celém programu mají konstanty stejnou hodnotu.

Deklarace konstanty je obdobná jako deklarace proměnné. Například v jazyce C deklarujeme konstantu klíčovým slovem **const** a překladač programu tak hned pozná, že jde o neměnnou hodnotu. Také konstanta má daný datový typ, např. int nebo char.

```
const int mojecislo = 13;
```

Pokud kdekoli v algoritmu bude použita tato konstanta, bude stále stejná. Naopak, změníme-li její hodnotu, automaticky bude mít vliv všude. Uveďme si jednoduchý příklad použití konstanty. Chceme vypsat násobilku 8 od 1 do 20. Nastavíme si proto konstantu

```
const byte nasobilka = 8;
```

a poté použijeme cyklus **for** pro generování součinu *k**8, například v pseudokódu takto:

```
for k = 1 to 20

zapis k*násobilka
```

Konstanty mají velký význam, stejně jako proměnné. Konstanta nezmění svou hodnotu od počátku deklarace, pokud ji ručně nezměníme.

3.3 Výrazy a operátory

Výrazy lze jednoduše definovat jako spojení více proměnných či konstant pomocí operátorů. Z hodin matematiky jste se jistě s pojmem výraz setkali. Je to například $(x+2)(x-2)$ nebo $2a+3b$. Podobně je to v programování. Proměnné jsou *x*, *a*, *b* a výraz je jednoduše 2+*a*, tedy konstanta 2, operátor + a proměnná *a*.

Operátory rozlišujeme **aritmetické**, **logické** a **relační**. V matematických výrazech se dodržuje používaná *priorita operátorů*. Mezi **aritmetické operátory** patří +,-,*,/,^. Operátor minus může být binární i unární. Binární minus je znakem pro odečítání, unární minus je u záporného čísla. **Logické operátory** se používají ve výrokové logice a jsou to **not**, **and**, **or**, **xor**, **implikace** a **ekvivalence**. **Relační operátory** porovnávají hodnoty nebo výrazy a patří k nim =, <,>, <=. >= a <>. Nyní podrobněji.

3.3.1 Aritmetické operátory, priorita operací

Obecně nejvyšší prioritu ve vyhodnocování výrazů mají závorky, stejně jako znáte z matematiky. Pokud výraz závorky neobsahuje, je pořadí priorit operátorů následující od nejvyšší po nejnižší:

1. **^** (umocnění)
2. **-** (unární minus)
3. ***** a **/** (součin, podíl)
4. **+** a **–** (součet, rozdíl)
5. všechny relační operátory
6. **not** (negace)
7. **and** (konjunkce)
8. **or** disjunkce

Dle uvedeného pořadí priorit vyhodnoťme následující matematický příklad:

5+4^2/-10 = 5+16/-10 = 21/-10 = -2,1.

O logických a relačních operátorech dále. K unárním operátorům patří i unární plus, pokud potřebujeme exaktně zapsat kladnou hodnotu a dále operátor inkrementace ++ a dekrementace --, což je tzv. počítadlo neboli postupné přičítání či odečítání konstantní hodnoty. Význam mají tyto operátory především v cyklech (viz kap. 4).

3.3.2 Relační operátory

Slouží k porovnávání a své využití mají zejména v podmínkách, kdy určujeme vztah (relaci) mezi dvěma proměnnými, konstantami nebo výrazy.

RELAČNÍ OPERÁTORY

OPERÁTOR	VÝZNAM A PŘÍKLAD ZÁPISU
>	Větší, x>3
<	Menší než, y<1
>=	Větší nebo rovno
<=	Menší nebo rovno
==	Rovná se, x==6
!=	Nerovná se

Všimněme si operátoru pro rovnost, že znaky rovnosti jsou dva. Je to z důvodu, že operátor „=“ je tzv. ***operátor přiřazení***, tzn., přiřazuje hodnotu proměnné nebo konstantě.

3.3.3 Logické operátory výrokové logiky

Abychom mohli popisovat ***logické operátory***, nejprve je nutno mít alespoň minimální znalosti o ***výrokové logice***. Výrok je tvrzení, o němž se dá jednoznačně rozhodnout, zda je či není pravdivý. Opakem je hypotéza. V matematické logice se setkáváme s formálními výroky. V běžném životě můžeme uvést tyto příklady výroků:

- **Karel IV. žil v 18. století.** (nepravda)
- **Existuje přirozené číslo *n*, které je větší než 10.** (pravda)
- **Venku sněží.** (vždy je možno rozhodnout o pravdivosti)

Tomu říkáme ***jednoduché výroky***. Pomocí logických spojek neboli logických operátorů tvoříme ***složené výroky***. Než k nim přejdeme, zastavme se u výroku „Venku sněží.“. Tato věta, tedy výrok, nabývá pravdivosti či nepravdivosti za předem v kontextu chápaných podmínek, tedy že myslíme, v kontextu fakt, zda právě teď v místě kde jsme, venku sněží. Ověřit to můžeme pohledem z okna apod. Výrok by to však nebyl ve smyslu obecného prohlášení „Venku sněží“, ve smyslu tom, že někde na světě venku sněží a zároveň nevíme kdy. Obecně však výrok lze prohlásit za pravdivý, neboť zcela jistě víme, že obecně venku sněží (někdy a někde). Abychom byli naprosto přesní, pak bychom museli říci „Venku sněžívá.“, což je fakt, který nelze popřít.

Zpět k logickým operátorům. Složené výroky se konstruují pomocí logických operátorů. Uvedeme pouze ty základní: **not**, **and**, **or**, **xor**. Negace znamená opačný výrok, **and** ve smyslu zároveň a operátor **or** ve smyslu nebo. Rovněž se používá terminologie disjunkce (or), konjunkce (and). Podívejme se na negaci tří výroků, které jsme uvedli výše:

- **Karel IV. žil v 18. století.** (nepravda)
- **Neexistuje přirozené číslo *n*, které je větší než 10.** (pravda)
- **Venku nesněží.** (vždy je možno rozhodnout o pravdivosti)

Dva a více výroků spojíme operátory **and**, **or** či **xor**. Příkladem může být:

Existuje přirozené číslo, pro které platí $x>20$ **and** $x<0$. (Existuje přirozené číslo $x>20$ a zároveň přirozené číslo $x<0$. Což je nepravda, neboť obor **N** jsou čísla větší než 0.)

Venku prší nebo svítí slunce. V tomto případě bude výrok pravdivý, pokud je pravdivý alespoň jeden z nich. Tedy pokud prší nebo svítí slunce. Nepravdivý bude v případě zatažené oblohy bez deště.

Nechť máme 2 jednoduché výroky *výrok1* a *výrok2*, u nichž lze jednoznačně určit pravdivost. Pak pravdivost složených výroků je dána takto:

- *výrok1* **and** *výrok2* – pravda pouze, pokud jsou pravdivé oba dva zároveň
- *výrok1* **or** *výrok2* – pravda právě tehdy, pokud je platný alespoň jeden z nich
- *výrok1* **xor** *výrok2* – pravda právě tehdy, pokud je platný právě jeden z nich (nikoli však oba)

Ve výrokové logice se definují ještě operátory implikace (označovaný ⇒) a ekvivalence (označovaný ⇔). Pravdivost výroků je dána:

- *výrok1* ⇒ *výrok2* – pravda právě tehdy, pokud nenastává, že *výrok1* je pravdivý a *výrok2* nepravdivý
- *výrok1* ⇔ *výrok2* – pravda právě tehdy, pokud jsou oba výroky nepravdivé nebo oba pravdivé

Nechť 1 je logická pravda a 0 nepravda, pak platí tato pravdivostní tabulka.

VÝROK1	VÝROK2	AND	OR	XOR	⇒	⇔
0	0	0	0	0	1	1
0	1	0	1	1	0	0
1	0	0	1	1	1	0
1	1	1	1	0	1	1

Ještě poznámka k zápisu logických operátorů. Na rozdíl od aritmetických či relačních, které jsou tvořeny znaky k tomu určenými, logické operátory se dají zapsat různými způsoby. V některých jazycích, jako například PHP, lze logické operátory zapisovat přímo slovně, tedy **or**, **and** apod. Avšak například v jazyce C nebo Java zapisujeme speciálními znaky takto: && pro operátor **and** a || pro **or**.

Tímto krátkým exkurzem do výrokové logiky jsme definovali základní logické operátory výrokové logiky. Jsou důležité například u složených podmínek, kde lze díky logickým operátorům zadat více podmínek do jedné. O podmínkách podrobně v kapitole 4. Zájemce o hlubší poznání výrokové i predikátové logiky 1. stupně odkazujeme např. na (Polák, 2016) či například na web Nabla.cz[5]. Matematická logika je základem axiomatické výstavby matematika a důkazů.

3.3.4 Bitové operátory

Tento typ operátorů patří ke složitějším na pochopení. Navazují na operátory logické, resp. z nich vycházejí. Bitové operátory nejsou logickými hodnotami, ale celými čísly rozloženými na jednotlivé bity. Definují se pro celá čísla, nejsou definovány pro plovoucí desetinnou čárku. Jelikož jejich popis je poměrně složitý, odkazujeme zde čtenáře na literaturu (Pšenčíková, 2009). K bitovým operátorům

[5]http://www.nabla.cz/obsah/matematika/vyrokova-logika-vyrok-negace-konjunkce-disjunkce-implikace-ekvivalence.php

taktéž patří ***operátory bitového posuvu***, jejich podrobný výklad s praktickým použitím čtenář najde v (Vrbík, 2002).

3.4 Funkce

V této chvíli již znáte všechny potřebné pojmy – proměnná, výraz a operátor, abychom mohli přejít k funkcím. Jelikož algoritmizace a matematika jsou velmi blízké obory, opět si vypůjčíme motivaci definice ***funkce*** z matematiky. V matematickém světě je funkce definována jako zobrazení z množiny A do množiny, kdy každému číslu x z množiny A je přiřazeno právě jedno číslo *y* z množiny B. Množiny **A** i **B** jsou neprázdné množiny, například z oboru reálných čísel $\mathbb{R}$. Například funkce $f(x)=x^2$ přiřazuje každému *x* z množiny **A** hodnotu druhé mocniny, tedy x^2 z množiny **B**. Často používané funkce mají svůj název, například logaritmus, sinus, kosinus, absolutní hodnota apod. To samé platí i pro algoritmy, potažmo programování. Každý programovací jazyk disponuje základními matematickými i nematematickými funkcemi.

V algoritmech používáme buď standardní funkce, které používá daný programovací jazyk a patří mezi základní funkce a funkce definované uživatelem. Každá funkce má své jméno (název), počet a druh proměnných, resp. parametrů. Pokud by funkce v programovacích jazycích neexistovaly, museli bychom každou i poměrně jednoduchou operaci definovat pomocí elementárních operací. Funkce, které jsou v daném programovacím jazyce, např. C nebo Java, k dispozici již předdefinované, nazýváme standardní funkce pro daný jazyk.

Mějme například funkci druhé, resp. *n*-té odmocniny. Chceme-li vypočítat druhou odmocninu z čísla *x*, v mnoha programovacích jazycích existuje standardní funkce sqrt(*x*), kde sqrt je její název (square root) a x je parametr (proměnná). Pokud by tato funkce neexistovala, bylo by nutné definovat odmocninu: Nechť je číslo *n* libovolné přirozené číslo, a nezáporné číslo, pak existuje jediné nezáporné číslo *b*, pro které platí $b^n=a$. Dále bychom museli definovat operaci umocnění. Standardní funkce velmi usnadňují mnoho početních operací a nemusíme se tak zabývat algoritmizací elementárních kroků. Vzpomeňte si na příklad výpočtu faktoriálu z kap. 1. Faktoriál rovněž patří ve většině vyšších programovacích jazyků mezi standardní funkce.

Úloha 1: Sestavte algoritmus pro výpočet absolutní hodnoty z výrazu x+10, x je z R. Využijte funkci abs(x).

1. načti proměnnou *x*

2. spočti funkci abs(*x*+10)

3. vypiš výsledek absolutní hodnoty

V pseudokódu by program mohl vypadat takto:

```
int x;

ah=abs(x+10); //pomocná proměnná ah

print ah
```

Například pro *x* = -78, by výstup byl 68. Pokud by nebyla funkce abs(*x*) k dispozici, pak by algoritmus vypadal následovně:

1. načti proměnnou *x*

2. spočti *x*+10

3. IF (*x*+10)>=0

a) THEN výsledek bude hodnota výrazu (*x*+10)

b) ELSE výsledek bude hodnota výrazu -(*x*+10)

4. vypiš výsledek absolutní hodnoty

Tedy bylo by potřeba definovat, že absolutní hodnota pro číslo *x* >= 0 je stejná hodnota a pro *x* < 0 je absolutní hodnota rovna -*x*.

Další příklad k funkcím:

Úloha 2: Výpočet sinu a kosinu úhlu zadaného ve stupňové míře a převedeného na obloukovou míru. Použijeme proměnné *uhelstup* a *uhelrad* s převodním vzorcem.

Vstup: Velikost úhlu ve stupňové míře

Výstup: Sinus a kosinus úhlu

1. Vstup int *uhelstup*

2. Převod int *uhelrad* = *uhelstup*/180*pi

3. Výpočet real *sinus* = sin(*uhelrad*) a *kosinus* = cos(*uhelrad*)

4. Výpis hodnot na obrazovce

V této kapitole byly uvedeny základní pojmy, které souvisejí s algoritmizací i programováním samotným. Patří k nim vysvětlení datových typů, co jsou proměnné, konstanty a výrazy. Výrazy se tvoří pomocí operátorů a to aritmetických, relačních a logických. Nechybí ani důležitá část věnovaná prioritě operací. Poslední část kapitoly je věnována funkcím.

1. Vyjmenujte alespoň 3 datové číselné typy.
2. Kolik znaků může obsahovat proměnná datového typu **char**?
3. Popište stručně konstrukci výrazů pomocí operátorů. Má operátor binární + nejvyšší prioritu?

Literatura k tématu:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 4

Podmínky a cykly

Po prostudování kapitoly budete umět:

- používat podmínky a znát jejich význam
- konstruovat cykly for, while a until
- spojit používání podmínek a cyklů do funkčního celku algoritmu

Klíčová slova:

Podmínky, cykly, for, while, until.

4.1 Podmínky

Podmínky a cykly patří ke stěžejním prvkům programování, resp. návrhu algoritmů.

Podmínky určují tzv. ***větvení*** algoritmu, kdy při splnění podmínky se vykoná nějaký blok, při nesplnění jiný. To znamená, že algoritmus může vykonat různé kroky při splnění různých podmínek. Opět však musejí být tyto kroky jednoznačné, což plyne z požadavků na determinismus algoritmu (viz kap. 1).

Cykly umožňují opakování určité operace dle zadané podmínky. Nejčastějšími typy cyklů, s nimiž se při programování setkáte, jsou **for**, **while** a **until**. U cyklů rozlišujeme, zda předem víme počet průchodů (opakování) či nikoli. Popíšeme si podmínky a cykly na jednoduchých příkladech.

4.1.1 Pravidla IF-THEN-ELSE, podmínka if

Jak již víme, podmínky umožní větvení kroků programu. Obecně podmínku zapíšeme jako syntaxi

výraz1 **relační operátor** *výraz2*

Což může být například $x>2$. Podmínka v obecnosti začíná klíčovým slovem **if** a to platí ve všech programovacích jazycích. Například zmíněnou podmínku bychom napsali jako

```
if x>2
```

Příklad algoritmu pro výpočet složeného úrokování a amortizace. Oba problémy mají stejné vstupní hodnoty: Výchozí částka (cena) c_0, počet let n a p% (úročení nebo odpis) za rok. Uživatel do programu zadá vstupní hodnoty c_0, n a p a volbu amortizace nebo úročení. Žádná vstupní hodnota nesmí být záporná.

