# Základy algoritmizace

**Miroslav Virius**

# Obsah

# Předmluva

Toto skriptum je určeno posluchačům prvního ročníku FJFI ČVUT se softwarovým zaměřením. Navazuje na přednášku *Základy algoritmizace*.

Algoritmy jsou nerozlučně spjaty s datovými strukturami. Proto se hned na počátku zabýváme nejčastěji používanými datovými strukturami a základními operacemi s nimi. V dalších kapitolách se pak seznámíme s některými metodami návrhu algoritmů.

Samostatné kapitoly jsou věnovány známým a často používaným algoritmům, jako jsou algoritmy pro třídění polí a souborů nebo základní algoritmy pro práci s binárními stromy. V kapitole o seminumerických algoritmech se čtenář seznámí se základními algoritmy pro práci s čísly.

Příklady k probírané látce jsou napsány převážně v Turbo Pascalu, který posluchači již znají. V několika případech se však ukázalo vhodnější použít programovací jazyk C++, neboť ten nabízí možnost elegantnějšího a přehlednějšího zápisu. (Tak tomu je např. ve výkladu o rychlé Fourierově transformaci, kde potřebujeme komplexní čísla. V C++ můžeme použít datový typ, definovaný ve standardní knihovně, a operace můžeme zapisovat pomocí běžných aritmetických operátorů.)

Vzhledem k omezenému rozsahu přednášky obsahuje toto skriptum pouze nejzákladnější informace. Analýze algoritmů, tj. stanovení jejich časové a paměťové náročnosti, věnujeme pouze okrajovou pozornost. V některých případech pouze uvádíme známé výsledky a nedokazujeme je, neboť podrobné odvozování by často přesahovalo znalosti posluchačů 1. ročníku. Ostatně analýza algoritmů je předmětem zvláštní přednášky ve vyšších ročnících.

Také numerické algoritmy jsou předmětem zvláštní přednášky a proto zde chybí.

Na závěr skripta je zařazeno povídání o softwarovém projektu, o návrhu programu na základě analýzy požadavků a úvod do objektově orientovaného programování. Jde o velmi stručný úvod do této problematiky. Podrobnější informace najde čtenář např. v literatuře, uvedené v odkazech.

Na závěr bych chtěl poděkovat všem, kteří svými radami a připomínkami přispěli ke zdárnému dokončení tohoto díla, zejména však recenzentovi, RNDr. Januši Drózdovi, který skriptum velice pečlivě přečetl a měl k němu řadu podnětných připomínek.

M. Virius

# Kapitola 1

# Algoritmus

V převážné části tohoto skripta se budeme zabývat algoritmy a metodami pro jejich návrh. Začneme ale tím, že se dohodneme, co to vlastně algoritmus je a jak jej budeme zapisovat.

## 1.1 Vymezení pojmu

Ve většině publikací, věnovaných úvodu do algoritmizace nebo programování, se pojem *algoritmus* nezavádí - autor prostě předpokládá, že čtenář rozumí, o co jde. Následující vymezení tohoto pojmu jsme převzali z [9].

### 1.1.1 Co je to algoritmus

Algoritmus je základní matematický pojem. To znamená, že jej nelze definovat - musíme se uchýlit k opisu, podobně jako u dalších elementárních pojmů, jakými jsou např. bod nebo množina.

Algoritmus je v podstatě návod, jak provést určitou činnost; v případě programování půjde zpravidla o transformaci množiny vstupních dat na množinu výstupních dat. Ovšem ne každý návod představuje algoritmus. Jako algoritmus budeme označovat návod, který má následující vlastnosti:

1. Je *elementární*. To znamená, že se skládá z konečného počtu jednoduchých, snadno realizovatelných činností, které budeme označovat jako *kroky*. (Dále si povíme, co myslíme tou „jednoduchou“ činností.)
2. Je *determinovaný*, tj. po každém kroku lze určit, zda popisovaný proces skončil, a pokud neskončil, kterým krokem má algoritmus pokračovat.
3. Je *konečný*. Počet opakování jednotlivých kroků algoritmu je vždy konečný. Algoritmus tedy musí skončit po konečném počtu kroků.
4. Je *rezultativní*. Vede ke správnému výsledku.
5. Je *hromadný*. To znamená, že algoritmus můžeme použít k řešení celé (velké) skupiny podobných úloh.

V souvislosti s algoritmy se pro označení objektu (může to být stroj nebo i člověk), který bude provádět popisovanou činnost, používá termín procesor. Je jasné, že při formulaci algoritmu musíme znát procesor. Přesněji řečeno musíme vědět, jak vypadají elementární kroky, které může návod obsahovat - a ty zřejmě závisí na povaze procesoru.

> Poznámka: Pojem algoritmus se dá formalizovat např. pomocí matematické konstrukce, označované jako Turingův stroj, nebo pomocí teorie parciálně rekurzivních funkcí. Výklad o nich však přesahuje rámec našeho skripta.

### 1.1.2 Příklad 1.1: procesor

Jako příklad vezmeme řešení kvadratické rovnice, zadané koeficienty $a$, $b$ a $c$. Budeme-li za procesor považovat např. CPU počítače PC, budou elementární krok představovat jednotlivé instrukce strojního kódu a algoritmus se bude skládat z pokynů tvaru *„přesuň obsah proměnné c do registru ST“*, *„vynásob obsah ST(1) a ST“* apod.[1]

[1] ST resp. ST(i) jsou registry matematického koprocesoru ix87.

Budeme-li však uvažovat o počítači, vybaveném překladačem jazyka Pascal, budou elementární kroky představovat jednotlivé příkazy Pascalu. Algoritmus se pak bude skládat z příkazů tvaru

```
d := sqr(b) - 4*a*c;
```

Bude-li procesorem člověk, obeznámený se základy středoškolské matematiky, postačí mu instrukce *„vyřeš kvadratickou rovnici s koeficienty a, b a c“*.

### 1.1.3 Metody shora dolů a zdola nahoru

Algoritmus je tedy zápisem postupu, použitelného pro řešení určité třídy problémů. Jak dospět k formulaci, který bude splňovat výše uvedené podmínky? Samozřejmě nejprve musíme daný problém umět vyřešit. V následujících kapitolách se seznámíme s některými postupy, které hledání řešení usnadňují, a s řadou příkladů - tedy vyřešených problémů.

Jestliže již řešení známe, potřebujeme je zapsat jako algoritmus. Přitom postupujeme obvykle tak, že postup řešení rozkládáme na jednodušší operace, až dospějeme k elementárním krokům.

Tento postup návrhu algoritmu se obvykle označuje jako metoda „shora dolů“.

### Příklad 1.2: kvadratická rovnice

Zůstaňme u kvadratické rovnice. Budeme chtít napsat program, který bude řešit rovnice, jejichž koeficienty jsou v uloženy souboru *A*. Výsledek se má uložit do souboru *B*.

Nejhrubší formulace může vypadat takto:

*Dokud nenarazíš na konec souboru A, řeš rovnice, určené koeficienty uloženými v souboru A a výsledky zapisuj do souboru B.*

Nyní potřebujeme upřesnit význam fráze *„řeš rovnice, určené...“* - musíme ji rozložit na jednodušší kroky. Vedle toho si ale musíme uvědomit, že každý program obsahuje nezbytné (ale v zápisech algoritmů často opomíjené) úvodní a závěrečné operace, jako je otevírání a zavírání souborů, inicializace pomocných proměnných aj.). Ani náš program nebude výjimkou, a proto je do algoritmu zahrneme, i když jejich přesný význam určíme později.

Dostaneme následující formulaci:

1. *Proveď úvodní operace.*
2. *Dokud nenarazíš na konec souboru A, opakuj kroky* 3 - 5*, potom přejdi na krok* 6*.*
3. *Přečti se souboru koeficienty a, b a c.*
4. *Vyřeš kvadratickou rovnici s koeficienty a, b, c.*
5. *Zapiš výsledky do souboru B a vrať se na* 2*.*
6. *Proveď závěrečné operace.*

Nyní je třeba zpřesnit body 1, 4 a 6. My se podíváme jen na bod 4, ostatní si můžete zkusit sami. Postup řešení kvadratické rovnice jistě znáte, takže budeme struční.

4a. Vypočti $d = b^2 - 4ac$.

4b. Je-li $d < 0$, pokračuj bodem 4f, jinak pokračuj bodem 4c.

4c. Polož $d = \sqrt{d}$.