Úloha 1: Algoritmus pro výpočet amortizace nebo složeného úrokování

Vstup: c_0, n, p

Výstup: cn po n letech při p%

1. vstup c_0, n a p

2. IF c_0 or n or $p < 0$ THEN chyba

3. ELSE volba A/U (amortizace úročení)

4. IF A THEN $c_n=c_0(1-p/100)^n$

5. ELSE $c_n=c_0(1+p/100)^n$

6. Výsledek

Pravidla **IF-THEN-ELSE** patří k nejpoužívanějším podmínkám. Nechť máme proměnnou *odmena* a chceme, aby při odměně vyšší než 1000 Kč byl stržen poplatek 60 Kč, při nižší než 1000 Kč bude poplatek jen 20 Kč. Pravidlo tedy bude

IF *odmena*>1000 THEN *odmena*-60 ELSE *odmena*-20, po přepisu do pseudokódu následovně:

```
if odmena>1000 then

poplatek_vysoky=odmena-60;

else

poplatek_nizky=odmena-20;

dan
```

Výstupem bude hodnota po zdanění.

Díky logickým operátorům (kap. 3) lze definovat i velmi složité podmínky, které spojujeme v jeden celek logickými operátory. Říkáme jim ***složené podmínky***. Příkladem může být:

```
if (x+y)>200 and (x-y)<40 then

//blok
```

Což říká, že je-li součet $x + y > 200$ a zároveň platí, že rozdíl $x – y < 40$, pak se něco provede. Takto lze definovat složené podmínky, kde jednotlivé výrazy jsou logickými výroky, o kterých lze rozhodnout o pravdivosti.

4.1.2 Vícenásobná podmínka switch-case

Slovo switch znamená přepínač nebo přepnout, case znamená případ a default je česky výchozí. Tedy zjednodušeně lze říct, že switch-case-break pracuje způsobem "v případě že něco, udělej něco (blok příkazů). Praktická implementace v programovacích jazycích je obdobná. V pseudokódu konstrukci switch-case zapíšeme:

```
switch vyraz

      case podminka_1
```

```
            prikaz_1;
      case podminka_2
            prikaz_2;
      otherwise
          prikaz;
end
```

Pokud nenastane případ *podminka_1* ani *podminka_2*, provede se blok za **otherwise**. Jednoduše můžeme říct, že větvení za pomoci switch-case slouží k porovnání stejného výrazu s mnoha hodnotami. Praktický příklad pro výpis jména (kód v Javascriptu):

```
jmeno = "Aleš";
switch (jmeno){
case "Tomáš" :
document.write("Tomáš");
break;
case "Felix" :
document.write("Felix");
break;
case "Lucka" :
document.write("Lucka");
break;
default :
document.write("Aleš");
}
```

V příkladu je srovnání hodnoty proměnné *jmeno*. Výchozí (default) je Aleš, což je namísto otherwise, plní stejnou funkci. Pokud *jmeno* není jeden z případů case, vypíše se výchozí jméno Aleš. Zde jsou jednotlivé případy odděleny příkazem **break**, jinak by se jich provádělo více. Konkrétní syntaxe závisí na programovacím jazyce, myšlenka konstrukce switch-case je však stále stejná.

4.2 Cykly

Cyklus v algoritmu znamená, kolikrát se má opakovat určitý krok. Často bývají cykly spojeny s podmínkami. Rozlišujeme cykly, u nichž předem známe počet kroků a cykly, u nichž počet předem neznáme a závisí na dalších vlivech, právě třeba na podmínce. Krokům v cyklu říkáme též průchody.

4.2.1 Cyklus for

Bezesporu je základním typem cyklu, u kterého předem nastavíme počet průchodů. Typickým příkladem s předem známým počtem opakování je cyklus **for**, který lze v pseudokódu zapsat

```
for i=cislo1 to cislo2 do
//blok
```

Tento pseudokód znamená, že pro nějaké *i* od hodnoty do hodnoty se provede operace v bloku. Konkrétní příklad může být:

```
for i=1 to 20 do
i++
```

Což znamená, že se přičítá postupně 1 pro i od 1 do 20.

4.2.2 Cyklus while, until

Oproti cyklu **for** pracují cykly **until** a **while** odlišně. Oba cykly se definují pomocí podmínek, nikoli exaktně stanoveného počtu průchodů. Rozdíl mezi **while** je **until** je následující:

- **while** probíhá tehdy, je-li podmínka splněna (dokud vrací pravdivost true)
- **until** probíhá tehdy, dokud není podmínka splněna (dokud vrací pravdivost false)

Obecně v pseudokódu můžeme zapsat tyto typy cyklů následovně s příkladem testování, že proměnná *alfa* > 10:

```
while alfa>10 do
// blok
```

a cyklus until takto:

```
until alfa>10 do
```

```
//blok
```

Zatímco v případě **while** bude cyklus probíhat, pokud platí, že *alfa* > 10 a v případě **until** jen dokud nenastane, že *alfa* > 10.

Cykly a podmínky patří ke stěžejním prvkům návrhu algoritmů. Podmínky umožňují větvení kroků algoritmu dle splnění určité podmínky na základě pravidel IF-THEN-ELSE. Cykly nám umožňují opakovaně provádět stejnou operaci, opět na základě podmínek. Rozlišujeme cykly s předem známým počtem průchodů (opakování), což je cyklus **for** a cykly, u nichž je stanovena podmínka výrazem, patří sem cykly **while** a **until**.

1. Co je větvení algoritmu a co znamená podmínka **if**?
2. Vysvětlete princip cyklu **for**. Je to cyklus s předem známým počtem průchodů?
3. Popište rozdíl mezi vykonáním cyklu **while** a **until**.

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 5

Výpočetní složitost algoritmů, třídy P a NP

Po prostudování kapitoly budete umět:

- definovat časovou a prostorovou složitost pro optimalizaci algoritmu
- rozlišit složitostní třídy P a NP a jejich význam při návrhu algoritmu
- popsat některé příklady NP-úplných úloh

Klíčová slova:

Výpočetní složitost, časová složitost, prostorová složitost, optimalizace algoritmu, P, NP, NP-úplnost.

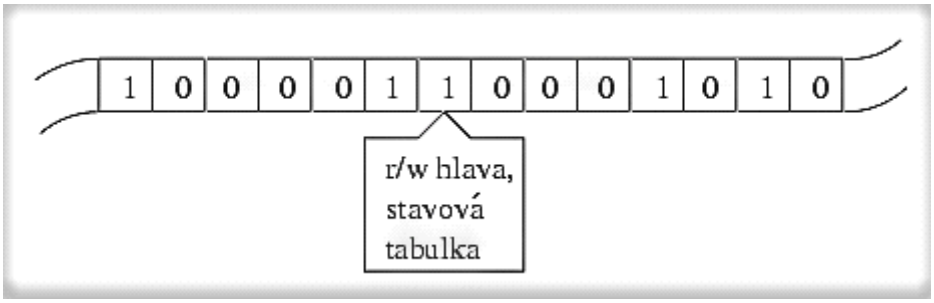
5.1 Co je složitost algoritmu

Často se při návrhu algoritmů zajímáme o dva faktory – jak bude algoritmus rychlý a kolik spotřebuje paměti. Formálně se těmito věcmi zabývá teorie složitosti, přesněji řečeno časová a prostorová složitost algoritmu. Existuje několik tříd časové a prostorové složitosti, které algoritmy klasifikují dle časových a paměťových nároků. Tyto třídy hrají stěžejní roli při optimalizaci algoritmů, kdy se snažíme o co nejefektivnější běh, tedy aby byl co nejrychlejší a zároveň spotřeboval nejméně paměti.

Tato kapitola bude matematicky poněkud náročnější, jelikož ***teorie složitosti*** je matematika na vyšší úrovni. Seznámíte se s pojmy časová a prostorová složitost, složitostní třídy, P a NP problémy, NP-úplné problémy a asymptotická složitost. Teorie složitosti patří k náročnějším postupům optimalizace algoritmů. U obou typů složitosti nás zajímají jejich třídy, pomocí nichž je možné analyzovat složitost algoritmu a na základě toho navrhnout možnou optimalizaci.

Součástí teorie složitosti je pojem ***rozhodnutelnost***. Nechť máme úlohu U. Ta je rozhodnutelná, právě tehdy když pro ni existuje algoritmus, který pro libovolný vstup dodá požadovaný výstup v konečném čase. Viz vlastnosti algoritmu v kap. 1. Pokud takový jednoznačný algoritmus neexistuje, úlohu U nelze považovat za rozhodnutelnou.

Třídy složitosti se obecně definují pro abstraktní model počítače zvaný ***Turingův stroj*** (dále jen TS). Turingův stroj je abstraktním modelem počítače s nekonečnou páskou a čtecí hlavou. Hlava postupně čte symboly z pásky a dle vnitřního stavu lze rozhodnou o dalším kroku. Velmi zjednodušeně lze Turingův stroj ilustrovat následujícím obrázkem.



Obrázek 5.1 Turingův stroj

Formální definice ohledně konečných automatů a TS lze najít v doporučené literatuře či například v prezentaci od Jana Konečného z Univerzity Palackého v Olomouci[6].

[6] http://phoenix.inf.upol.cz/~konecnja/vyuka/2014W/VYSLfiles/01.pdf

Turingův stroj a jeho abstraktní model je základním modelem pro rozhodnutelnost. Turing-Churchova teze říká, že každý algoritmus je implementovatelný nějakým TS. Díky TS je možné všechny problémy rozdělit na 2 základní skupiny:

- problémy (funkce) algoritmy nevyčíslitelné, nejsou ani částečně rozhodnutelné
- problémy (funkce) algoritmicky aspoň částečně rozhodnutelné

Příklad algoritmicky nevyčíslitelného problému je sestrojení algoritmu, který by rozhodl o každém algoritmu rozhodl, zda jeho činnost po *n* krocích skončí či nikoli.

Smysl má zkoumat pouze problémy, které lze algoritmizovat, tedy jsou algoritmicky vyčíslitelné.

Složitostní třídy dělíme na v základu deterministické a nedeterministické.

5.1.1 Funkce složitosti, asymptotická složitost

Složitost obecně je matematickou funkcí. Vyjadřuje závislost sledovaného parametru (paměťového prostoru nebo spotřebovaného výpočetního času) na množství vstupních dat. Funkci složitosti zapisujeme tzv. O-notací, matematicky $\boldsymbol{O}(f(N))$. V praxi se vyjadřují ***třídy složitosti***, nikoli přesné vyjádření. Aditivní a multiplikativní konstanty obvykle nehrají roli, tzn. například $\boldsymbol{O}(1000+N)$ či $\boldsymbol{O}(500N)$ má stále charakter $\boldsymbol{O}(N)$. Rozlišujeme 3 typy složitostí:

- **dolní odhad složitosti** – označuje se $\boldsymbol{\Omega}(f(N))$ a vyjadřuje nejefektivnější složitost, ideální stav, platí ovšem pro pouze určitou podmnožinu vstupních dat
- **horní odhad složitosti** – nejhorší možná složitost, označuje se $\boldsymbol{O}(f(N))$
- **průměrná složitost** – očekávaná při náhodném vstupu, označuje se $\boldsymbol{\Theta}(f(N))$

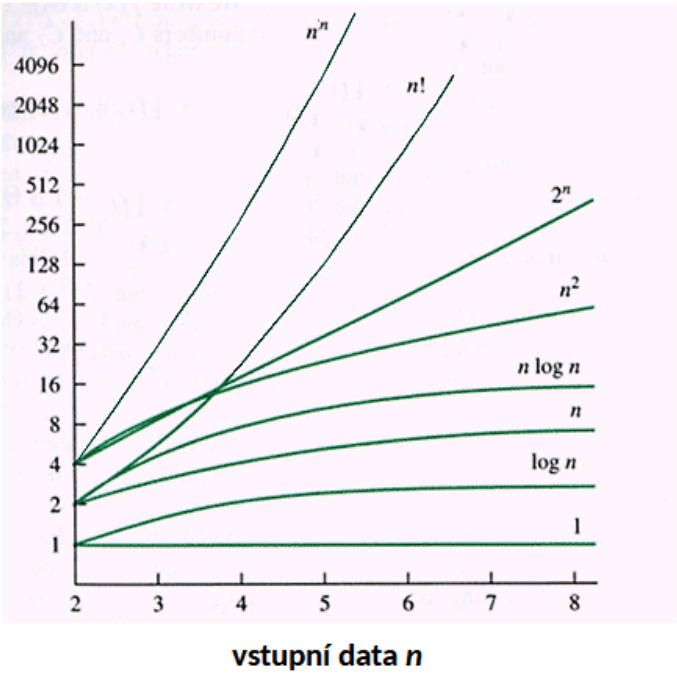
V praxi se obvykle používá horní odhad, jelikož zahrnuje i nejhorší (extrémní) případy vstupních dat.

Musíme zavést pojem ***asymptotická složitost***. Ta vyjadřuje chování funkce. V případě dolního odhadu složitosti říkáme, že algoritmus probíhá asymptoticky stejně rychle nebo pomaleji, než funkce $f(N)$. U horního odhadu pak platí, že algoritmus probíhá asymptoticky stejně rychle nebo rychleji než funkce $f(N)$. Pro průměrnou složitost platí, že algoritmus probíhá asymptoticky stejně rychle jako funkce $f(N)$. Asymptotická složitost je motivována z geometrie pojmem asymptota, která se limitně blíží.

Tímto definujeme časové a prostorové složitostní třídy. Následující tabulka uvádí třídy složitosti, s nimiž se setkáme v praxi.

OZNAČENÍ SLOŽITOSTI	NÁZEV
$\boldsymbol{O}(1)$	konstantní
$\boldsymbol{O}(\log n)$	logaritmická
$\boldsymbol{O}(n)$	lineární
$\boldsymbol{O}[n(\log n)]$	lineárně logaritmická
$\boldsymbol{O}(n^2)$	kvadratická
$\boldsymbol{O}(n^3)$	kubická
$\boldsymbol{O}(2^n)$	exponenciální

Jaká je rychlost nárůstu složitostech tříd, vyjadřuje názorně graf níže[7]. Jsou to vlastně grafy příslušných matematických funkcí.



Obrázek 5.2 Složitostní třídy v závislosti na velikosti vstupu n

[7] http://www.cs.odu.edu/~cs381/cs381content/function/growth_files/summary.gif

Na webu O-Big Cheat Sheet[8] najdete přehlednou tabulku srovnání časové a prostorové složitosti pro různé algoritmy, například typy řadících algoritmů.

5.2 Časová složitost

Časovou složitost algoritmu definujeme jako počet kroků, které algoritmus provede na základě velikosti vstupu (vstupních dat). Velikost vstupních dat je měřitelná v bitech. Můžeme říci, že časová složitost je kritičtější, nežli prostorová, neboť čas si nijak nekoupíme.

5.2.1 Časové složitostní třídy

Mezi časové složitostní třídy patří **PTIME**, **NPTIME**, **EXPTIME**, **NEXPTIME**.

Třída PTIME zahrnuje všechny úlohy řešitelné algoritmem v polynomiálním čase. EXPTIME obdobně zahrnuje úlohy řešitelné v exponenciálním čase.

Mezi nejdůležitější časové složitostní třídy patří NPTIME, zkráceně NP. Patří sem všechny rozhodovací úlohy, pro něž existuje polynomiální algoritmus rozhodující, zda je řešení správné.

5.3 Prostorová složitost

Prostorovou složitost definujeme podobně jako časovou složitost, jen na rozdíl od času měříme využití paměti a diskového prostoru v závislosti na vstupu. Jednoduše řečeno, jaké nároky algoritmus má, kolik paměti a diskového prostoru bude potřebovat.

5.3.1 Prostorové složitostní třídy

K prostorovým složitostním třídám patří tyto: **PSPACE**, **NPSPACE**, **EXPSPACE**, **NEXPSPACE**.

Třída PSPACE zahrnuje všechny úlohy řešitelné algoritmem s polynomiální spotřebou paměti. EXPSPACE obdobně zahrnuje úlohy řešitelné s exponenciální spotřebou paměti.

[8] http://bigocheatsheet.com/

Všechny složitostní třídy definujeme jako deterministické a nedeterministické (mají na začátku písmeno N, např. NPSPACE či NEXPTIME).

5.4 Základní vztahy mezi složitostními třídami

Významné rovnosti a inkluze složitostních tříd:

PSPACE=NPSPACE
EXPSPACE=NEXPSPACE
$P \subseteq NP$
$EXP \subset NEXP$
$PSPACE \subseteq NEXPSPACE$
$NP \subseteq PSPACE = NPSPACE \subseteq EXP \subseteq NEXP$
$LOGSPACE \subseteq NLOGSPACE \subseteq P \subseteq NP$

Tyto vztahy uvádíme bez důkazu. Problematika dokazování vztahů složitostních tříd patří do vyšší matematiky. Zájemce odkazujeme na literaturu, např. (JaJa, J., 2002) nebo uvedené webové zdroje.

5.5 P a NP problémy, NP-úplnost

Třída P představuje rozhodovací úlohy, které lze vyřešit algoritmem v polynomiálním čase. Rozhodovací úloha je taková, u níž je výstupem ano/ne, resp. logická hodnota true/false. Úloha U je NP-těžká, jestliže platí, že libovolnou úlohu ze třídy NPTIME je možné v polynomiálním čase redukovat na úlohu U.

Úloha U je NP-úplná, právě tehdy když je NP-těžká a současně sama patří do třídy NPTIME (NP).

Třída NP představuje rozhodovací úlohy, které mohou být vyřešeny nedeterministickým algoritmem v polynomiálním čase.

NP-úplné úlohy patří k nejtěžším mezi NP třídou. Dosud je jednou z největších otázek teoretické informatiky, zda platí či neplatí rovnost P=NP. NP-úplné problémy jsou podmnožinou NP, platí tedy NP-úplné $\subset$ NP.

P a NP je pouze jiné označení pro třídy PTIME a NPTIME.

Mnohé NP-úplné problémy se týkají problémů z teorie grafů, například známý problém obchodního cestujícího (TSP; Travelling Salesman Problem), obarvení grafu *k* barvami nebo úloha splnitelnosti booleovských formulí. Problém obchodního cestujícího spadá mezi diskrétní kombinatorické optimalizační problémy. Nechť je dána neprázdná množina n měst. Problém spočívá v nalezení nejkratší okružní cesty, aby cestující každé město navštívil právě jednou. Formálně se jedná o problém z teorie grafů, konkrétně o nalezení nejkratší hamiltonovské kružnice. Samotný problém nalezení hamiltonovské kružnice je NP-úplný. V bakalářské práci (Pokorná, P., 2008) je zajímavá tabulka uvádějící, jak rychle roste počet cest v závislosti na počtu měst. Například pro pouze 10 měst existuje 181440 možných cest. V roce 2004 byla nalezena nejoptimálnější cesta pro 24978 měst. Problém TSP je obecně pro *n* měst stále otevřený a je NP-úplný.

Tento krátký exkurz do teorie složitosti je pouze zcela úvodní. Není zde prostor pro matematické definice a důkazy. Spíše jde o to, že teorie složitosti je podstatnou částí návrhu algoritmu v praxi a pro hlubší pochopení je třeba řádně pochopit všechna úskalí a chápat souvislosti mezi třídami, vyčíslitelností na TS a jakým způsobem se vůbec dá měřit složitost algoritmů.

V této kapitole byly vyloženy a vysvětleny základní pojmy týkající se teorie složitosti. Časová a prostorová složitost a jejich třídy. Asymptotická složitost a odhady složitosti. Časová složitost vyjadřuje časovou náročnost algoritmu a prostorová složitost náročnost na paměťový prostor. Třídy složitosti vyjadřují efektivitu algoritmu časově a pro spotřebu paměti a diskového prostoru. Analýza složitosti algoritmů je výchozím bodem pro jejich optimalizaci a je důležitou součástí především při návrhu komplexnějších algoritmů, u nichž chceme zajistit co nejnižší náročnost.

1. Co vyjadřuje časová a prostorová složitost?
2. Vysvětlete rozdíl mezi dolním a horním odhadem složitosti.
3. Co je složitostní třída PSPACE a co znamená NP-úplný problém?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 6

Lineární datové struktury

Po prostudování kapitoly budete umět:

- Rozlišit typy základních lineárních datových struktur
- Vysvětlit použití struktur množina, pole, fronta a spojový seznam
- Aplikovat tyto struktury na jednoduchých algoritmech

Klíčová slova:

Lineární datové struktury, množina, pole, fronta, LIFO, FIFO, spojový seznam

6.1 Lineární datové struktury

V této kapitole se budeme zabývat lineárními datovými strukturami včetně struktury abstraktních a dynamických.

Lineární datové struktury představují způsob uložení několika prvků v množině. Například namísto toho, abychom uložili deset izolovaných proměnných stejného datového typu (např. čísla), uložíme všechna čísla do této struktury jako celek. Lineární datové struktury se vyznačují tím, že prvky množiny dat jsou uloženy za sebou. Tedy každý prvek kromě prvního má svého předchůdce a rovněž každý kromě posledního má svého následníka. Lineární datové struktury mohou být statické (pole), u nichž je počet prvků předem dán a dynamické, kde se počet prvků v nich uložených mění v průběhu algoritmu. Takovým příkladem je například seznam a tzv. ***abstraktní datové typy*** (ADT) fronta či zásobník. Ty se dají implementovat právě pomocí polí a seznamů.

6.2 Seznam

Mezi nejjednodušší a zároveň nejobecnější datové struktury patří seznam (angl. list). Seznam definujeme jako neprázdnou množinu prvků, které jsou uloženy lineárně za sebou. To znamená, že *k*-tý prvek je před *k*+1 prvkem v seznamu. Na seznamu jsou definované operace přístupu k prvkům, mazání prvků, přidání prvku do seznamu, vyhledávání v seznamu, spojení a rozdělení seznamu.

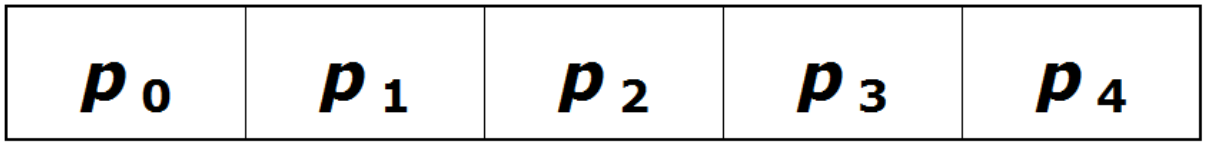
Pomocí seznamu se dají implementovat abstraktní datové typy fronta a zásobník, viz níže v podkapitole 6.4.

Pozor, neplést si se spojovým seznamem, který patří mezi dynamické datové struktury, viz dále.

6.3 Pole

K základním lineárním datovým strukturám patří ***pole***, angl. array. Jednotlivé prvky pole jsou odlišeny, obvykle číselným, indexem. Pole může být číselné i nečíselné s textovými řetězci, např. názvy dnů v týdnu. Stejně jako proměnné, také pole je nutno v programu deklarovat. Právě deklarace pole říká předem, kolik bude mít prvků a jaký je rozměr pole. V praxi se setkáme s poli jednorozměrnými

(vektor) a dvourozměrnými (matice *m*×*n*). Každý prvek má svůj index, tedy pořadí v poli. U jednorozměrného pole je jasná pozice od počátku, u matice je počet indexů roven rozměru matice, tedy *m*×*n*. Názorně se představme pole takto:



Obrázek 6.1 Schematické znázornění pole s indexovanými prvky

Každý prvek má svůj index, první prvek má index 0. Nechť máme tedy pole o n prvcích, poslední prvek bude mít index roven *n*-1. Viz obrázek – pole s pěti prvky a poslední index je 4. Konkrétně by v poli mohlo být uloženo třeba 5 čísel typu integer nebo real. Pole patří k datovým strukturám, u nichž musí předem deklarován (zadán) typ, název a rozsah.

Deklarace pole v programu je stěžejní záležitost. Pomocí ní řekneme, jakou strukturu a velikost bude pole mít. Například v jazyce C# deklarujeme pole o 20 prvcích následovně

```
int[] mojepole = new int[20];
```

A podobně tomu je v jiných programovacích jazycích. Klíčové slovo **new** je zde tzv. konstruktor.

Ukažme si praktický příklad deklarace a inicializace pole. Deklarace a inicializace pole jsou dva odlišné kroky. Deklarace uvádí datový typ, jméno pole a počet prvků. Inicializace pole je naplnění pole hodnotami. Mějme příklad, že vytvoříme 2 pole: první obsahuje názvy dnů v týdnu a druhé jejich číslo od 1 do 7. Obě pole tak budou mít 7 prvků. Uveďme deklaraci a inicializaci pole v jazyce C#:

```
string[] nazvydnu = new string [7];

byte[] cisladnu = new string [7];

string[] nazvydnu = {'pondělí','úterý','středa','čtvrtek','pátek','sobota','neděle'};

byte[] cisladnu={'1','2','3','4','5','6','7'};
```

Vytvořili jsme tato dvě pole:

nazvydnu	pondělí	úterý	středa	čtvrtek	pátek	sobota	neděle
cisladnu	1	2	3	4	5	6	7

Obrázek 6.2 Textové a číselné pole

Pole tak může být nejen číselné, ale i textové. U číselných polí využijeme nejen řadící algoritmy a prohledávání pomocí binárních stromů (kap. 10).

V tomto textu jsme si ukázali pouze jednorozměrné pole. Setkat se však můžete i s poli vícerozměrnými, základním typem je dvourozměrné pole, což je matematicky matice o velikosti $m \times n$. Rovněž se lze setkat s obecně n-rozměrnými poli[9].

6.4 Dynamické datové struktury

Dynamické datové struktury mají tu vlastnost, že předem neznáme počet prvků v nich obsažených, na rozdíl od deklarace pole, v němž je vždy počet prvků dán deklarací. Setkáme se rovněž s označením abstraktní datové typy, jelikož jsou abstrakcí a jejich implementace probíhá například pomocí pole nebo seznamu.

6.4.1 Fronta a zásobník

Pojmy fronta a zásobník si patrně dokážete představit z běžného života. Fronta u pokladny nebo zásobník v automatické pušce. Na tomto principu pracují i tyto dvě datové struktury. Obě slouží k dočasnému ukládání dat během algoritmu. Odlišují se způsobem práce s uloženými prvky. ***FIFO*** neboli First In First Out (první přichází, první odchází) je reálným příkladem fronty. Přijdete-li k pokladně první, odcházíte také první. Naopak ***LIFO*** je Last In First Out (poslední přichází, první odchází) je opět analogií reálného zásobníku pušky. Obě tyto struktury patří mezi dynamické množiny, neboť jejich prvků není předem znám. ***Fronta*** i ***zásobník*** mají společné vlastnosti:

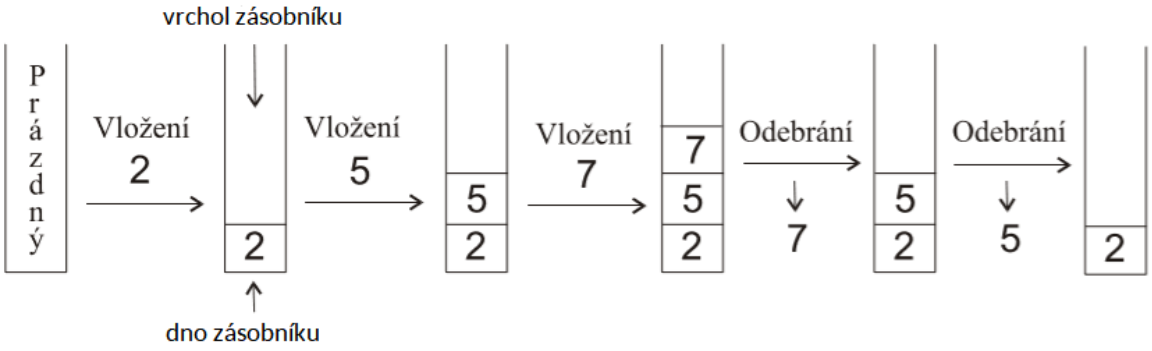
- dynamické lineární datové struktury
- lze je implementovat v programu pomocí polí nebo spojovým seznamem

[9] https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=65556

Základním rozdílem je právě způsob práce s prvky, což bude patrné nejlépe z následujícího textu a ilustrativních obrázků[10].

Zásobník je obecně velice využívanou dynamickou strukturou. Jeho použití je především pro ukládání informací o stavu programu. Se zásobníkem se setkáme u prohledávání do hloubky a v základu všech rekurzivních algoritmů.

Zásobník má dno a vrchol. Dno je neměnné a na opačném konci je vrchol zásobníku. Zásobník má definované dvě operace – uložení prvku na vrchol zásobníku a odebrání prvku z vrcholu. Tedy, operace probíhají výlučně na vrcholu zásobníku a ostatní prvky nejsou tak přímo dostupné. Na začátku je zásobník prázdný, tedy dno=vrchol. Nejlépe lze ilustrovat operace zásobníku na následujícím obrázku:



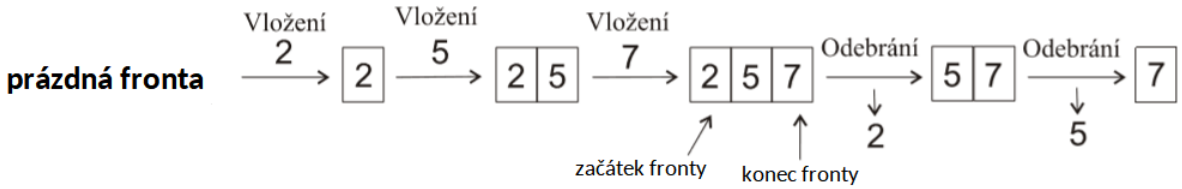
Obrázek 6.3 Schéma principu operací s prvky v zásobníku

Velký význam má zásobník u Java Virtual Machine (JVM), u něhož tvoří nedílnou součást jeho funkcionality[11].

Fronta má rovněž jako zásobník definované operace vložení a odebrání prvku z fronty. Uložení prvku probíhá jeho zařazením na konec fronty. Odebrání prvku naopak odebere první prvek ve frontě. Je to skutečně algoritmická analogie fronty u pokladny – přijde-li nový zákazník, je na konci fronty (přidání prvku) a naopak, který zaplatil, odchází (odebrání prvku). Opět názorný obrázek operací ve frontě.