4d. Rovnice má kořeny $x_{1,2} = \frac{-b \pm d}{4ac}$ .

4e. Jdi na 5.

4f. Polož $d = \sqrt{-d}$ .

4g. Rovnice má kořeny $x_{1,2} = \frac{-b \pm id}{4ac}$ , kde $i$ je imaginární jednotka.

Dále si musíme ujasnit, co budeme dělat v případě chyby (nepodaří se otevřít soubor, budou v něm chybná data atd.), v jakém formátu jsou uložena vstupní data (zda to budou např. celá, reálná nebo komplexní čísla), v jakém formátu budeme zapisovat výsledky (jak budeme zapisovat komplexní čísla atd.).

Kromě metody „shora dolů“ se občas setkáme i s metodou, označovanou jako návrh „zdola nahoru“. Při postupu zdola si postupně z elementárních kroků vytváříme prostředky, které nakonec umožní zvládnout požadovaný problém.

S trochou nadsázky lze tvrdit, že při metodě zdola nahoru si vytváříme nový procesor tím, že ho učíme nové operace (přesněji: učíme ho chápat skupiny elementárních operací jako nové elementární kroky.)

Obvykle se kombinuje postup shora dolů s postupem zdola nahoru. Postup shora dolů, tedy rozklad postupu řešení na elementární kroky, doplníme „částečným krokem“ zdola nahoru tím, že např. použijeme překladače některého vyššího programovacího jazyka, knihovny procedur a funkcí nebo systému pro vytváření programů (CASE).

### 1.1.4 Základní složky algoritmu

V algoritmech se setkáváme se třemi základními konstrukcemi, které označujeme jako posloupnost (sekvenci) příkazů, s cyklus (iteraci) a s podmíněnou operaci (selekci, výběr).

*Posloupnost* (*sekvence*) je tvořena jedním nebo několika kroky, které se provedou právě jednou v daném pořadí. Přitom nemusí jít o kroky elementární; při dalším zjemňování se součásti sekvence mohou rozpadnout na součásti, které samy budou tvořit posloupnosti, cykly nebo podmínky. V Pascalu může být posloupnost vyjádřena složeným příkazem, v Céčku blokem.

Za příklad posloupnosti můžeme považovat kroky 4c - 4e v příkladu 1.2.

*Cyklus* (*iterace*) představuje část algoritmu, která se opakuje, dokud je splněna podmínka opakování. Cyklus se vždy skládá z podmínky opakování a z těla cyklu, tedy z operací, které se opakují.

Podmínka se může vyhodnocovat před provedením těla cyklu (v Pascalu nebo v Céčku příkazy **while** nebo **for**), po skončení těla cyklu (v Pascalu příkaz repeat, v Céčku příkaz **do - while**) nebo i uvnitř těla cyklu (pokud bychom takový příkaz potřebovali v tradičním Pascalu, museli bychom si jej vytvořit pomocí podmíněného skoku; v Céčku - a také v Turbo Pascalu 7.0 - lze použít kombinaci podmínky a příkazu **break**).

V příkladu 1.2 tvoří kroky 2 - 5 cyklus s podmínkou na počátku.

*Podmíněná operace* (*selekce*) představuje vždy větvení algoritmu. Je tvořena podmínkou a jednou, dvěma nebo více výběrovými složkami. Nejprve se vždy vyhodnotí podmínka a ta určí, zda se bude provádět některá z výběrových složek - a pokud ano, která. Nemusí se tedy provést žádná z výběrových složek. Pokud se jedna z nich zvolí, provede se jednou.

Pro vyjádření selekce slouží v Pascalu příkazy **if** (úplné nebo neúplné) a **case**. V Céčku máme k dispozici příkazy **if** a **switch** (v kombinaci s příkazem **break**).

V příkladu 1.2 je podmíněný příkaz 4b.

Jestliže se určitá část algoritmu opakuje na několika místech (může přitom používat různá data), stačí rozložit ji na elementární kroky pouze jednou (např. když na ni narazíme poprvé). Na ostatních místech se na ni pak odvoláme jako na *dílčí algoritmus nebo podprogram*.

V běžných programovacích jazycích odpovídají podprogramům procedury a funkce.

### 1.1.5 Algoritmus a data

Jak jsme si již řekli, vyjadřují počítačové algoritmy zpravidla návody, jak transformovat množinu vstupních dat v množinu jinou výstupních dat. Proto je samozřejmé, že struktura vstupních, výstupních, ale (často především) vnitřních dat spoluurčuje strukturu algoritmu.

Je jasné, že při zpracování vstupních příp. výstupních dat bude posloupnosti datových položek různých druhů zpravidla odpovídat posloupnost různých příkazů. Pokud se opakují datové položky téhož druhu (iterace), bude jejich zpracování vyžadovat nejspíše použití cyklu. Pokud se na jednom určitém místě v datech (vstupních nebo výstupních) mohou vyskytnout údaje několika různých druhů, použijeme při jejich zpracování selekci (podmíněný příkaz).

Ukazuje se také, že přirozeným prostředkem pro zpracování rekurzivních datových struktur, jako jsou stromy nebo seznamy, jsou obvykle rekurzivní algoritmy.

### 1.1.6 Časová a paměťová náročnost algoritmů

Při analýze algoritmů nás zajímá nejen správnost (zda dostaneme správný výsledek) a v případě numerických algoritmů přesnost, ale také doba, kterou budeme k provedení algoritmu potřebovat, a množství operační paměti, které bude potřebovat program, realizující algoritmus.

Při hodnocení časové náročnosti můžeme vycházet z celkového počtu elementárních kroků, tedy např. instrukcí strojového kódu, které musíme provést, v závislosti na rozsahu vstupních příp. výstupních dat.

Při hodnocení časové náročnosti musíme vzít v úvahu, že doby, potřebné pro provedení různých instrukcí, se mohou drasticky odlišovat. Např. na procesoru Intel 80486 trvá sečtení dvou reálných čísel v průměru 10 „tiků“, zatímco výpočet absolutní hodnoty zabere 3 „tiky“. Výpočet sinu (také jedna instrukce procesoru Intel 80486) zabere v průměru 291 tiků. Přitom 1 „tik“ při hodinové frekvenci procesoru 33 MHz trvá cca $3.10^{-8}$ s.

Na druhé straně přečtení jednoho sektoru na pevném disku trvá v současné době např. 12 - 15 ms, tedy přibližně o 3 řády déle než provedení těch nejnáročnějších instrukcí procesoru. Proto se ve skutečnosti zpravidla stačí orientovat se podle vybraných skupin instrukcí, které jsou časově nejnáročnější.

## 1.2 Popis algoritmů

Algoritmy lze vyjadřovat mnoha různými způsoby. Některé se opírají pouze o slovní vyjádření, jiné používají grafických prostředků. Volba vhodného prostředku se může lišit podle charakteru řešené úlohy a podle osobních zvyklostí programátora. Ukážeme si některé často používané.

### 1.2.1 Jazyk pro popis programů

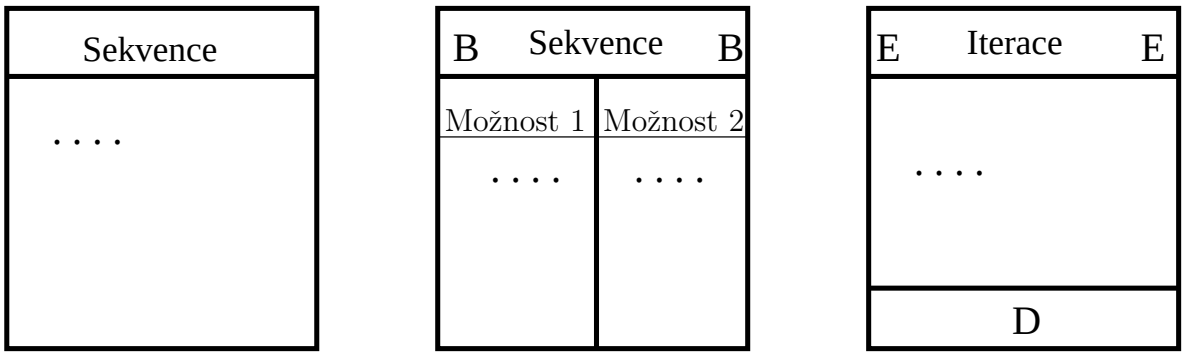
Tento způsob popisu algoritmů se dnes používá patrně nejčastěji. Jeho označení *jazyk pro popis programů* je doslovným překladem anglického termínu *Program Description Language* (PDL). Vychází se slovního popisu algoritmu. Zachycuje postupný rozklad algoritmu na jemnější kroky při návrhu metodou shora dolů.