[10] https://phoenix.inf.upol.cz/esf/ucebni/zakladni_alg.pdf

[11] https://www.codeproject.com/articles/30422/how-the-java-virtual-machine-jvm-works



Obrázek 6.4 Schéma principu operací s prvky ve frontě

Fronta má dva speciální případy. Speciálním případem je tzv. prioritní fronta (prvky mají danou prioritu a prvky s vysokou prioritou mohou "předběhnout" prvky s nižší prioritou). Je to analogie, když ve frontě u pokladny bude někdo, komu jede za 5 minut vlak a tak jej někdo pustí. Druhým případem je tzv. kruhová fronta, kde začátek i konec fronty je na $i = 0$, z čehož plyne obtížný dotaz na prázdnost fronty.

6.4.2 Množina

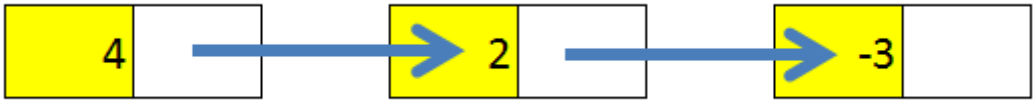
Množina patří k abstraktním datovým strukturám, u nichž nezáleží na pořadí prvků. Je implementována dle matematické definice množiny, tudíž každý prvek v ní může být nejvýše jednou. Tedy, prvek v množině je nebo není. Příkladem je například množina přirozených čísel, kde každé číslo je v množině právě jednou nebo jakákoli omezená množina prvků, nemusejí to nutně být jen čísla.

6.4.3 Spojový seznam

Spojový seznam je dynamickou datovou strukturou, obsahující 1 či více datových položek stejného typu. Položky jsou vzájemně provázány odkazy pomocí referencí či ukazatelů. Aby seznam byl lineární, musí být splněno, že neexistují cykly v odkazech. Rozlišují se tři typy seznamů:

- **jednosměrný** – každá položka odkazuje pouze na následující položku
- **obousměrný** – položka odkazuje na následující i předchozí položky
- **kruhový** – cyklem navazuje začátek a konec seznamu

Spojový seznam lze jednoduše znázornit následujícím způsobem na obrázku níže.



Obrázek 6.5 Schéma spojového seznamu

Na obrázku je jednosměrný seznam, tedy každá položka (uzel) odkazuje pouze na uzel následující.

Spojový seznam je implementačně poměrně jednoduchý, je zároveň univerzálně využitelný – je základem mnoha implementací zásobníku, fronty, grafu i jiných datových struktur.

Lineární datové struktury patří mezi základní součásti algoritmů. Mezi nejvíce používanou strukturou je pole, které může být jednorozměrné i vícerozměrné. Každý prvek v poli má svůj jedinečný index. Mezi další struktury patří seznam. Dále v kapitole jsou stručně představeny abstraktní datové typy fronta a zásobník, které se programují pomocí pole nebo seznamu. Fronta i zásobník patří k velmi používaným dynamickým datovým strukturám. Dynamické struktury jsou takové, u nichž se mění počet prvků v průběhu algoritmu. Naopak statické pole má předem definovaný datový typ a rozměr.

1. Co je pole a jaké má vstupní parametry u deklarace?
2. Vysvětlete stručně rozdíl mezi statickou a dynamickou datovou strukturou.
3. Jaký je rozdíl mezi LIFO a FIFO přístupem k prvkům u zásobníku a fronty?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 7

Řazení, řadící algoritmy – insertion sort, selection sort

Po prostudování kapitoly budete umět:

- definovat problém řazení (třídění) a jejich algoritmů
- vysvětlit princip algoritmu insertion sort
- vysvětlit princip algoritmu insertion sort

Klíčová slova:

Řazení, řadící algoritmy, třídění, sorting, insertion sort, selection sort.

7.1 Řadící (třídící) algoritmy

Než přikročíme k samotnému popisu jednotlivých řadících algoritmů, vysvětlíme si nejprve problém řazení. Problém řazení spočívá v seřazení číselných prvků v poli podle velikosti. Máme-li 10 čísel, hravě to zvládneme sami, ale co když je čísel stovky či tisíce? A nejsou jen celá? Právě pro to se využívají více či méně efektivní řadící neboli třídící algoritmy. V řadě případů potřebujeme mít pole čísel seřazené vzestupně či sestupně. Existuje mnoho řadících algoritmů s různou efektivitou pro menší nebo vetší pole. V tomto textu si představíme stručně princip následujících:

- **insertion sort** a **selection sort** – kapitola 7
- **bubble sort** a **quick sort** – kapitola 8

Existuje i mnoho dalších, ale těmi se nebudeme zabývat.

U třídících algoritmů využijeme poznatky z teorie složitosti. Časová složitost třídících algoritmů je důležitým kritériem pro výběr optimálního algoritmu pro daný problém. Velikost vstupu, v tomto případě pole, hraje významnou roli pro časovou složitost. Mnohé algoritmy vynikají rychlostí a jednoduchostí u malých polí, naopak se zvyšujícím se vstupem složitost rapidně roste.

Přehled mnoha používaných třídících algoritmů včetně ukázek a implementace v různých programovacích jazycích je pěkně popsán na webu itnetwork.cz[12].

7.2 Insertion sort

Vhodný především pro menší pole, u nichž je tento algoritmus velmi efektivní a jednoduchý pro pochopení i naprogramování. V případě menších polích je obecně považován za efektivnější než například Quick sort (kap. 8), naopak u velkých polí se projevuje jeho časová složitost $\mathbf{O}(n^2)$.

Principem je rozdělení vstupního pole na setříděnou a nesetříděnou oblast. Postupně vybírá prvky z nesetříděné části a vkládá je mezi prvky v setříděné části tak, aby zůstala setříděná. Od toho jeho název - vkládá prvek přesně tam, kam patří a nedělá další kroky navíc.

Inicializace spočívá v tom, že pouze první prvek pole se považuje do skupiny setříděných prvků. Druhý prvek pole se uloží do pomocné paměti. Nyní vytvoříme místo pro tento prvek v již setříděné části, kam ho poté vložíme (insertion). Postupně se posouvají prvky v setříděné části pole doprava

[12] https://www.itnetwork.cz/algoritmy/razeni

tak dlouho, dokud buď nenarazíme na prvek menší (nebo stejný) nebo na začátek pole. V těchto případech algoritmus končí a pole je celé setříděné.

Implementace je pomocí cyklu **for**. Princip algoritmu:

1. První prvek pole ponecháme na svém místě.

2. Druhý prvek porovnáme s prvním. Je-li menší, zařadíme ho na první místo a první prvek posuneme, jinak je ponecháme na místě.

3. Vezmeme třetí prvek a porovnáme jej s prvními dvěma prvky. Je-li menší než některý z nich, zařadíme jej na odpovídající pozici a následující prvky dle potřeby posuneme. Jinak je ponecháme na původních místech.

4. Stejně postupujeme v cyklu i s ostatními prvky v poli, dokud není zařazen poslední prvek.

7.3 Selection sort

Patří svým principem mezi nejjednodušší algoritmy na pochopení i na implementaci. Není však stabilní, což je jeho slabina. Hodí se zejména pro řazení malých polí a nevyžaduje pomocnou paměť, na rozdíl od insertion sort a dalších algoritmů.

Jeho myšlenka spočívá v nalezení minima, které se přesune na začátek pole (nebo můžeme hledat i maximum a to dávat na konec). V prvním kroku tedy nalezneme nejmenší prvek v poli a ten přesuneme na začátek pole. Postup se opakuje, opět se hledá minimum, případně maximum ze zbývajících prvků. Tedy v dalším kroku již nebudeme při hledání minima brát v potaz dříve nalezené minimum (maximum). Po dostatečném počtu kroků je pole seřazené.

Princip algoritmu:

1. V posloupnosti najdeme nejmenší prvek a vyměníme jej s prvkem na první pozici. Dojde k rozdělení posloupnosti na dvě části. Setříděná část obsahuje pouze jeden prvek, nesetříděná ostatních n-1 prvků.

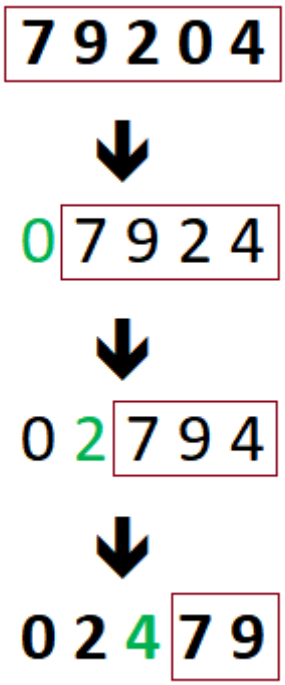
2. V nesetříděné části najdeme nejmenší prvek a vyměníme ho s prvním prvkem v nesetříděné části, čímž dojde k zařazení tohoto prvku do setříděné části.

3. Obsahuje-li nesetříděná část více než jeden prvek, cyklicky se pokračuje krokem 2, jinak je řazení hotovo.

Jednoduchý příklad na vysvětlení principu:

Vstup: Pole čísel 7 9 2 0 4

Výstup: Setříděné pole 0 2 4 7 9



Obrázek 7.1 Princip algoritmu Selection Sort na poli s 5 prvky

Σ

Tato kapitola je úvodem do problematiky řadících (třídících) algoritmů. Popisuje význam řazení a použití řadících algoritmů a proč je potřeba implementaci provádět podle složitosti. Některé algoritmy jsou vhodné pro malá, jiné pro velká pole. Jsou zde představeny dva základní algoritmy – Insertion sort a Selection sort, jejich stručný princip a výhody/nevýhody.

?

1. Stručně vysvětlete princip algoritmu Insertion Sort. Je vhodný pro malá pole?
2. Proč je Selection Sort vhodný pro řazení malých polí? Uveďte jeho základní princip.
3. Proč je časová složitost zásadním kritériem pro výběr vhodného řadícího algoritmu k implementaci vzhledem k velikosti pole?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 8

Řazení, řadící algoritmy – bubble sort, quick sort

Po prostudování kapitoly budete umět:

- vysvětlit princip algoritmu bubble sort
- vysvětlit princip algoritmu quick sort
- popsat rozdíly mezi algoritmy řazení vzhledem k jejich složitosti

Klíčová slova:

Řazení, řadící algoritmy, třídění, sorting, bubble sort, quick sort.

8.1 Bubble sort

V mnoha případech není tento algoritmus příliš efektivní a tím je pomalý, časová složitost je $\boldsymbol{O}(n^2)$. Algoritmus probíhá v tzv. vlnách, přičemž při každé vlně propadne "nejtěžší" prvek na konec (nebo nejlehčí „vybublá“ nahoru, záleží na konkrétní implementaci). Vlna srovnává vzájemně dvojice sousedních prvků a v případě, že je levý prvek větší než pravý, prvky prohodí. Algoritmus je však stabilní.

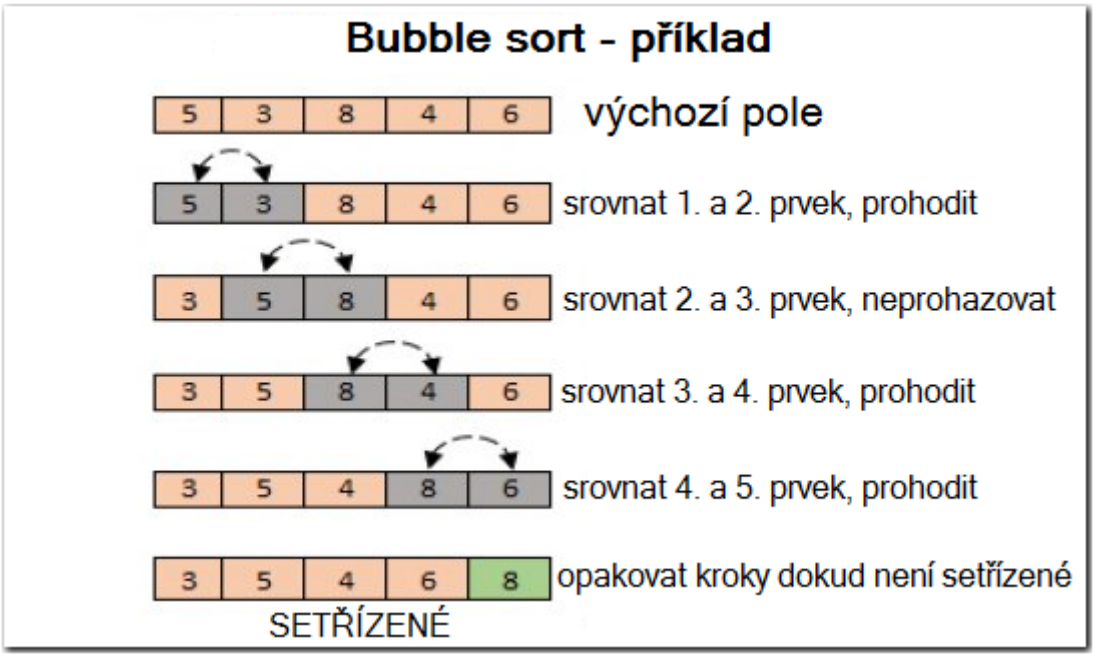
Princip algoritmu:

1. Vstupní pole rozdělíme na dvě části, setříděnou a nesetříděnou. Setříděná část je prázdná.

2. Postupně porovnáme všechny sousední prvky v nesetříděné části, a pokud nejsou v požadovaném pořadí, prohodíme je.

3. Krok 2 se v cyklu opakuje tak dlouho, dokud nesetříděná část obsahuje více než jeden prvek. Jinak algoritmus končí.

Podívejte se na jednoduchý příklad, jak jednotlivé kroky algoritmu fungují na poli o 5 prvcích:



Obrázek 8.1 Příklad setřízení pětiprvkového pole pomocí Bubble sort algoritmu

8.2 Quick sort

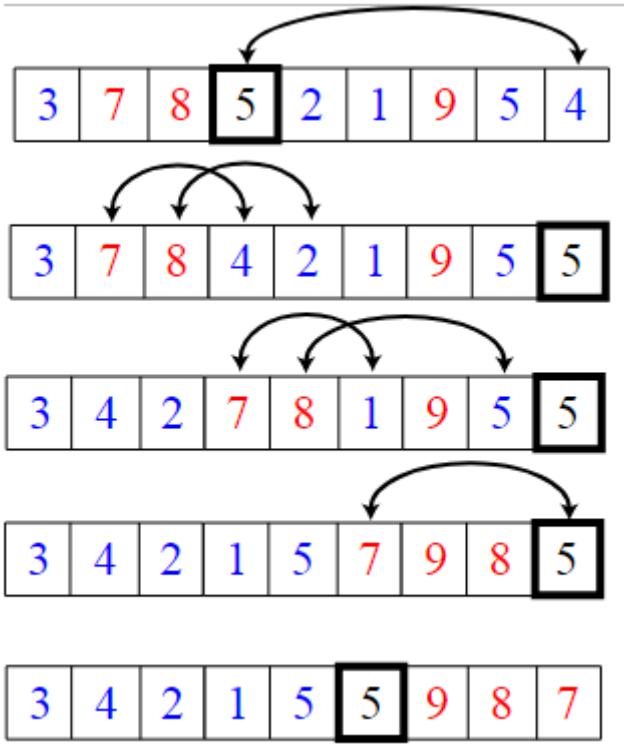
Název tohoto třídícího algoritmu napovídá, že bude rychlý. A opravdu je tomu tak, patří mezi nejrychlejší. V praxi se používá velmi často. Princip je založen na metodě Rozděl a panuj (viz kap. 10). Je však o něco složitější na pochopení i na implementaci, odvděčí se však velkou rychlostí i na velkých polích. Asymptotická časová složitost je lineárně-logaritmická $\boldsymbol{O}(n \log n)$. Myšlenka algoritmu vychází ze skutečnosti, že nejefektivnější jsou výměny prvků v poli na velké vzdálenosti. Jestliže např. vezmeme pole s n prvky seřazenými v obráceném pořadí, lze je utřídit pomocí pouhých $n/2$ výměn.

Stěžejní roli hraje prvek pole, který se nazývá ***pivot***. Ten zvolíme na začátku náhodně. Cílem algoritmu je srovnávat prvky větší a menší než pivot a dosáhneme toho, že na jedné straně pole jsou prvky menší než pivot, poté je pivot a na druhé straně větší než pivot. Představme si velmi malé vstupné pole lichých čísel: 3 9 7 5 1. ideální výběr pivot je prvek hodnotou 5 (je to jeho medián). Poté bude 1 3 **5** 7 9 a pivot je přesně uprostřed (tučně označený).

Celková rychlost a optimálnost algoritmu závisí na vhodné volbě pivota. Pokud zvolíme pivot, který je maximem nebo minimem v poli, pak složitost je $\boldsymbol{O}(n^2)$, kdežto při správné volbě pivota se dostáváme na O(n*log2n), čímž je Quick sort rozhodně nejrychlejší. Quick sort se používá především pro řazení velkých polí, u malých se často nedaří správná volba pivota vzhledem k malému rozsahu „správné trefy“. Velmi efektivní bude tento algoritmus například v případě seřazení 5000 hodnot náhodně generovaných čísel normálního rozdělení v intervalu (0; 1). Zvolíme-li například pivot = 0,5, algoritmus bude rychlý. Šance na vhodnou volbu pivota je právě u velkých polí, díky čemuž je algoritmus optimálnější než u malých polí. V pseudokódu můžeme princip algoritmu Quicksort zapsat takto:

```
procedure quicksort(List hodnoty)
 if hodnoty.size <= 1 then
  return hodnoty
pivot = náhodný prvek z values
  mensi = { prvky větší než pivot }
  pivot = { pivot }
  vetsi = { prvky menší než pivot }
return quicksort(mensi) + pivot + quicksort(vetsi)
```

Následující obrázek[13] ukazuje princip třídění s volbou pivotu a rozdělení na 2 subpole.



Obrázek 8.2 Princip algoritmu Quick sort s volbou pivotu = 5

Princip je tedy založen na tvorbě 2 subpolí – prvky menší než pivot a větší než pivot. Quick sort se dá naprogramovat rekurzivně i bez použití rekurze.

Třídící algoritmus Quick Sort je dobrou ukázkou použití rekurze a zároveň principu metody „Rozděl a panuj“:

1. Rozdělení vstupního pole na dvě podpole pomocí pivotu – aplikace metody Rozděl a panuj
2. Rekurzivní přeskládávání prvků do chvíle, kdy je pouze jedno pole.

8.3 Stručné shrnutí řadících algoritmů

Shrňme si základní poznatky o čtyřech řadících algoritmech, které jsme poznali v kapitolách 7 a 8.

[13] https://upload.wikimedia.org/wikipedia/commons/8/84/Partition_example.svg

- **Insertion sort** - vhodná pro malá pole, kde je rychlejší než Quick sort, vyžaduje pomocnou paměť
- **Selection sort** - nevyžaduje pomocnou paměť, pro malá pole rychlý, jednoduchý na implementaci
- **Bubble sort** - obecně nepříliš efektivní, je však stabilní
- **Quick sort** - velmi rychlý pro velká pole, efektivní, lineární asymptotická složitost, složitější implementace

Efektivita každého algoritmu (i dalších, zde neprobíraných), vždy spočívá nejen ve velikosti pole, ale rovněž na způsobu implementace, včetně například možnosti paralelního zpracování (viz kap. 12).

Algoritmy řazení patří mezi základní algoritmické dovednosti. Správnou volbou řadícího algoritmu dosáhneme optimální časové i prostorové složitosti.

Kapitola 8 se soustředila na vysvětlení dalších dvou řadících algoritmů – Bubble sort Quick sort. Stručně byl představen jejich princip a výhody i nevýhody. Na závěr kapitoly je stručné shrnutí základních poznatků o probíraných řadících algoritmech z kapitol 7 a 8.

1. Vysvětlete stručně princip Bubble sort algoritmu.
2. Je Quick sort efektivnější u malých nebo velkých polí a proč?
3. Kdy bude pivot mediánem prvků v poli pro Quick sort?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 9

Strukturovaný přístup, podprogramy, modulární programy

Po prostudování kapitoly budete umět:

- vysvětlit princip strukturovaného přístupu
- chápat k čemu slouží podprogramy jako ucelené části algoritmu
- popsat princip a přínos modulárního programování

Klíčová slova:

Strukturovaný přístup, podprogram, modulární programování

9.1 Strukturovaný přístup

Strukturovaný přístup se opírá o myšlenku rozčlenění celého programu na jednotlivé struktury, resp. celky. Existuje více definic, jak lze definovat strukturovaný přístup k programování a vůbec k návrhu algoritmů. Mezi strukturované programovací jazyky patří např. Pascal či C. Oproti tomu jsou objektově orientované jazyky využívající OOP (viz kap. 1).

Obecně lze uvést tyto podmínky, které pro strukturovaný přístup musejí být splněny:

- modul návrh algoritmu metodou shora-dolů (viz kap. 1), rozdělení na části
- používat pouze tři základní řídící struktury – sekvence, selekce (IF) a opakování
- přesnou definici datových struktur odložit až do fáze, než ji v algoritmu použiji
- program se skládá z modulů, podprogramů, funkcí a procedur

Strukturovaný přístup má samozřejmě v dnešní době spoustu limitací oproti OOP, např. neexistence návrhových vzorů[14]. Strukturovaný přístup je však jednoduchý, poměrně přímočarý a tím snadno pochopitelný jako základ pro algoritmické myšlení. OOP tkví ve zcela jiných základech.

Princip tedy tkví ve vytvoření struktur programu, které jsou co nejvíce nezávislé mezi sebou. Podstata tkví v rozdělení celého algoritmu na jednotlivé části a ty řeší funkce a procedury.

9.2 Podprogramy

Podobně jako si deklarujeme proměnné a konstanty pro jejich užití v různých krocích programu nebo voláme stejné funkce, obdobným způsobem můžeme v programu používat celé části kódu, které nazýváme ***podprogramy***. Představme si situaci, že je třeba v algoritmu několikrát použít poměrně náročnou část výpočtu se značným počtem kroků. Při programování bychom bez využití podprogramu museli pokaždé stejný kus kódu kopírovat. Podprogram lze kdekoli zavolat podobně jako funkci a netřeba tak kopírovat velké části kódu. Nejenže to ušetří čas, ale kód programu bude přehlednější. Podprogram zanořen, de facto je podmnožinou celého programu. Podprogram může obsahovat další zanořené podprogramy.

Rozlišujeme dva druhy podprogramů, vycházející z principů strukturovaného programování – funkce a procedury. Obojí znamená posloupnost kroků (instrukcí), které potřebujeme v algoritmu opakovaně použít.

[14] https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=5855

9.3 Modulární programování

Myšlenka ***modulárního programování*** spočívá v principu modulární architektury. Jako příklad můžeme uvést například operační systém Windows Server, v němž nastavíme jednotlivé role (moduly), které bude systém plnit, například tiskový, webový nebo aplikační server. Podobně například program jako je Adobe Photoshop, v němž existují stovky modulů, které lze kdykoli vypnout. Modul je tedy něco samostatného, co lze deaktivovat nebo z programu odstranit a nemá vliv na program jako celek.

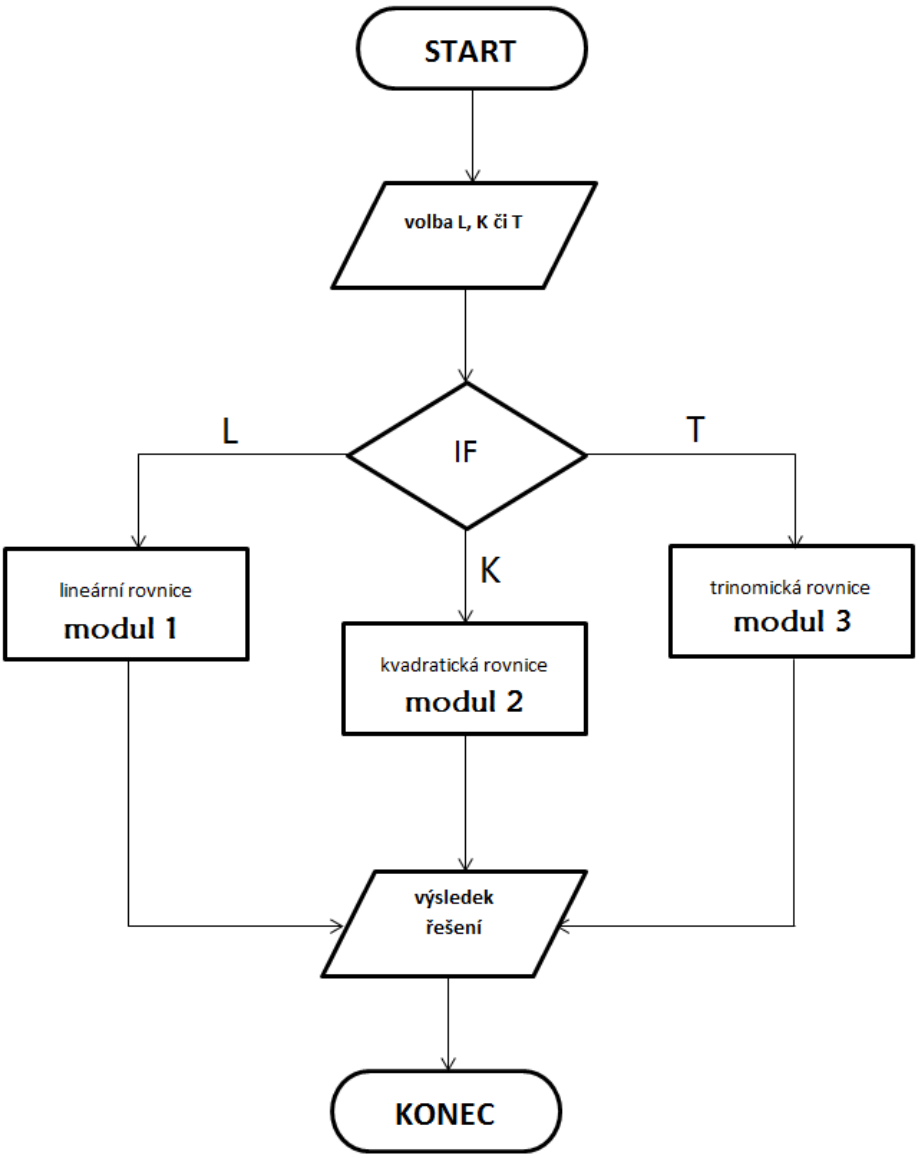
Na tomto chápání funguje model modulárního programování. Je založen na metodě rozděl a panuj (viz kap. 10) a principu návrhu algoritmu shora-dolů (viz kap. 1). Návrh algoritmu rozdělíme na nezávislé celky, moduly, z nichž se skládá celý program. Důležitým prvkem je zjemňování, tedy rozdělení programu na menší celky, z nichž se program skládá. Vyžadováno je, aby si jednotlivé moduly předávaly co nejméně společných dat a fungovaly de facto samostatně.

Uveďme si příklad. Mějme program na řešení lineárních, kvadratických a trinomických rovnic. Tento celý program rozdělíme na 3 moduly:

- modul pro řešení lineární rovnice
- modul pro řešení kvadratické rovnice (algoritmus stručně viz kap. 1)
- modul pro řešení trinomické rovnice

Celý program se bude skládat ze tří nezávislých modulů, které lze definovat jako podprogramy (kap. 9.2).

Princip práce by mohl být navržen tímto algoritmem popsaným vývojovým diagramem:



Obrázek 9.1 Princip modulárního návrhu algoritmu pro řešení tří typů rovnic

Což znamená z tohoto případu, že program se větví dle zadání L, K nebo T z klávesnice a dle toho se aktivuje jeden z modulů. Stěžejní je zde to, že program by fungoval, i kdybychom kterýkoli modul odstranili. Modul nebude dostupný, ale ostatní zůstanou plně funkční. To je princip jejich použití – program je funkční, i když některý modul odstraníme.

Například kdybychom odstranili modul 1 pro výpočet lineární rovnice, při stisku L se vypíše chybová hláška, že modul není dostupný, přičemž ostatní 2 bude nadále zcela funkční. Naopak, mohli bychom do programu přidat modul pro výpočet kubické rovnice a obecně reciprokých rovnic.

Modulární programování je de facto mezistupeň mezi ryze strukturovaným (procedurálním) přístupem a OOP paradigmatem.

9.4 Objektově-orientované paradigma (OOP)

Tato publikace není zaměřena na OOP, proto zcela okrajově ohledně OOP. Je to zcela odlišný způsob návrhu algoritmů a myšlení programátora. Přestože pro programátory, kteří jsou zvyklí na strukturovaný přístup například z jazyka C, je přechod na OOP nepochopitelný. Ve skutečnosti se OOP mnohem více blíží způsobu myšlení člověka.

Základem je tzv. ***objekt***, například člověk, zvíře, databáze, apod. Něco reálného. Každý objekt má své atributy a metody. Atributy jsou například věk, barva, počet nohou, relační model a metody jsou schopnosti objektu – u člověka to může být chodit, u zvířete lovit, u databáze přidat záznam do tabulky.

V OOP dále definují tzv. ***třídy***. Sdružují objekty do celků se stejnými atributy. Například třída *Studenti* bude obsahovat objekty *Lucka*, *Adam*, *Petr*, *Ivo*.

Pokud se chcete o OOP paradigmatu dovědět více, začít můžete například na webu ITnetwork.cz[15].

Tato kapitola se věnuje problematice členění algoritmu na části. Jednotlivé části algoritmu, které využíváme na více místech, je možné oddělit jako samostatné celky, které se snažíme co nejméně vázat. Podprogram může obsahovat několik funkcí a procedur. Modulární programování je postavené na návrhu modulů, které jsou schopny pracovat samostatně a lze je v programu přidávat a mazat, přičemž celý program není nijak narušen, jen bude chybět nebo přidána určitá funkcionalita.

1. Co je strukturované programování? Může v podprogramu být další podprogram?
2. Stručně vysvětlete princip modulárního programování.
3. Zkuste navrhnout modulární řešení pro výpočet základních goniometrických funkcí úhlu.