Příklad 1.2 ukazuje postupné vytváření popisu algoritmu pro řešení kvadratických rovnic. Přitom je často rozumné při postupném zpřesňování ponechat i původní rámcový popis „většího“ kroku, neboť usnadňuje čtenáři orientaci.

Poznamenejme, že slovní popis algoritmu se může stát základem dobrého komentáře k výslednému programu.

Vedle toho často k zápisu algoritmů používáme lehce upravený programovací jazyk - např. *Pascal*. Při takovémto zápisu často v zájmu přehlednosti a srozumitelnosti porušujeme syntaktická pravidla: v identifikátorech používáme písmena s diakritickými znaménky, vkládáme slovní popis operací apod. S touto formou zápisu se zde budeme setkávat poměrně často a budeme ji také označovat jako *Pseudopascal*.

Poznamenejme, že tradičním jazykem pro popis algoritmů byl programovací jazyk *Algol 60*. Setkáme se s ním běžně v publikacích ze 60. a 70. let, dnes již spíše výjimečně. (Jednou takovou výjimkou je např. [26].)




Obr. 1.1: Možné tvary struktogramů

Vzhledem ke značné podobnosti mezi Algolem a Pascalem je obvykle zápis algoritmu v Algolu srozumitelný i pro čtenáře, který zná jen Pascal.[2]

### 1.2.2 Struktogramy

Struktogramy graficky znázorňují strukturu algoritmu. Používají tvarově (nebo i barevně) odlišné vyjádření pro základní algoritmické struktury (posloupnost, cyklus, podmínka).

Základem struktogramu je vždy obdélník, v jehož záhlaví je označení algoritmu nebo dílčí operace. Uvnitř obdélníku jsou vypsány kroky, které algoritmus tvoří; mohou do něj být vnořeny i další struktogramy. Struktogramy můžeme používat jak při rozboru úlohy na nejvyšší úrovni tak při vlastním programování. Poskytují také dobrou dokumentaci postupu při návrhu.

**Příklad 1.3: opět kvadratická rovnice**

Vrátíme se ještě jednou k programu na řešení kvadratických rovnic. Struktogramy, které vyjadřují postup řešení, vidíte na obrázku 1.2. Vzhledem k tomu, že struktogram pro operaci *„Vyřeš rovnici s danými koeficienty“* nelze dost dobře vložit dovnitř struktogramu, popisujícího cyklus čtení dat ze souboru, zakreslíme jej vedle.

### 1.2.3 Jacksonovy diagramy

Další možností, kterou lze použít při popisu algoritmů, jsou Jacksonovy diagramy, se kterými se setkáme v kapitole 10.2.1.

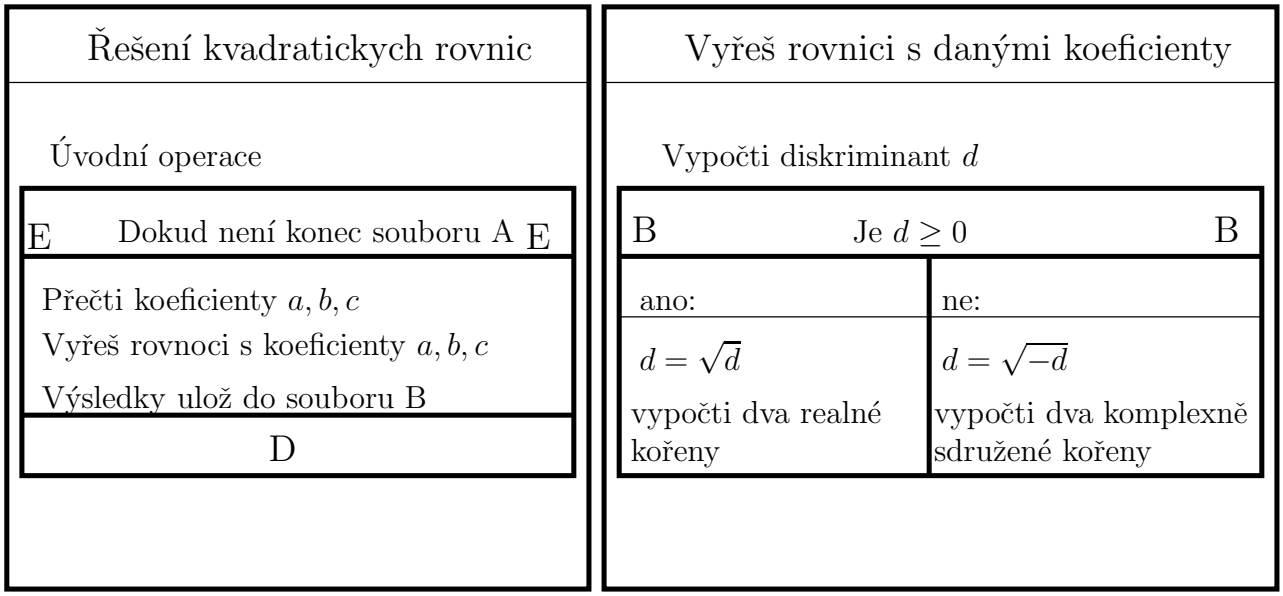
### 1.2.4 Vývojové diagramy

Klasickým prostředkem pro znázornění algoritmu jsou vývojové diagramy. Znázorňují „tok řízení“ v algoritmu. Značky v nich, používané u nás, jsou upraveny normou [10].

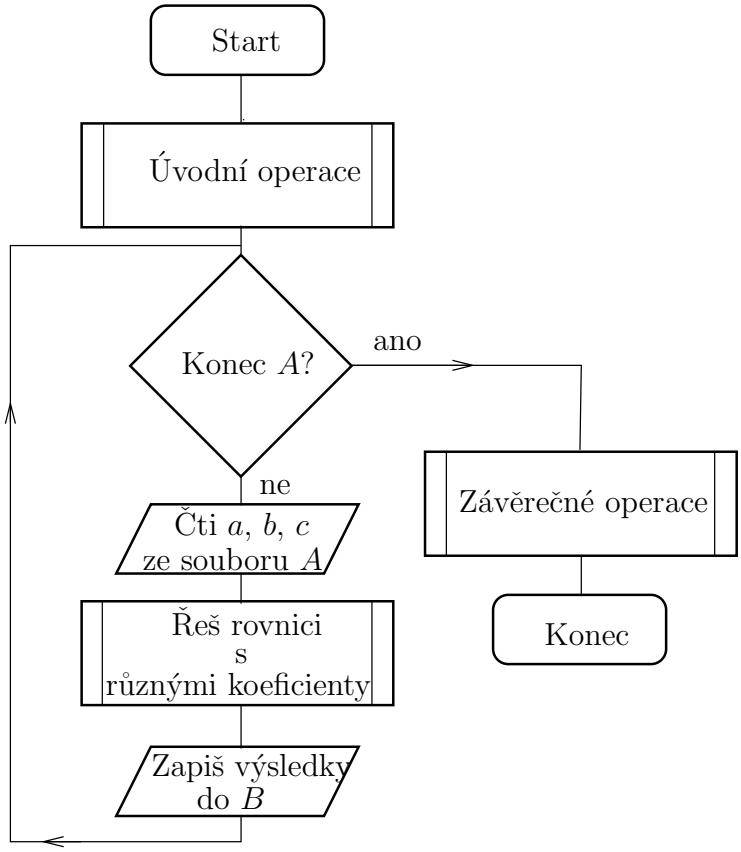
Od používání vývojových diagramů při programování se dnes upouští (přesněji: prakticky se nepoužívají), ve starších publikacích se s nimi ovšem lze stále ještě setkat. Zpravidla se jim vytýká, že spíše než logickou strukturu programu zdůrazňují druh operací.

Hlavním problémem vývojových diagramů ovšem je, že jde o graficky značně náročný způsob dokumentace, který často zachycuje i jednotlivé příkazy. Při pozdějších změnách programu - i nepatrných - se zpravidla vývojový diagram již neaktualizuje, takže velice rychle zastarává. Jako příklad uvedeme vývojový diagram programu pro řešení kvadratických rovnic (viz obr. 1.3).