[15] https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 10

Metoda „Rozděl a panuj“, rekurze

Po prostudování kapitoly budete umět:

- vysvětlit princip rozdělení algoritmu metodou Rozděl a panuj
- používat rekurzi
- navrhnout metodou Rozděl a panuj jednoduché algoritmy, pochopit metodu pro třídění

Klíčová slova:

Rozděluj, panuj, rekurze, iterace

10.1 Metoda „Rozděl a panuj“

Metoda ***Rozděl a panuj*** patří k často používaným metodám v algoritmizaci sloužící k rozdělení algoritmu na dílčí části. V tomto případě jde o rozdělení algoritmu na stejné dílčí úlohy. Často se tato metoda uvádí v latinském znění „divide et impera“ či anglicky „divide and conquer“. Principiálně jde o nalezení způsobu, jak algoritmus rozdělit na stejné části, které poté spojíme dohromady. Obecně jsme se o problematice rozdělení na podúlohy zmínili v kapitole 1. Podstata spočívá dále v tom, že pokud rozdělíme výchozí problém na menší celky, opět tyto menší celky lze rozdělit, až se dostaneme na de facto atomární úlohy, které samy o sobě mají triviální a zcela jasné řešení. Trochu to připomíná matematický algoritmus půlení intervalů[16]. Metoda Rozděl a panuj se často uplatňuje u třídících algoritmů, viz kap. 7 a 8.

Metoda je založena na principu rekurze. Fáze „rozděl“ - vstupní problém rozdělíme na *n* stejně řešitelných podúloh a ty opět lze rozdělit dále na podúlohy, až se dostaneme k triviálnímu řešení úlohy, kterou umíme jednoduše vyřešit. Poté následuje fáze „panuj“ a postupně skládáme dílčí řešení do celku, tedy původní úlohy.

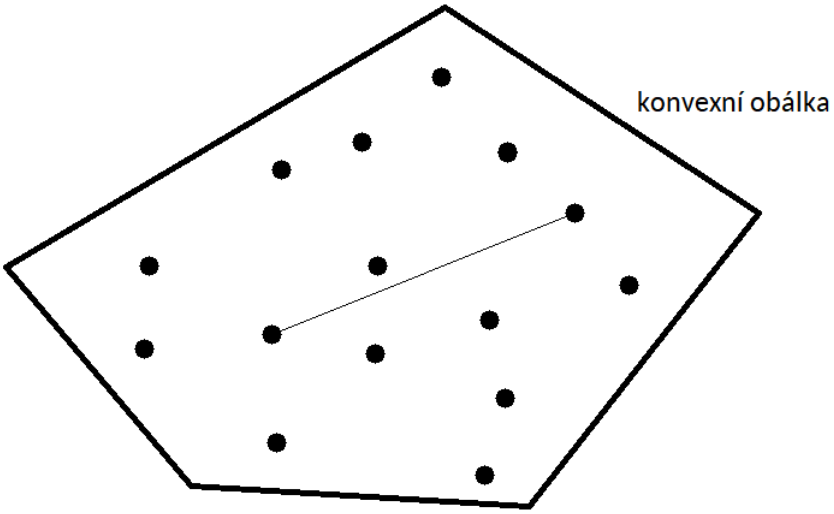
Popišme si tuto metodu matematickým myšlením na množinách. Mějme množinu **S** jako řešení problému (Solution). Množinu **S** rozdělíme na konečný počet disjunktních (nemají žádný společný prvek) podmnožin $\mathbf{D_1}$, $\mathbf{D_2}$,…, $\mathbf{D_n}$ (Divide). Vyřešíme problém každé podmnožiny a spojením dílčích výsledků dostáváme celou množinu **S**.

Praktickým příkladem pro využití metody „Rozděl a panuj“ je například problém nalezení konvexní obálky (Convex Hull) množiny bodů. Problém konvexní obálky je nalezení minimálního ohraničení množiny bodů, přičemž platí:

- žádný bod nesmí být mimo obálku
- spojíme-li 2 zcela libovolné body, vznikne úsečka a ta musí být celá v konvexní obálce

Názorně a zjednodušeně je možno princip konvexní obálky ilustrovat pomocí následujícího obrázku:

16 http://math.fce.vutbr.cz/vyuka/matematika/numericke_metody/node3.html



Obrázek 10.1 Konvexní obálka množiny bodů

Použití metody Rozděl a panuj s algoritmem „procházka“.

Problém nad množinou bodů **B** rozdělíme na dvě podmnožiny $\mathbf{B_1}$ a B_2 o stejné velikosti, tyto podmnožiny jsou zpracovávány samostatně. Obě řešení poté spojíme horní a dolní tečnou. Body, které zůstanou uvnitř, jsou z obálky vyřazeny. Tím vznikne celé řešení.

Toto dělení se stále opakuje, dokud nenarazíme na triviální řešení o třech bodech. Po nalezení triviálních obálek (triviální obálka je trojúhelník ve 2D), je třeba tyto obálky spojit. Spojování se provádí pomocí nalezení dolních a horních tečen. Využije se k tomu algoritmus nalezení tečen, který se nazývá „procházka“.

Naše dvě malé obálky nazveme O_1 a O_2, obě obálky mají extrémní body ni a mi. Nyní potřebujeme k bodu n najít takový bod *m*, aby jejich spojnice byla dolní tečnou obálky O_1. Analogicky pak hledáme z bodu m bod n, jejichž spojnice tvoří dolní tečnu O_2. Postup opakujeme tak dlouho, dokud nenajdeme dolní tečnu, která je dolní tečnou O_1 i O_2.

10.2 Princip a využití rekurze

Než přejdeme k algoritmům, uveďme si případy rekurze v matematice a v běžném životě. V matematice se lze rekurzí setkat například u definice oboru přirozených čísel. Nechť 1 je z **N** a libovolné číslo *n* je z **N**, pak platí, že každé číslo *n*+1 je rovněž z **N**. Rekurzivní funkce jsou takové, že volají samy sebe. Příkladem nechť je výpočet faktoriálu (viz kap. 1). V praktickém životě se s rekurzí setkáváme

a ani o tom možná nevíme. Například, už jste někdy viděli obrázek, na němž je tentýž obrázek a vnořuje se do sebe? Ano, i to je příklad rekurze.

Pro začátek řekněme to, že rekurze umožňuje v řadě případů nahradit konstrukci cyklů, viz kap. 4. V některých programovacích jazycích, jako je například funkcionální Lisp, dokonce neexistuje přímá konstrukce cyklu **for** či **while**. Proto je rekurze zde nevyhnutelná. Ale i v mnoha běžných jazycích pomocí rekurze můžeme obejít použití cyklů.

V oblasti programování rekurzí chápeme opakované použití programové konstrukce při řešení téže úlohy. Na rozdíl od cyklu se u rekurze vyskytuje použití téže konstrukce uvnitř konstrukce samotné. Příklad použití jsou například podprogramy (viz kap. 9) či definování datové struktury neboť struktura obsahuje prvek, který má stejnou podobu jako definovaná struktura. Příkladem jsou například právě binární stromy, které nyní rozebíráme.

Hlavním smyslem použití rekurze je nahrazení cyklů rekurzivním voláním funkce.

Součástí rekurzivní funkce musí být nutně i ukončující podmínka, která definuje, kdy se rekurze ukončí. Jinak by rekurze teoreticky mohla trvat nekonečně dlouho.

Rekurze je způsob deklarování podprogramu, při kterém podprogram ve svém těle volá sám sebe. Typický, často uváděným příklad pro rekurzi je výpočet faktoriálu. S tímto algoritmem jsme se již několikrát setkali. Rekurzivní je zde volání součinu. Nechť máme spočítat n!, tak v každém kroku je volána tatáž funkce pro násobení n×(n-1)(n-2)…1.

Úloha 1: Mějme umocnit číslo *x* na *n*-tou celočíselnou mocninu, například x^5.

Vstup: Zadáme číslo *x* a zadáme mocninu *n*

Výstup: x^n

Princip rekurzivního algoritmu:

1. Zadání čísla *x* a mocniny *n*

2. Přečtení vstupů

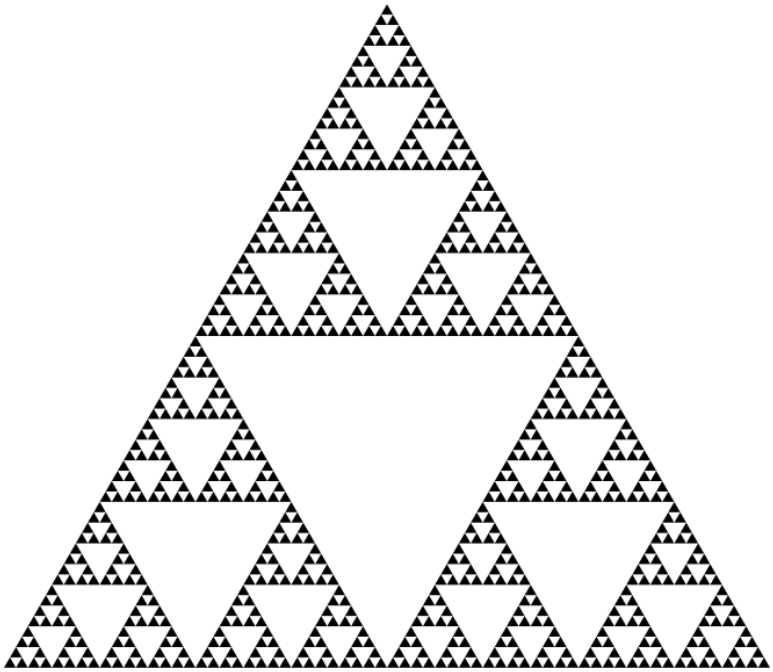
3. Rekurzivní volání násobení čísla *x* tolikrát, kolik se rovná *n*

4. Výstup x^n

Rekurzivním podprogramem je krok 3., v němž se volá stále funkce součinu čísla *x* tak dlouho, dokud se n nerovná zadanému vstupu. V našem případě bude součin $x \cdot x \cdot x \cdot x \cdot x$.

Dalším příkladem rekurze může být výpis přirozených od 1 do *n* nebo nalezení největšího společného dělitele (NSD) dvou čísel dle Euklidova algoritmu[17]., převody mezi různými polyadickými soustavami nebo hledání maxima v poli.

S rekurzí se setkáváme u binárních stromů, viz následující kapitola. Mezi hrami se s rekurzí setkáváme u známé logické hry Hanojské věže. Výborným příkladem je také tzv. ***želví grafika***[18] založena na rekurzivním vykreslování a rovněž ***fraktály***[19], např. Sierpińského trojúhelník, viz níže. Tento fraktál je založen na rekurzivním vykreslování rovnostranných trojúhelníků. Každé vykreslení trojúhelníku na nižší (menší) úrovni je stále tou stejnou funkcí, která se tak rekurzivně volá. Na obrázku je vidět 7. rekurze.



Obrázek 10.2 Sierpińského trojúhelník jako příklad rekurze v počítačové grafice

[17] http://www.karlin.mff.cuni.cz/~zemlicka/10-11/uda_CRT.pdf

[18] https://www.root.cz/clanky/zelvi-grafika-a-rekurze/

[19] http://www.cs.vsb.cz/dvorsky/Download/ALGI/Slides/Lecture07.pdf

10.3 Rekurze vs. iterace

K Tyto dva pojmy se zejména začínajícím programátorům často pletou a považují je za totéž. Jsou to však zcela rozdílné pojmy. Rekurzi jsme definovali jako proces, při němž funkce volá samu sebe, se říká rekurze a takové funkce jsou rekurzivní.

Iterace v matematice znamená v matematice opakovaný, postupně se zpřesňující výpočet určité hodnoty, přičemž se přibližná mezihodnota použije jako výchozí k výpočtu hodnoty následné, která je přesnější. Na tomto principu pracuje například Newtonova metoda (rovněž nazývána metoda tečen) k řešení mnohých algebraických a transcendentních rovnice[20]. V oblasti algoritmů je za iteraci považován taktéž opakovaný výpočet, ale provedený postupně pro jednotlivé členy řady argumentů.

Metoda „Rozděl a panuj“ patří k často používaným přístupům při návrhu algoritmů. V principu jde o rozdělení úlohy na podúlohy, které lze triviálně vyřešit. Následným spojením dostaneme celé řešení. Rekurze je stěžejním pojmem mezi programátory a umožňuje efektivně nahradit cykly v algoritmech. Rekurze je založena na principu volání funkce obsahující sama sebe. Příkladem využití rekurze je výpočet faktoriálu přirozeného čísla, hledání NSD nebo výpočet *n*-té celočíselné mocniny. Mnohé algoritmy, jako například třídící Quick sort, lze navrhnout rekurzivně i nerekurzivně.

1. K čemu slouží metoda Rozděl a panuj?
2. Jednoduše vysvětlete smysl rekurze a kde je možné ji využít.
3. Proč je výpočet faktoriálu n! typickým příkladem pro využití rekurze?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

[20] http://physics.ujep.cz/~jskvor/NME/DalsiSkripta/Numerika.pdf

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 11

Binární stromy, hashing

Po prostudování kapitoly budete umět:

- vysvětlit princip binárního stromu
- ukázat použití binárního stromu na jednoduchém příkladu
- vysvětlit základní princip hashingu a hashovacích funkcí a uvést některé základní

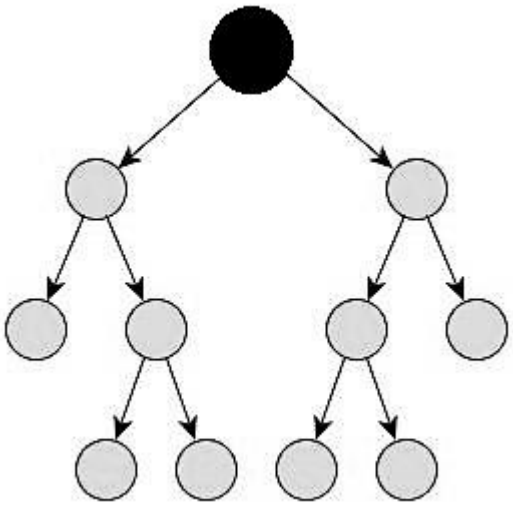
Klíčová slova:

Binární strom, hashing, hashovací funkce, MD5, otisk, hash

11.1 Binární stromy

Binární strom je pojmem z teorie grafů[21] a zároveň patří mezi nelineární datové struktury, oproti lineárním jako například pole (viz kap. 6). Je tvořen uzly, z nichž každý má maximálně 2 potomky. Jediný uzel je kořenový a ten nemá žádné „rodiče“, tedy nejsou nad ním žádné uzly.