[2] Totéž však nelze říci o programu v Algolu. Jazyk Algol obsahuje některé konstrukce, se kterými se v ostatních programovacích jazycích setkáme jen výjimečně - např. předávání parametrů procedur a funkcí jménem. Tato poněkud problematická konstrukce se naštěstí při popisu algoritmu zpravidla nevyužívá.




Obr. 1.2: Struktogram řešení kvadratické rovnice




Obr. 1.3: Hrubý vývojový diagram řešení kvadratické rovnice

# Kapitola 2

# Datové struktury

Povídání o datových strukturách je nezbytnou součástí jakéhokoli výkladu o algoritmech; podobně při výkladu o datových strukturách nelze pominout algoritmy, které se používají při práci s nimi.

V této kapitole si povíme o nejčastěji používaných datových strukturách. Pro přehlednost si je rozdělíme na *základní* a *odvozené*.

## 2.1 Základní datové struktury

Jako základní datové struktury budeme označovat *proměnnou, pole, záznam,* a *objekt*. S těmito datovými strukturami - snad až na objekt - se setkáme ve většině funkcionálně orientovaných programovacích jazyků.

Oddíl, věnovaný základním datovým strukturám, doplňuje a upřesňuje vědomosti, které získal čtenář v základním kurzu programování.

### 2.1.1 Proměnná

Proměnná představuje vlastně pojmenované místo v paměti počítače. Vytvoří se na základě *deklarace*, ve které sdělíme její *jméno* (obvykle identifikátor) a *typ*.

(Připomeňme si, že specifikací typu určujeme množinu hodnot, které do dané proměnné budeme smět ukládat, a množinu operací, které s danou proměnnou budeme moci provádět. Nepřímo tím obvykle také určujeme velikost proměnné, tj. množství paměti, které bude proměnná zabírat.)

Počítač zachází s proměnnou prostřednictvím její adresy. V programu je tato adresa vyjádřena jménem proměnné. Použití proměnné v programu může mít dva poněkud odlišné významy. Podívejme se na následující přiřazovací příkaz v Pascalu:

```
var i: integer;
...
i := i+1;
```

Zápis proměnné $i$ na pravé straně přiřazovacího příkazu znamená odkaz na hodnotu typu *integer*, uloženou na místě, označeném jménem $i$. Zápis této proměnné na levé straně přiřazovacího příkazu však znamená pouze odkaz na místo v paměti (na které se uloží výsledek, tedy hodnota pravé strany).

S podobným rozdílem se setkáváme i při předávání parametrů procedur a funkcí hodnotou nebo odkazem. Budou-li $f$ a $g$ dvě procedury s hlavičkami

```
procedure f(k: integer);
procedure g(var k: integer);
```

znamená zápis $f\,(i)$ volání procedury, při kterém se bude parametr předávat hodnotou. To znamená, že procedura $f$ dostane hodnotu výrazu, zapsaného jako skutečný parametr, tedy hodnotu, uloženou v paměti na místě jménem $i$.

Na druhé straně zápis $g(i)$ představuje volání procedury, při kterém se bude parametr předávat odkazem. To znamená, že procedura $g$ dostane odkaz na proměnnou $i$, zapsanou jako skutečný parametr, tj. její adresa. Procedura $g$ může využít buď hodnotu, uloženou ve skutečném parametru, nebo místo v paměti, které tento parametr zabírá.

Některé programovací jazyky - např. C++ - také umožňují, aby funkce vracely vypočtenou hodnotu odkazem (referenční funkce). Také v takovém případě se vrací odkaz na místo, kde je výsledek uložen, tedy adresa výsledku. Podobně jako při předávání parametrů odkazem i zde můžeme využít buď vrácenou hodnotu nebo místo v paměti, které vrácený odkaz označuje.

#### Druhy proměnných

Ve většině programovacích jazyků se setkáme se třemi základními druhy proměnných, které se liší způsobem alokace (přidělení paměti):

**Globální proměnné** se vytvoří při spuštění programu a existují po celou dobu běhu. To znamená, že mají v programu stálou adresu. Např. v Pascalu jsou to proměnné, deklarované na úrovni programu nebo jednotek, v Céčku proměnné deklarované mimo těla funkcí nebo proměnné s paměť'ovou třídou **static**.

**Lokální proměnné** (v Céčku se označují jako *automatické*) jsou proměnné, deklarované v procedurách nebo funkcích. Vznikají v okamžiku volání podprogramu, při ukončení podprogramu zanikají. Při rekurzivním volání podprogramu ve vytvoří zvláštní instance lokálních proměnných pro každou aktivaci. Při různých voláních téhož podprogramu může být jedna lokální proměnná uložena na různých místech v paměti počítače.

**Dynamické proměnné** vznikají za běhu programu na základě příkazů (obvykle volání procedur pro alokaci paměti - např. v Pascalu procedury *New*, v Céčku funkce *malloc*). Podobně na základě příkazů programu i zanikají. Prostor pro ně čerpá program z volné paměti. Existence dynamických proměnných není vázána na začátek nebo konec žádného podprogramu nebo bloku.

### 2.1.2 Pole

Pole je posloupnost proměnných stejného typu (složek), uložených v paměti v nepřetržité řadě za sebou, a chápaných jako jeden celek.

V deklaraci pole určujeme jeho jméno, tj. jméno, které označuje danou posloupnost jako celek, a typ počet složek. Počet složek je dán rozsahem indexů. Chceme-li pracovat s jednotlivými prvky, určujeme je pomocí indexů.

#### Jednorozměrné pole

Jako *jednorozměrná* označujeme pole, jejichž prvky již nejsou pole - tedy pole s jedním indexem. Podívejme se na příklad deklarace jednorozměrného pole:

```
var a: array [m .. n] of typ_složky;
```

Pole $a$ se skládá z $n - m + 1$ složek typu *typ_složky*. Přitom adresa prvního prvku, $a[m]$, je totožná s adresou celého pole $a$. Adresu $i$-tého prvku, $a[i]$, vypočteme pomocí tzv. indexovací funkce

$$t(i) = a + (i - m)\, v,$$

kde v je velikost typu *typ_složky*, tj. počet adresovatelných jednotek paměti[1], které zabírá jedna složka pole, a $a$ je adresa počátku pole.

V tomto vzorci zacházíme s adresami jako s celými čísly.[2]

[1] Na PC je adresovatelnou jednotkou paměti 1 byte, slabika velikosti 8 bit ; na jiných počítačích to mohou být slova různé velikosti - např.16 nebo 32 bit .

[2] Nepoužíváme tedy např. adresové aritmetiky jazyka C.

**Vícerozměrné pole**

Pole s $n$ indexy označujeme jako $n$-rozměrné.

Je-li $n > 1$, chápe se $n$-rozměrné pole zpravidla jako pole jednorozměrné, jehož prvky jsou $(n-1)$-rozměrná pole. Proto např. jsou v Pascalu následující dvě deklarace ekvivalentní:

```
var b: array [m1..n1, m2..n2, ... mk..nk] of typ_složek;
var b: array [m1..n1] of array [m2..n2] of ...
                      ... of array [mk..nk] of typ_složek;
```

Složky vícerozměrného pole jsou v paměti ukládány zpravidla tak, že se nejrychleji mění poslední index (zapsaný nejvíce vpravo)[3]. To znamená, že dvourozměrné pole, tj. matice, je uloženo po řádcích.

Indexovací funkce pro vícerozměrné pole je podstatně složitější než v případě pole jednorozměrného. Adresa prvku $b[i_1, i_2, \ldots, i_k]$ bude

$$\begin{aligned} t(i_1, \ldots, i_k) &= b + [(i_1 - m_1)(n_2 - m_2 + 1) \cdots (n_k - m_k + 1) + (i_2 - m_2)(n_3 - m_3 + 1) \cdots \\ &\quad \cdots (n_k - m_k + 1)(i_k - m_k)]\, v \end{aligned}$$

Zde $b$ znamená adresu počátku pole $b$ a $v$ představuje opět velikost jednotlivých složek pole. (Také v tomto vzorci zacházíme s adresami jako s celými čísly.)

Při výpočtu hodnoty adresovací funkce $t$ pro $k$-rozměrné pole potřebujeme $k$ sčítání a $k$ násobení (výrazy $(n_s - m_s + 1) \cdots (n_k - m_k + 1)$, $s = 2, \ldots, k-1$ jsou pro dané pole konstantní a lze je spočítat předem).

Pro přístup ke složkám vícerozměrných polí se také někdy používají přístupové (Iliffovy) vektory. V případě dvourozměrných polí to jsou pole ukazatelů na řádky; pro vícerozměrná pole to mohou být pole ukazatelů na pole ukazatelů na jednorozměrná pole apod.

Podívejme se na příklad, ve kterém deklarujeme a použijeme přístupový vektor pro dvourozměrné pole:

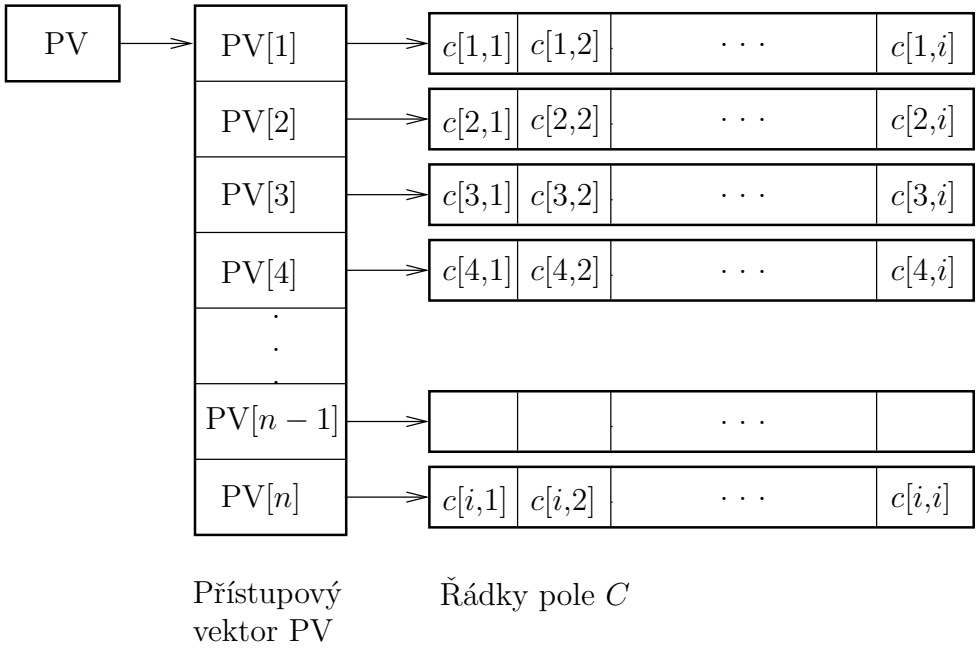
```
const n = 10;
type Pole = array [1..n] of integer;
var c: array [1..n, 1..n] of integer;
    pv: array [1..n] of ^Pole;
{ ... }
procedure init;
  var i: integer;
begin
  for i := 1 to n do pv[i] := @c[i];
end;

var i: integer;
{ ... }
 init;
 for i := 1 to n do pv[i]^[i] := i;
```

Poslední příkaz znamená totéž jako cyklus

```
 for i := 1 to n do c[i,i] := i;
```

Viz též obr. 2.1.
Na počítačích, na kterých je velikost ukazatele rovna 1, tj. na kterých zabírá ukazatel právě jednu adresovatelnou jednotku paměti, znamená použití přístupových vektorů zrychlení výpočtu, neboť odpadne většina násobení. Na PC a obecně na počítačích, kde je velikost ukazatele větší než 1, je použití přístupových vektorů v podstatě stejně náročné jako výpočet indexovací funkce.

[3]Tak tomu je např. v Pascalu nebo v C/C++. Na druhé straně ve Fortranu jsou vícerozměrná pole ukládána tak, že se nejrychleji mění první (nejlevější) index.




Obr. 2.1: Použití přístupového vektoru

### 2.1.3 Záznam (struktura)

Záznamy (v některých programovacích jazycích označované jako *struktury*) představují skupinu proměnných chápanou jako jeden celek. Na rozdíl od pole jde ovšem o skupinu nehomogenní, jednotlivé složky mohou být různých typů.

Chceme-li zacházet s jednotlivými složkami záznamu, používáme *kvalifikace*: Spolu se jménem proměnné typu záznam uvedeme i jméno složky. Jméno složky vlastně znamená relativní adresu složky vzhledem k počátku proměnné typu záznam.

Složky záznamu jsou uloženy v paměti za sebou, zpravidla v pořadí, ve kterém jsou uvedeny v deklaraci. (Přesněji: některé programovací jazyky vyžadují ukládání složek v daném pořadí, jiné dovolují, aby si překladač stanovil pořadí uložení sám.)

Mezi jednotlivé složky záznamu může překladač vložit prázdná místa, která zajistí, aby uložení jednotlivých složek vyhovovalo požadavkům systému. (Může se např. stát, že procesor vyžaduje, aby proměnné určitých typů začínaly na sudých adresách.) To znamená, že velikost záznamu může být větší než součet velikostí jednotlivých složek.

#### Variantní záznamy (unie)

Variantní část pascalského záznamu (unie v jazyku C) představuje skupinu proměnných, *přeložených přes sebe*. To znamená, že všechny složky začínají na téže adrese a velikost variantní části je rovna velikosti největší složky.

Ke složkám variantních záznamů přistupujeme podobně jako ke složkám „obyčejných“ záznamů.

### 2.1.4 Objekt

Vysvětlení základních pojmů objektově orientovaného programování najdete v kapitole 11..

V definici objektového typu specifikujeme jednak atributy, tedy datové složky, jednak metody, tj. funkční a procedurální složky. Instance, tedy proměnná objektového typu, obsahuje ovšem pouze atributy (navíc pouze atributy instancí).

Atributy třídy jsou vlastně globální proměnné, pouze formálně přidružené k dané třídě. Proto se ukládají obvykle nezávisle na instancích třídy.

Pro ukládání atributů instancí platí podobná pravidla jako pro ukládání složek záznamů.

#### Skryté atributy

Překladač může pro zajištění správné funkce objektů přidat do třídy skryté atributy. Může se jednat jak o atributy třídy tak i o atributy instancí. Tyto atributy nejsou zpravidla uživateli přímo přístupné.

#### Příklad 2.1

Překladače používají skrytých atributů objektových typů poměrně často - např. v souvislosti s polymorfismem (virtuálními metodami).

V Pascalu nebo v C++ zřídí překladač pro každou polymorfní třídu, tj. třídu, která má alespoň jednu virtuální metodu, *tabulku virtuálních metod* (*VMT*). Tato tabulka obsahuje adresy všech virtuálních metod dané třídy a umožňuje efektivně realizovat pozdní vazbu. VMT je zřejmě skrytým atributem třídy.

V každé instanci takové třídy pak bude skrytý atribut instancí, obsahující adresu VMT. Při volání libovolné virtuální metody se nejprve pomocí adresy, uložené v instanci, najde VMT. V ní se pak vyhledá adresa metody, kterou je třeba volat.

Další skryté atributy používají některé překladače C++ v souvislosti s problémy okolo vícenásobného dědictví.[4]

## 2.2 Odvozené datové struktury: seznam a strom

Místo o *odvozených* datových strukturách by nepochybně bylo vhodnější hovořit o *abstraktních* datových strukturách. Protože se však tento pojem používá především pro objektové typy, a datové struktury, o kterých zde bude řeč, nemusí být nutně implementovány jako objekty, budeme raději hovořit o strukturách odvozených.

Mezi nimi zaujímají zvláštní postavení seznamy a stromy, proto s nimi začneme.

### 2.2.1 Seznam

Seznam (anglicky *list*) je datová struktura, která představuje posloupnost složek. Složky jsou uspořádány podle určitého klíče. Jako klíč může sloužit hodnota dat, uložených ve složkách, nebo hodnota funkce, vypočítané na základě těchto dat. Jiné uspořádání seznamu může být dáno pořadím, ve kterém byly složky do seznamu přidávány.

Přestože seznam představuje uspořádanou datovou strukturu, nemusí jednotlivé složky ležet v paměti za sebou. Seznam lze implementovat také tak, že každá složka bude obsahovat odkaz na následující prvek; tento odkaz (ukazatel, označovaný často jako *spojka*, anglicky *link*) bude zajišťovat návaznost složek. První prvek seznamu označujeme jako *hlavu seznamu* (*head*), zbytek seznamu po odtržení hlavy se někdy označuje jako *ohon* (*tail*). Seznam může být i prázdný, nemusí obsahovat žádný prvek.