V teorii grafů je pojem strom definován následovně: Strom je souvislý graf neobsahující cyklus. Což znamená, že mezi každými dvěma vrcholy existuje cesta a jelikož je strom souvislým grafem, pak nenastává situace více cest, protože neexistuje možnost cyklu. Obecné příklady stromů jsou na obrázku níže. Podívejme se, jak takový strom vypadá[22]:



Obrázek 11.1 Příklad stromu jako grafové struktury

Definujme dále tyto pojmy:

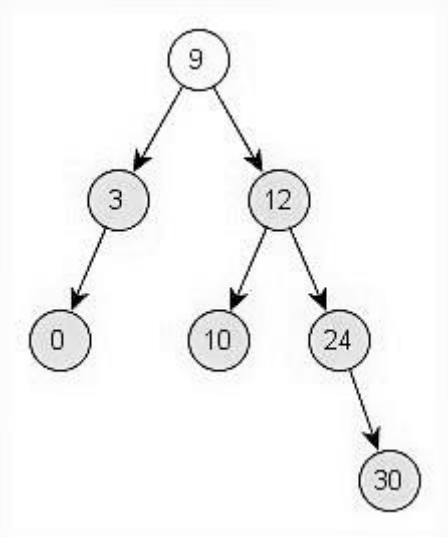
- **kořen stromu** – nejvrchnější uzel, který nemá rodiče
- **hloubka stromu** – délka cesty od kořene k danému uzlu (počet uzlů mezi nimi)
- **výška stromu** – největší hloubka libovolného uzlu

21 http://teorie-grafu.cz/zakladni-pojmy/stromy.php

22 http://www.linuxsoft.cz/img/c_plus_plus_algoritmy/example1.jpg

11.1.1 Binární vyhledávací stromy

Binární vyhledávací stromy jsou konkrétním příkladem stromu definovaného v teorii grafů a v algoritmizaci, resp. informatice slouží coby způsob uložení dat. Jsou speciálním případem obecných binárních stromů[23]. Každý uzel je ohodnocen klíčem, tedy nějakou číselnou hodnotou. Opět názorný příklad binárního vyhledávacího stromu[24]:



Obrázek 11.2 Příklad binárního vyhledávacího stromu s ohodnocenými uzly

Kořen stromu je 9. Dále jsme přidali číslo 12, které je větší než 9, proto je číslo 12 na pravé straně pod číslem 9. Následovalo číslo 24, které je větší než 9, je rovněž větší než 12 a proto je číslo 24 pravým potomkem čísla 12. Číslo 3 je menší než 9, je tedy levým potomkem čísla 9. Tímto způsobem jsme do stromu uložili i další zbylá čísla. Je tedy patrné, že při přidávání prvků do binárního vyhledávacího stromu začínáme od kořene a postupujeme dle toho, zda je číslo menší nebo větší.

Patrně se ptáte, k čemu je to v praxi užitečné pro algoritmy užitečné, kde se s binárními vyhledávacími stromy setkáte. Patrně nejzákladnější použití mají binární vyhledávací stromy, jak napovídá název, pro vyhledávání.

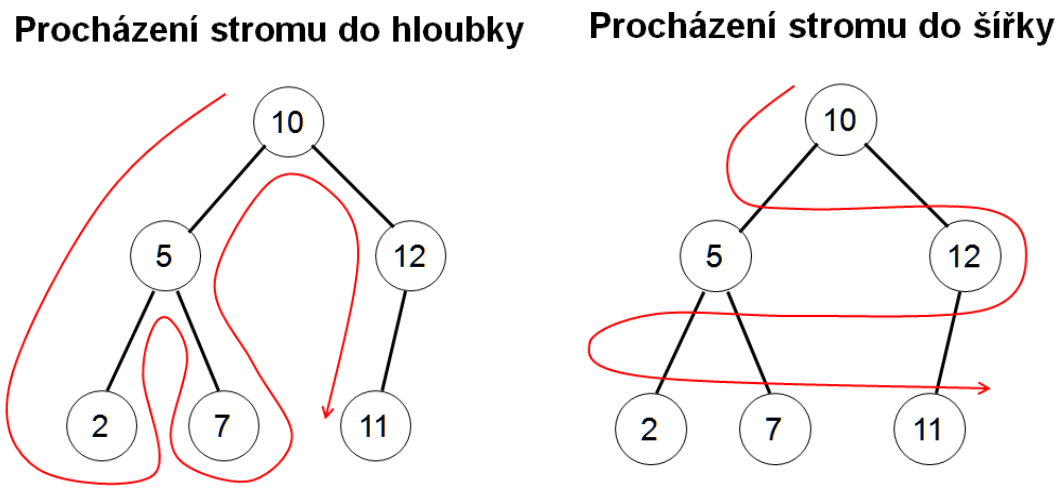
Máte v textovém souboru uloženo 1 000 000 čísel. Mezi těmito čísly je třeba nějaké vyhledat. Někoho by napadlo, že si čísla uloží do pole a poté bude pole prohledávat. Pokud by prohledával metodou "pokus omyl", čili by procházel prvky postupně za sebou, mohl by mít tu smůlu, že jeho číslo by bylo úplně až na konci pole, tudíž by potřeboval 1 000 000 porovnání, což může být časově složitá operace (viz kap. 5 o složitosti). Metoda půlení intervalu by byla efektivnější, jenže aby fungovala,

23 http://www.linuxsoft.cz/article.php?id_article=1772

24 http://www.linuxsoft.cz/img/c_plus_plus_algoritmy/example2.jpg

pole by muselo být nejprve setříděno, což též nějakou dobu potrvá. V případě, že bychom ale čísla uložili do stromu, maximální počet porovnávání by byl roven výšce stromu, která by v 99 % případů nebyla 1 000 000. Binární vyhledávací stromy tedy slouží k jakési organizaci dat s jejich rychlejším vyhledáváním. Jde tedy o záležitost optimalizace algoritmu oproti použití běžných struktur, jako jsou pole.

Existují dva základní způsoby procházení stromu – do hloubky a do šířky. Rozdíl názorně ilustrují následující 2 obrázky[25]:



Obrázek 11.3 Princip průchodu stromem při prohledávání do hloubky a do šířky

Vyhledat prvek v binárním stromu je velmi snadné. Začneme tím, že srovnáme hledané číslo s číslem kořenu. Mohou nastat tři možnosti:

- číslo, které obsahuje kořen, je rovno hledanému číslu - hledání úspěšné
- číslo, které hledáme, je větší než kořen - pak pokračujeme rekurzivně na pravém potomkovi
- číslo, které hledáme, je menší než kořen - pak pokračujeme rekurzivně na levém potomkovi

Tímto se rekurzivně prohledá celý binární strom, až k hledanému číslu.

Dalším příkladem je nalezení nejmenšího či největšího prvku (čísla) ve stromu. Pokud se podíváte na organizaci prvků v binárním vyhledávacím stromu, zjistíte, že najít nejmenší, resp. největší prvek je snadné - nachází se úplně vlevo (nejmenší) či úplně vpravo (největší). Stačí tedy od kořene postupovat směrem doleva nebo doprava, dokud nenarazíme na uzel, který již žádného levého, resp. pravého potomka nemá.

[25] https://www.fd.cvut.cz/personal/xfabera/PRG2/cviceni3/abstr_dat_typy.ppt

Nechť je v binárním stromu uzel *U*, který obsahuje hodnotu *x*. Pak pro uspořádání dat v binárním stromu platí, pak všechny uzly v levém podstromu uzlu *U* mají hodnoty menší než *x* a uzly v pravém podstromu mají hodnoty větší než *x*.

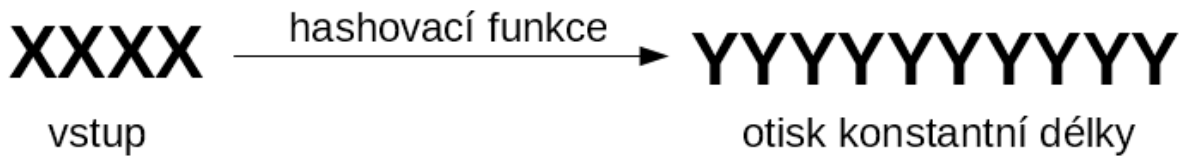
Tato definice platí i rekurzivně, tedy pro celý strom. U stromů se tedy rekurzivní algoritmy uplatňují při vkládání prvků, při vyhledávání v uspořádaném stromu a při jejich procházení.

11.2 Hashing, hashovací funkce

Velmi stručně řečeno, hashing je algoritmus využívající ***hashovací funkci***, která zajistí téměř nemožné zjištění původní hodnoty. Hashovacích funkcí je mnoho, k jedněm z nejznámějších patří **MD5** nebo **SHA** (Secure Hash Algorithm; různé verze SHA-1, SHA-2, SHA-3, SHA-256, SHA-512 a další).

Hashovací funkce je obecně složitá matematická funkce převádějící vstupní řetězec libovolné délky (zpráva, datový soubor) na řetězec konstantní délky a vytváří tzv. ***otisk vstupního řetězce***. Tento výstupní otisk se označuje jako výtah, ***hash***, ***fingerprint*** nebo miniatura a je závislý na všech bitech vstupního řetězce. Tyto funkce slouží ke kontrole integrity dat, k porovnání dvojice zpráv, k vyhledávání, indexování a využívají se rovněž k implementaci digitálních podpisů. Využití hashovacích algoritmů je velmi rozmanité.

Délkou výstupního otisku se určuje složitost a spolehlivost algoritmu. Délka hashe může být například 64, 128, 256, 324 nebo 512 bitů i více. Princip hashingu lze opravdu zjednodušeně vyjádřit následujícím obrázkem.



Obrázek 11.4 Zjednodušený princip tvorby otisku (hashe)

Je to však skutečně velmi zjednodušené. V praxi jsou hashovací funkce velmi složité a pro jejich důkladné pochopení lze mít vyšší znalosti programování a matematiky. Podrobný popis lze najít například na webu mendelu.cz[26].

Matematicky lze vyjádřit hashovací funkci takto: Nechť V_B je vstupní posloupnost bitů a H je hash. Pak hashovací funkce *hf* je

$$hf = V_B \rightarrow H.$$

Ještě je nutno zmínit důležitý pojem – ***kolize***. Není to nic jiného, než že pro různé dva vstupní řetězce existuje stejný hash, v tomto případě je hashovací funkce považována za nedostatečně bezpečnou.

11.2.1 Hashovací funkce MD5

MD5 (Message-Digest 5) patří k jednodušším, avšak stále hojně používaným hashovacím funkcím. Výstupní hash má délku 128 bitů, což odpovídá 32 znakům. Ať je vstupní řetězec jakkoli dlouhý, třeba celý odstavec textu, výstupem bude vždy pouze 128bitový řetězec. Příkladem je ukázka zakódovaného textu ahoj pomocí MD5:

79C2B46CE2594ECBCB5B73E928345492 = 32 znaků, což je 128 bitů

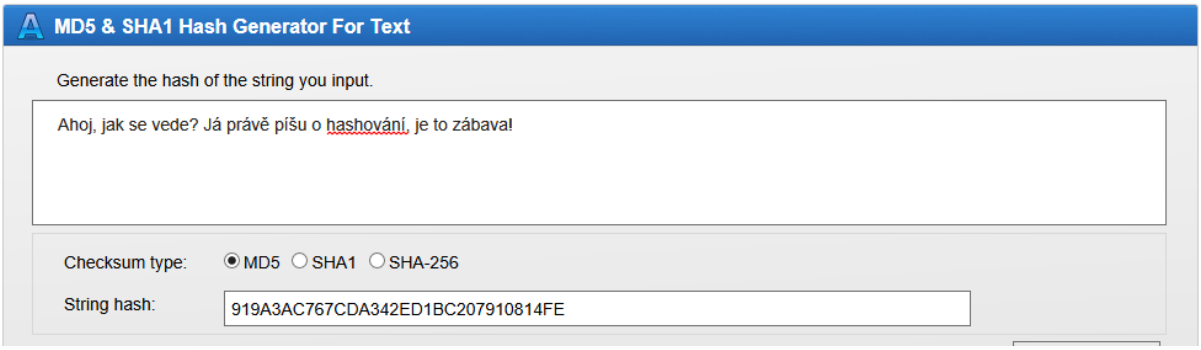
MD5 hash má na výstupu 32 hexadecimálních znaků, tedy číslice 0 až 9 a písmena A až F (šestnáctková číselná soustava).

Pro detailní matematický popis algoritmu hashovaní funkce odkazujeme na doporučenou literaturu, případně na zmíněný odkaz v předchozí podkapitole.

V praxi již v dnešní době není funkce MD5 považována za bezpečnou, prolomena byla útokem v roce 2005. Je však základní funkcí a stále se využívá například ke kontrole integrity souboru (originální vs. jiný soubor a shoda MD5 otisku), tzv. checksum. Na Internetu lze najít mnoho online nástrojů pro ověření otisku MD5 i jiných funkcí pro soubory.

Online si můžete vyzkoušet, jak funguje v praxi algoritmus MD5 a SHA, např. na webu onlinemd5.com.

[26] https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=7029



Obrázek 11.5 Ukázka z online generátoru hashe MD5 a SHA ze vstupního textu

V praxi uvidíte sami, že ať zadáte jakkoli dlouhý text, výstupem bude vždy pouze hash s délkou 32 hexadecimálních znaků. O něco silnější je pak funkce SHA-1 s délkou otisku 160 bitů, sofistikovanější jsou další verze SHA, například SHA-256 s délkou 256 bitů nebo SHA-512 s délkou hashe 512 bitů.

Problematika hashingu, hashovacích funkcí a celkově kryptografie je velmi složitá a rozsáhlá. Opírá se o znalosti programování a zejména matematiky na nejvyšší úrovni. Některé programovací jazyky, jako je např. PHP, již mají v sobě jako základní funkce například funkci pro MD5 hash.

V této kapitole byla probrána důležitá nelineární datová struktura – binární strom a jeho varianta binární vyhledávací strom včetně definice základních pojmů. Vyhledávací stromy slouží zejména pro efektivní hledání prvků či nalezení maxima/minima. Druhou částí této kapitoly je stručné vysvětlení pojmů hashing a hashovací funkce včetně uvedení některých základních jako je MD5 či SHA. Hashovací funkce mají za úkol zakódovat vstupní data do řetězce konstantní délky, např. 128 bitů v případě MD5. Detailní matematické vysvětlení algoritmů hashovacích funkcí přesahuje rámec a odbornost této opory. Pro praktickou ukázku je uveden odkaz na web, kde si mohou studenti vyzkoušet MD5 a SHA v praxi.

1. Popište datovou strukturu stromu. Co je hloubka a výška stromu?
2. Uveďte vlastnosti binárního stromu a binárního vyhledávacího stromu.
3. Co je hash a hashovací funkce? Kde je problém bezpečnosti hashovací funkce?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Kapitola 12

Úvod do paralelního programování

Po prostudování kapitoly budete umět:

- vysvětlit význam paralelního programování
- popsat princip vícevláknového zpracování
- uvést jednoduché příklady použití paralelního programování

Klíčová slova:

Paralelní programování, C#, paralelizace algoritmu, vlákna, multihreading

12.1 Co je paralelní programování

Primárním cílem je vývoj algoritmů, které nabídnou větší výkon a efektivně využijí dostupný výkon vícejádrových procesorových architektur. Obecně lze říci, že paralelní algoritmy nemají smysl u velmi jednoduchých programů, které hravě zvládne jednojádrový procesor a algoritmus je natolik jednoduchý, že nemá význam rozdělovat jeho běh do vláken. Paralelizace by nepřinesla očekávaný přínos (nárůst výkonu) a pouze bychom ztratili čas návrhem. Přesto je dnes paralelní programování trendem vývojářů, jelikož se k tomu plně nabízí software (operační systémy) i hardware (vícejádrové procesory, grafické čipy apod.).

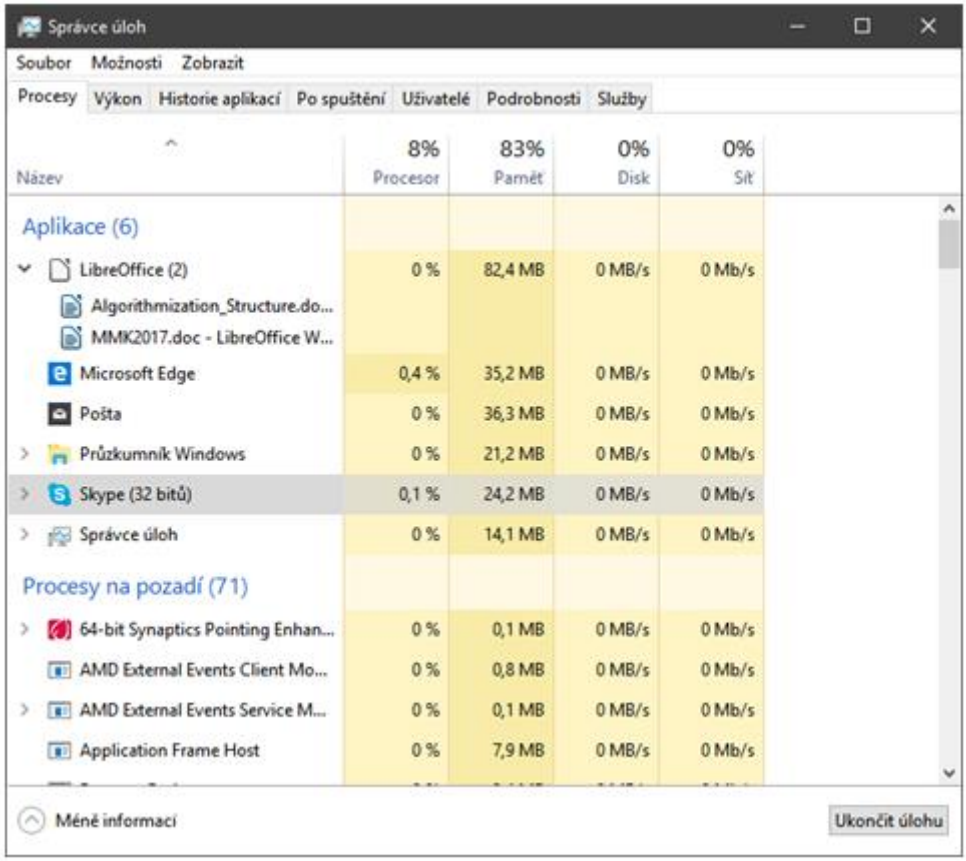
Paralelizace je běžná v dnešních operačních systémech. Běží Vám několik programů zároveň a některé programy lze otevřít vícekrát jako jednotlivé instance. Procesy jsou paralelně běžící instance téhož programu. Každý proces se sestává minimálně z jednoho vlákna. Obvykle běží jedno hlavní vlákno a to obsluhuje další vlákna a jejich běh.

Skutečně na jednom fyzickém procesoru (jednojádrovém) může současně běžet opravdu pouze jedno vlákno. Dojem paralelizace spočívá v přepínání vláken.

12.2 Aplikace, procesy a vlákna

Než se začneme hlouběji zabývat principy paralelních algoritmů, je nutno vysvětlit klíčové související pojmy. Těmi jsou ***aplikace***, ***proces*** a ***vlákno*** (angl. thread, proto mluvíme o tzv. multithreadingu, viz dále).

Pojem aplikace je Vám patrně zřejmý. Je to například webový prohlížeč, grafický editor nebo antivirový program. Běží ve svém okně, případně v Příkazovém okně (resp. v konzole v Linuxu). Jelikož aplikaci obvykle můžu spustit vícekrát nezávisle na sobě, každá spuštěná instance aplikace je proces. Některé rozsáhlejší aplikace spouštějí několik procesů zároveň, například webový prohlížeč může spouštět proces pro každou samostatnou záložku či okno. V praxi to pěkně uvidíme ve Správci úloh ve Windows.



Obrázek 12.1 Běžící procesy ve Windows

Vlákno je pojem trochu složitější. V každém procesu může běžet souběžně, tedy paralelně, několik vláken. Aplikace má vždy minimálně 1 hlavní vlákno a další vytváříme sami. Užitečně to je v případě náročných operací. Například náročný výpočet grafického renderingu může běžet ve svém vlákně, aniž by narušil hlavní vlákno aplikace. Operační systém spustí naše vlákno na nějakém jádru a potom ho rychle uspává a probouzí podle potřeby. V tom tkví princip vícevláknového běhu, tedy zmíněného multithreadingu.

Cílem paralelního programování, jak jste se již možná dovtípili, je zvýšení výkonu aplikace a oddělit běh jednotlivých částí do vláken. V dnešní době, kdy se čtyřjádrové procesory stávají de facto low-end kategorií, je paralelní programování běžnou součástí. Moderní procesory mají 6, 8, ale i 18 jader. Aplikace tak umějí vytěžit více výkonu díky rozdělení běhu v jednotlivých vláknech. Ve většině moderních programovacích jazyků, jako je C# nebo Java, se můžete pustit do paralelního programování. Právě například jazyk C# z rodiny .NET má velmi pokročilé prostředky pro paralelní programování. Vynikající publikací dostupnou online je *„Introduction to Parallel Computing“*, v níž čtenář najde zevrubné vysvětlení principu paralelních algoritmů.

U paralelních algoritmů nám obvykle jde zejména o úsporu času, tedy měříme exekuční čas programu. Nechť sestavíme sekvenční (neparalelní) algoritmus a ten bude mít exekuční čas T_S a ten

samý algoritmus převedeme na paralelní, jehož exekuční čas bude T_P, pak nás bude zajímat poměr T_S/T_P ke zjištění nárůstu časové efektivity. Pak lze nárůst rozdělit do tří kategorií:

- sublineární, je-li $T_S/T_P < n$
- lineární, je-li $T_S/T_P = n$
- superlineární, je-li $T_S/T_P > n$

kde n je počet jader procesoru vícejádrového CPU. Z toho následně lze stanovit efektivitu využití zdrojů systému danou vzorcem

$$e = [(T_S/T_P)/n]\times 100 \text{ (hodnota v \%)}$$

Výborným zdrojem pro praktické paralelní programování je PDF kniha od společnosti Microsoft[27], v níž se probírají konkrétní ukázky paralelních algoritmů pro matematické úlohy, jako jsou operace s 3D vektory, řešení soustav lineárních rovnice nebo numerické integrování. To vše v jazyce C#, který má obecně velmi propracované prostředky pro paralelní algoritmy.

12.3 Princip vláken a jejich synchronizace

Jak již bylo zmíněno, základem paralelních algoritmů je využití vláken. Každé vlákno má svůj „životní cyklus“ a během algoritmu se uvádí do několika stavů. Jelikož je vláken několik, je nutná jejich synchronizace. Tyto pojmy jsou stěžejní k pochopení paralelních algoritmů obecně. Vlákno lze nastavit do několika následujících stavů:

- pozastavení činnosti vlákna
- uspání vlákna (sleep)
- ukončení činnosti vlákna

V praxi je využíván tzv. pětistavový model vláken, což znamená, že každé vlákno má svůj „životní cyklus“ v pěti možných stavech následovně:

- **New** (nově vytvořené vlákno, neběží)
- **Ready** (připravené k běhu)
- **Running** (aktivní běžící vlákno)
- **Blocked** (blokované vlákno – uspání)

[27] http://download.microsoft.com/download/5/7/1/5711A481-E190-4D11-BA71-98B425599040/PPP_C_40_C.pdf

- **Exit** (ukončení vlákna)

Po odemčení ze stavu Blocked se vrací do stavu Ready.

Synchronizace činnosti jednotlivých vláken[28] je nutností zejména v případě, že vlákna využívají společná data a zdroje či jejich běh je vzájemně závislý, tedy jedno vlákno nemůže fungovat bez činnosti jiného. Používá se několik tzv. synchronizačních prostředků – kritické sekce, semafory, mutex a podmínky. Proč je nutno vlákna synchronizovat, tedy řídit jejich činnost:

- zabránit konfliktům mezi vlákny při přístupu ke sdíleným prostředkům (paměti, souborům, periferním zařízením, ap.)
- pozastavení běhu vláken, dokud nejsou splněny určitě podmínky či nenastane určená událost
- zajistit vykonání určitých programových sekvencí v požadovaném pořadí

Nejjednodušším typem synchronizace je kritická sekce. Princip spočívá v tom, že vlákno požadující výlučný přístup k některému ze sdílených zdrojů, může tento zdroj dočasně uzamčít k zabránění pokusu o přístup z jiného místa aplikace a po ukončení úprav jej opět odemknout.

Podmínky se používají v případě, že funkčnost vlákna závisí na určité splněné podmínce.

Detailní informace k možnostem a programování synchronizace vláken lze najít v literatuře.

12.4 Základy programování s vlákny

Moderní programovací jazyky, jako je Java a C#, nabízejí plnohodnotné možnosti paralelního programování s využitím vláken.

Příklad spuštění vlákna v jazyce Java pro úlohu s názvem *delejtohle*:

```
Thread nazev_vlakna = new Thread(delejtohle);

nazev_vlakna.start(); // vlákno běží
```

Obdobně v jazyce C# si spustíme 2 vlákna, která budou vypisovat písmeno A a B do konzole. Kód vyžaduje základní znalosti OOP.

```
class PsaniAB

{
```

[28] http://aldebaran.feld.cvut.cz/vyuka/uvod_do_os/38UOS-2010-05z_Synchronizace.pdf

```
        public void NapisA()
        {
                while (true)
                {
                        Console.Write("A");
                }
        }
        public void NapisB()
        {
                while (true)
                {
                        Console.Write("B");
                }
        }

        public void PrepniVlakna()
        {
                Thread vlakno = new Thread(NapisA);
                vlakno.Start();
                NapisB();
        }
}
```

12.5 Paralelní algoritmy v oblasti HPC

Paralelní algoritmy hrají primární roli v nejnáročnějších výpočtech a simulacích, v oblasti HPC neboli ***High Performance Computing***. Určitě jste již slyšeli pojem superpočítač. Jsou to nejvýkonnější počítače na světě, které obsahují stovky až tisíce fyzických procesorů a stovky tisíc jader. Běží na nich nejnáročnější výpočty a simulace v oblastech fyzicky, biochemie a dalších interdisciplinárních vědách. Výkon takových počítačů se měří v jednotkách petaflopů. V současnosti (listopad 2017) jsou 3 nejvýkonnější počítače světa v Číně.

Paralelní algoritmy a paralelní programování je v současnost rychle rostoucím trendem v souvislosti s možnosti využití vícejádrových procesorů a vícevláknového programování. Paralelizace se uplatňuje zejména v náročných výpočetních aplikací, kdy jednotlivé úlohy rozdělíme do jednotlivých vláken, která se mezi sebou přepínají. Díky dvou a vícejádrovým procesorům se paralelní algoritmy uplatňují stále více.

1. Co je vlákno a co je proces?
2. Jak dělíme nárůst efektivity paralelního algoritmu oproti sekvenčnímu algoritmu?

Základní literatura:

[1] PŠENČÍKOVÁ, J. *Algoritmizace.* 2.vyd. Praha: Computer Media, 2009. 128 s. ISBN 978-7402-034-6.

[2] VRBÍK, V. *Algoritmy – řešené příklady.* 1.vyd. Plzeň: Pedagogické centrum Plzeň, 2002. 44. ISBN 978-80-702-0103-7.

[3] WRÓBLEWSKI, P. *Algoritmy: datové struktury a programovací techniky*. 1. vyd. Brno: Computer Press, 2007. 351 s. ISBN 978-80-251-0343-9.

Doporučená literatura:

viz seznam doporučené literatury na závěr studijní opory

Závěr

Cílem této studijní opory bylo obsáhnout úvod do problematiky algoritmizace. Publikace se věnuje stěžejním tématům pro pochopení algoritmického myšlení. Od pojmu algoritmus až po úvod do paralelních algoritmů. Jednotlivé kapitoly na sebe logicky navazují svým pojetím a v textu se vyskytují četné odkazy na související kapitoly.

Vzhledem k vymezenému rozsahu těchto studijních opor není možné se každé kapitole věnovat zevrubně do hloubky. V každé kapitole však čtenář najde to, co je základem pro pochopení tématu kapitoly a další znalosti lze čerpat z uvedené literatury a četných webových odkazů. Opory by měly studentům pomoct v orientaci v problematice obecné algoritmizace bez vazby n konkrétní programovací jazyk. Cílem této publikace není zvládnutí programování, ale osvojení si základních algoritmických postupů, prvků a návazností. Čtenář tak bude vědět, k čemu slouží vývojové diagramy a jak se používají, co znamená v algoritmu podmínka a cyklus či co je rekurze a že má využití nejen v počítačové grafice.

Vybraná témata jsou doplněna o zápis pomocí tzv. pseudokódu, který je univerzální pro pochopení, jak se daná struktura naprogramuje, ovšem bez konkrétní syntaxe programovacího jazyka. Opory se nevyhýbají ani základům výpočetní složitosti či úvodu do paralelního programování. Není cílem zahltit studenty formálními definicemi a matematickými vzorci, ale jde primárně o pochopení dané problematiky na jednoduchých příkladech. Autor publikace dává přednost intuitivnímu pojetí textu s názornými obrázky, například přirovnání abstraktního datového typu fronty na analogií s reálnou frontou u pokladny. Předmět Algoritmizace není o programování, ale o pochopení principů návrhu algoritmů a schopnosti daný problém analyzovat, rozdělit na menší části (dekompozice a metoda Rozděl a panuj) a pochopit jednotlivé vazby.

Na konci každé kapitoly jsou vždy kontrolní otázky, které umožní studentům si ověřit, jak zvládli probíraná témata. Zároveň tyto otázky jsou námětem k diskusi a odkazem k hlubšímu studiu probíraného tématu kapitoly.

Věřím, že tyto studijní opory budou pro studenty skutečně oporou a pevným základem pro další studium. Po absolvování předmětu Algoritmizace by studenti měli mít pevné základy algoritmického myšlení a znát klíčové prvky návrhu a analýzy algoritmů a to včetně porozumění výpočetní složitosti a jakým způsobem lze algoritmy optimalizovat. Zájemce o další vzdělávání odkazujeme na literaturu a v textu na webové stránky věnující se daným tématům.

Doporučená literatura

[1] CORMEN T. H., LEISERSON, C.E., RIEVEST, R.L., STEIN, C. *Introduction to Algorithms*. 3. vyd. Cambridge: The MIT Press, 2009. 1312 s. ISBN 978-02-620-3384-4.

[2] DVORSKÝ J. *Algoritmy*. Ostrava: Vysoká škola báňská – Technická univerzita, 2007. Dostupné z: http://www.cs.vsb.cz/dvorsky/Opora.html.

[3] JAJA, J. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1st Edition, 576 s., ISBN: 978-0201548563.

[4] KNUTH, D. *The Art of Computer Programming*. Volumes 1-4. Addison-Wesley Professional, 2011, 3168 p. ISBN: 978-0321751041.

[5] POKORNÁ, P. *Problém obchodního cestujícího pomocí metody Mravenčí kolonie*. Bakalářská práce, Univerzita Pardubice, Fakulta ekonomicko-správní, 2008.

[6] RYANT, I. *Algoritmy a datové struktury objektově*. 286 s., ISBN: 9788027016600.

[7] SEDGEWICK, R. a WAYNE, K. *Algorithms*. 4. vyd. Indianapolis: Addison-Wesley Professional, 2011. 992 s. ISBN 978-03-215-7351-X.

[8] TRETEROVÁ. E. *Návrh a vývoj algo*ritmů. 1. vyd. Ostrava: Ostravská univerzita v Ostravě, 2003. 64 s. ISBN 978-80-704-2854-6.

Online zdroje

Všechny online zdroje (poznámky pod čarou) 1 až 29 uvedené v textu byly ověřeny dne 22. 11. 2017.