Seznamy obvykle používáme jako dynamické datové struktury. To znamená, že v programu deklarujeme pouze ukazatel na hlavu seznamu (na obr. 2.2 je to proměnná **unh**); jednotlivé prvky seznamu podle potřeby dynamicky alokujeme nebo rušíme.

Seznamy se (spolu se stromy) označují jako *rekurzivní datové struktury*, neboť každý prvek seznamu obsahuje odkaz na položku stejného typu.

[4] Přesná pravidla pro uložení odkazu na VMT a samotné VMT v Turbo Pascalu najdete ve firemní dokumentaci. Základní informace o skrytých atributech tříd v C++ najdete ve [20]; podrobnosti o Borland C++ najdete např. v [17], část II.

### Jednosměrný seznam

*Jednosměrný seznam* je nejjednodušší variantou seznamu: každý prvek obsahuje pouze odkaz na následující prvek. Prvky jednosměrného seznamu bychom mohli v Pascalu deklarovat následujícím způsobem:

```
type uPrvek = ^Prvek;
     Prvek  =  record
        D:     data;
        Další: uPrvek;
     end;
```

*D* je složka typu *data*, do které budeme ukládat informace. Složka *Další* bude obsahovat ukazatel na další prvek seznamu nebo v případě posledního prvku **nil**.

Při práci s jednosměrným seznamem často používáme *zarážku*: poslední prvek seznamu nenese užitečná data, pouze slouží jako zarážka při prohledávání. Vedle ukazatele na hlavu seznamu si pak uchováváme také ukazatel na tuto zarážku.

### Příklad 2.2

Odvozené datové struktury je výhodné deklarovat jako objektové typy. Následující deklarace objektového typu *seznam* navazuje na deklaraci typu prvek a ukazuje procedury pro vytvoření prázdného seznamu, vložení prvku na konec seznamu a funkci, která vyhledá v seznamu prvek se zadanou hodnotou (pokud jej najde, vrátí jeho adresu, jinak vrátí **nil**).

```
type  seznam = object
        hlava, konec: uPrvek;       {ukazatel na hlavu a zarážku}
        constructor VytvořSeznam;
        procedure VložNaKonec(var dd: data);
        function Vyhledej(var dd: data):uPrvek;
        { ... a další metody ... }
      end;
```

Podívejme se na některé běžné operace se seznamem.

### Vytvoření prázdného seznamu

Prázdný seznam, složený z prvků typu *Prvek*, vytvoříme takto:

1. Deklarujeme ukazatel na hlavu seznamu *hlava* a ukazatel na zarážku *konec* jako proměnné typu *uPrvek* (ukazatel na prvek).

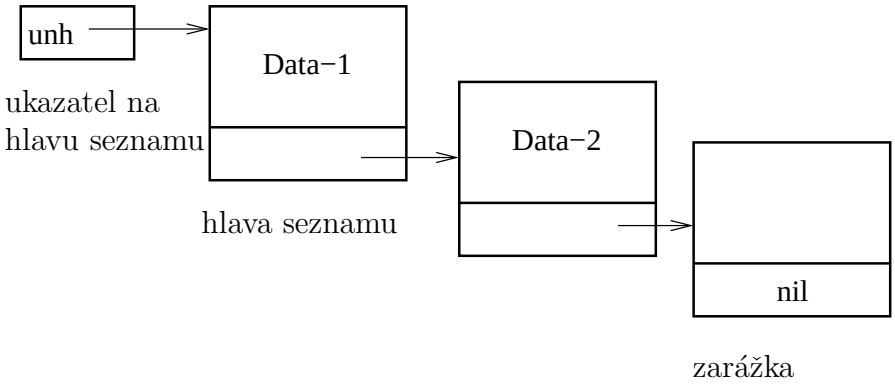
2. Vytvoříme dynamickou proměnnou typu *Prvek* a její adresu uložíme do proměnných *hlava* a *konec*.

3. Do složky *hlava^.Další*, tj. do složky *Další* nově vytvořené dynamické proměnné, vložíme hodnotu **nil**.

### Příklad 2.2 (pokračování)

Metoda *seznam.VytvořSeznam*, která vytvoří prázdný seznam, může vypadat takto:

```
{vytvoří prázdný seznam, obsahující pouze zarážku}
constructor seznam.VytvořSeznam;
begin
    New(hlava);
    konec := hlava;
    hlava^.Další := nil;
end;
```




Obr. 2.2: Jednosměrný seznam se třemi prvky

**Přidání nového prvku na konec seznamu**

Nový prvek, který bude obsahovat hodnotu *dd*, přidáme na konec seznamu pomocí následujícího algoritmu:

1. Požadovaná data vložíme do složky *D* zarážky, tj. do *konec^.D*.
2. Alokujeme nový prvek, tj. novou dynamickou proměnnou typu *Prvek*, a její adresu uložíme do složky *Další* zarážky a do *konec*. Nově alokovaný prvek převezme roli zarážky.
3. Do složky *Další* nové zarážky vložíme hodnotu **nil**.

**Příklad 2.2 (pokračování)**

Podívejme se, jak vypadá implementace metody *seznam.VložNaKonec*:
{vloží prvek s hodnotou dd na konec seznamu}
procedure seznam.VložNaKonec(var dd: data);

```
begin
    konec^.D := dd;                      {vloží data do zarážky}
    New(konec^.Další);                   {alokuje novou zarážku}
    konec := konec^.Další;      {aktualizuje ukazatel na zarážku}
    konec^.Další := nil;
end;
```

**Vyhledání prvku v seznamu**

Často také potřebujeme zjistit, zda je v seznamu prvek, který obsahuje danou hodnotu *dd*. K tomu poslouží následující algoritmus, ve kterém využijeme zarážky:

1. Do pomocné proměnné *p* typu *uPrvek* vložíme ukazatel na hlavu seznamu.
2. Do zarážky vložíme hledanou hodnotu, tj. přiřadíme *konec^.D := dd*;
3. Prohledáváme postupně seznam, dokud nenajdeme prvek, obsahující *dd*. Platí-li *p^.D=dd*, skončíme, jinak do p uložíme adresu následujícího prvku seznamu.
4. Jestliže po skončení obsahuje *p* adresu zarážky, seznam hledaný prvek neobsahuje. V opačném případě obsahuje *p* adresu hledaného prvku.

**Příklad 2.2 (pokračování)**

Při implementaci metody *seznam.Vyhledej* si zjednodušíme život a budeme předpokládat, že pro hodnoty typu *data* smíme používat operátor <>. Většinou to nebude pravda; potřebnou úpravu programu jistě zvládne čtenář sám.

```
{vyhledá prvek s hodnotou dd a vrátí jeho adresu nebo nil}
function seznam.Vyhledej(var dd: data): uPrvek;
var p: uPrvek;
begin
   p := hlava;  {pomocná proměnná na počátku ukazuje na 1. prvek}
   konec^.D := dd;            {uloží hledanou hodnotu do zarážky}
   while p^.D <> dd do p := p^.Další;           {prohledá seznam}
   if p = konec then Vyhledej := nil
   else Vyhledej := p;               {zarážka: dd není v seznamu}
end;
```

Zarážka v seznamu zaručuje, že se prohledávací cyklus zastaví na požadované hodnotě. Kdybychom ji nepoužili, byla by podmínka, ukončující prohledávání seznamu, složitější.

Nyní již můžeme napsané metody vyzkoušet na jednoduchém programu, který vytvoří seznam, uloží do něj nějaké hodnoty a pak je v něm bude hledat. Pro určitost (a také abychom si usnadnili zápis) definujeme *data* jako standardní typ *integer*.

```
type data = integer;
var t: data;
    S: seznam;
    q: uPrvek;
{ ... }
begin
  S.VytvořSeznam;
  { ... }
  S.VložNaKonec(t);
  q := S.Vyhledej(t);
  { ... }
end.
```

### Vložení nového prvku za daný prvek

Předpokládáme, že známe adresu $p$ prvku, za který chceme vložit do jednosměrného seznamu nový prvek. Postup bude jednoduchý; využijeme pomocnou proměnnou $q$ typu *uPrvek* (viz též obr. 1.3):

1. Alokujeme novou proměnnou typu *Prvek* a uložíme do ní potřebná data. Adresa nové proměnné je uložena v proměnné $q$.
2. Do $q\hat{}.Další$ uložíme adresu prvku, který bude v seznamu následovat - tedy obsah $p\hat{}.Další$.
3. Do $p\hat{}.Další$ uložíme adresu nově vloženého prvku, tedy obsah proměnné $q$.

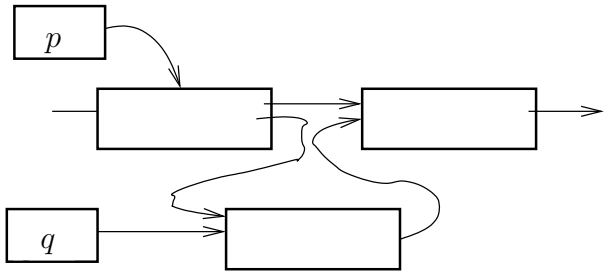
### Příklad 2.2 (pokračování)

Tento algoritmus implementujeme jako metodu *seznam.VložZaPrvek*:

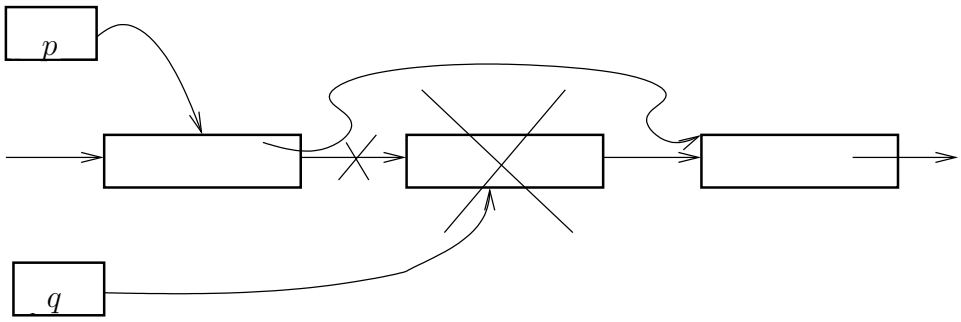
```
{vloží nový prvek s daty dd za prvek, na který ukazuje p}
procedure seznam.VložZaPrvek(p: uPrvek; var dd: data);
var q: uPrvek;
begin
    New(q);
    q^.D := dd;
    q^.Další := p^.Další;
    p^.Další := q;
end;
```

Příklad použití:

```
  t := S.Vyhledej(2);
  if t <> nil then S.VložZaPrvek(t,3);
```




Obr. 2.3: Vložení nového prvku na dané místo v seznamu




Obr. 2.4: Smazání prvku seznamu za označeným prvkem

### Vymazání prvku za daným prvkem

Chceme ze seznamu odstranit prvek, který *následuje za prvkem* s adresou, uloženou v proměnné *p*. K tomu budeme opět potřebovat pomocnou proměnnou *q*. Postup bude následující:

1. Do *q* uložíme *p^.Další* (adresu mazaného prvku).
2. Do *p^.Další* uložíme *p^.Další^.Další* (adresu prvku, který leží za mazaným prvkem).
3. Zrušíme prvek, na který ukazuje *q*.

### Příklad 2.2 (pokračování)

Tento algoritmus implementujeme jako metodu *seznam.SmažZa*:

```
{smaže prvek seznamu, který leží za prvkem s adresou p}
procedure seznam.SmažZa(p: uPrvek);
var q: uPrvek;
begin
    q := p^.Další;
    p^.Další := p^.Další^.Další;
    Dispose(q);
end;
```

Obrázek 2.4 naznačuje, co se při mazání prvku jednosměrného seznamu děje.

### Smazání daného prvku

Nyní chceme smazat prvek, na který ukazuje ukazatel *p*. Tato úloha se může na první pohled zdát složitější, nebot' neznáme prvek *před* tím, který chceme smazat. Jak tedy „napojit“ předcházející a následující prvek?

Velice jednoduše. Využijeme toho, že umíme smazat prvek, který leží *za* označeným prvkem. Protože na datech v mazaném prvku nezáleží, přesuneme do něj obsah prvku následujícího a ten pak smažeme:

1. Přesuneme obsah *p^.Další^.D* do *p^.D* (nyní prvky *p^* a *p^.Další^* obsahují táž data).
2. Smažeme prvek s adresou *p^.Další*.

**Příklad 2.2 (pokračování)**

Při implementaci se odvoláme na metodu *seznam.SmažZa*.

```
{smaže prvek seznamu, na který ukazuje p}
procedure seznam.SmažTen(p: uPrvek);
begin
    p^.D := p^.Další^.D;
    SmažZa(p);
end;
```

**Smazání celého seznamu**

Jakmile přestaneme seznam používat, je třeba ho smazat, uvolnit dynamicky přidělovanou paměť. Algoritmus mazání je velice jednoduchý. Připomeňme si, že seznam je buď prázdný nebo se skládá z hlavy (prvního prvku), za který je připojen ohon (což je zase seznam - tedy buď prázdný, nebo hlava + ohon atd..). Algoritmus mazání seznamu bude vycházet z tohoto popisu a bude rekurzivní:

1. Je-li seznam prázdný, konec. Jinak ulož adresu hlavy do pomocné proměnné.
2. Do ukazatele na hlavu vlož adresu hlavy ohonu; smaž hlavu.
3. Smaž ohon (tedy seznam, který zbyl po smazání hlavy).

**Příklad 2.2 (dokončení)**

Zrušení seznamu je typická úloha pro destruktor objektového typu. Destruktor typu *seznam* může vypadat takto (při implementaci se tentokrát rekurzi vyhneme):

```
{uvolní všechnu dynamicky alokovanou paměť}
destructor seznam.ZrušSeznam;
var q: uPrvek;
begin
   while hlava <> nil do begin
      q := hlava;
      hlava := hlava^.Další;
      dispose(q);
   end;
end;
```

Návrh dalších operací s jednosměrnými seznamy ponecháváme čtenáři.

**Jiné typy seznamů**

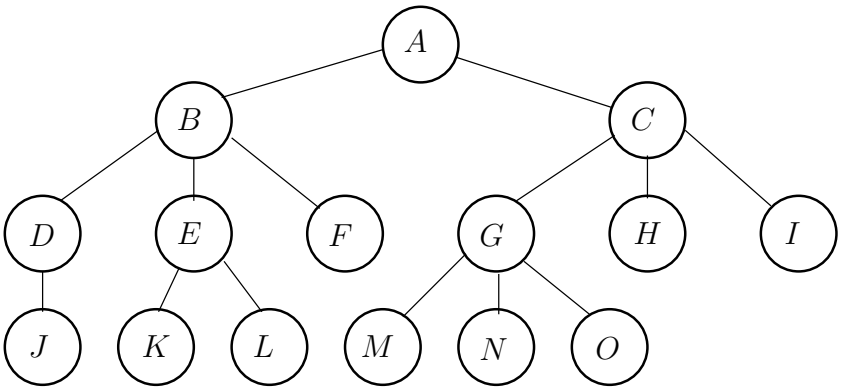
Někdy se setkáme se *dvousměrnými seznamy* (anglicky *double-linked list*). Od jednosměrných seznamů se liší tím, že každý prvek obsahuje odkaz nejen na následující prvek, ale i na prvek předcházející.

Dvousměrný seznam lze snadno procházet v obou směrech - jak od hlavy k poslednímu prvku tak i naopak. Prvkem takového seznamu může být struktura tvaru

```
type uPrvek2 = ^Prvek2;
     Prvek2  = record
        D:    data;
        Předchozí, Následující: uPrvek2;
     end;
```

Základní operace se dvousměrným seznamem jsou podobné operacím s jednosměrným seznamem, proto se jimi nebudeme podrobněji zabývat.

Další typ seznamu, se kterým se můžeme setkat, je *kruhový seznam*. Může jít o kruhový seznam jednosměrný i dvousměrný. V kruhovém seznamu se obvykle nepoužívá zarážka, místo toho poslední prvek obsahuje odkaz na první prvek (pokud jde o seznam dvousměrný, tak také první prvek obsahuje odkaz na poslední prvek).




Obr. 2.5: Příklad ternárního srtomu

### 2.2.2 Strom

Strom je další z běžně používaných rekurzivních datových struktur. Abstraktní definice stromu se základním typem **T** je rekurzivní a vypadá takto:

*Strom* se základním typem **T** je buď prázdná struktura nebo prvek typu **T**, na který je připojen konečný počet disjunktních stromových struktur se základním typem **T** (označujeme je jako *podstromy*).

Prvky stromu se obvykle označují jako *vrcholy* nebo *uzly*. Vrchol, ke kterému není připojen žádný podstrom, označujeme jako *koncový vrchol* nebo *list*. Vrchol, který sám není připojen k žádnému jinému vrcholu, označujeme jako *kořen*. To znamená, že kořen spolu se všemi podstromy, které jsou k němu připojeny, tvoří celý strom. Prvky, které nejsou listy, označujeme jako *vnitřní vrcholy* stromu.

Je-li $G$ kořen podstromu, připojeného k uzlu $C$ (viz obr. 2.5), říkáme také, že $G$ je (přímým) následovníkem $C$ a $C$ je (přímým) předchůdcem **G**. Kořen je vrchol, který nemá předchůdce; list je vrchol, který nemá žádného následovníka.

Přidáme k definici stromu požadavek, aby počet podstromů, připojený k libovolnému z vrcholů daného stromu, nepřesáhl $n$. Takový strom označujeme jako $n$-ární. Nejčastěji se setkáme s binárními stromy ($n = 2$) nebo ternárními stromy ($n = 3$). Číslo $n$ („-aritu“) budeme označovat jako *typ stromu*.

**Příklad 2.3**

Na obrázku 2.5 vidíte příklad ternárního stromu s vrcholy označenými $A, \ldots, O$.[5] Vrchol $A$ je kořen tohoto stromu, vrcholy $J$, $K$, $L$, $F$, $M$, $N$, $O$, $H$ a $I$ jsou listy.

Strom jsme znázornili jako graf, jehož uzly odpovídají vrcholům stromu a hrany odpovídají odkazům na připojené podstromy. Jeden takový podstrom, připojený k uzlu $C$, se skládá z uzlů $G$, $M$, $N$, $O$.

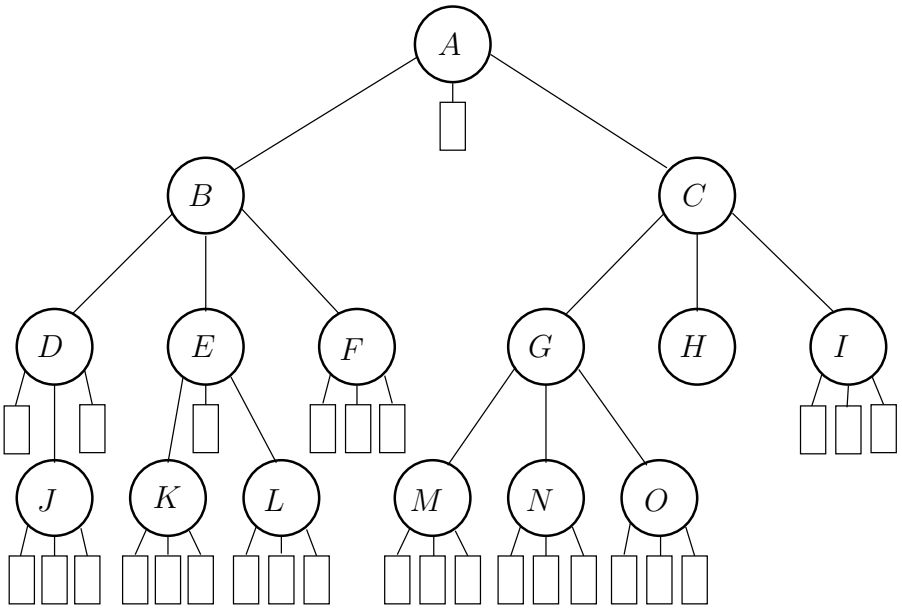
Všimněte si, že v tomto grafu jsme znázornili i odkaz na kořen stromu (jako hranu vedoucí do $A$).
Následující pojmy využijeme při úvahách o složitosti algoritmů, které využívají stromů:

O kořeni stromu říkáme, že je na první úrovni. Uzly, které jsou následovníky kořene, jsou na druhé úrovni. Obecně je-li uzel $S$ na úrovni $i$ a uzel $\check{S}$ je následovníkem $S$, je $\check{S}$ na úrovni $i + 1$.

Je-li $x$ úroveň vrcholu $X$, znamená to, že chceme-li projít stromem od kořene k $X$, musíme projít $x$ hran (včetně hrany, která vstupuje do kořene). Proto se místo o úrovni vrcholu často hovoří o *délce vnitřní cesty daného vrcholu*.

Součet délek cest všech uzlů v daném stromě se nazývá *délka vnitřní cesty stromu*. Průměrná délka vnitřní cesty stromu je pak definována vztahem

[5] Stromy se zpravidla zobrazují s kořenem jako nejvyšším vrcholem a s listy dole. Pokud vám to připadá nelogické, máte jistě pravdu - stromy obvykle rostou obráceně. Můžete s tím nesouhlasit, ale to je asi tak vše, co s tím můžete dělat (J. Cimrman).




Obr. 2.6: Strom z obr. 2.5 s přidanými zvláštními vrcholy

$$P_I = \frac{1}{n} \sum_i n_i i,$$

kde $n_i$ je počet uzlů na $i$-té úrovni stromu.

Kromě délky vnitřní cesty zavádíme také *délku vnější cesty stromu*. Než ji ale definujeme, musíme zavést *zvláštní vrcholy stromu* (na obr. 2.6 jsou znázorněny pomocí čtverečků). V $n$-árním stromu doplníme počet následovníků každého „obyčejného“ vrcholu zvláštními vrcholy na hodnotu $n$. Zvláštní vrcholy nebudou mít žádné následovníky.

Délku vnější cesty stromu definujeme jako součet délek cest všech zvláštních vrcholů. Průměrná délka vnější cesty stromu je

$$P_E = \frac{1}{m} \sum_i m_i i, \tag{2.1}$$

kde $m_i$ je počet zvláštních vrcholů na $i$-té úrovni stromu a $m$ je celkový počet zvláštních vrcholů.
Počet $m$ zvláštních vrcholů, které musíme do stromu přidat, závisí na typu stromu, vyjádřeném číslem $d$, a na počtu „původních“ vrcholů n. Protože do každého vrcholu rozšířeného stromu vstupuje právě jedna hrana, je v něm celkem $m + n$ hran. Protože z každého původního vrcholu vystupuje vždy $d$ hran, zatímco ze speciálních vrcholů žádné hrany nevystupují, obsahuje strom celkem $dn$ hran, vystupujících z vrcholů, a navíc jednu hranu, která vstupuje do kořene (ta nevychází ze žádného z vrcholů). Z těchto úvah dostaneme pro číslo $m$ rovnici

$$dn + 1 = m + n, \qquad \text{tj.} \qquad m = (d-1)\, n + 1.$$

Maximální počet vrcholů ve stromu typu $d$ s $k$ úrovněmi je roven součtu

$$N_{max}\,(d,k) = 1 + d + d^2 + \cdots + d^{k-1} = \sum_{i=1}^{k} d^{k-1} = \frac{d^k - 1}{d-1},$$

neboť na první úrovni je nejvýše jeden vrchol, který má nejvýše $d$ následovníků na 2. úrovni, z nichž každý má opět nejvýše d následovníků na 3. úrovni atd.

Speciálně binární strom s k úrovněmi má nejvýše $N_{max}\,(2,k) = 2^k - 1$ uzlů.

Stromy nejčastěji reprezentujeme jako dynamické datové struktury. Vrcholy stromu typu $d$ mohou být záznamy, které obsahují ukazatele na kořeny připojených podstromů (nebo **nil**, není-li podstrom připojen ):

```
type uVrchol = ^vrchol;
     vrchol  = record
       Dt: data;
       Další: array [1..d] of uVrchol;
     end;
```

Užitečné informace jsou uloženy ve složce *Dt*, která je typu *data* (podobně jako v případě seznamů).

Poznamenejme, že na jednosměrný seznam se můžeme dívat jako na unární strom (strom typu 1).

**Binární stromy**

V binárním stromu jsou ke každému vrcholu připojeny dva podstromy (jeden nebo oba mohou být prázdné). Jeden z nich označíme jako levý a druhý jako pravý podstrom.[6]

Nejčastěji se setkáme s binárními stromy, ve kterých jsou data uspořádána podle následujícího pravidla: *Pro každý vrchol U platí, že všechny údaje, uložené v levém podstromu, připojeném k U, jsou menší, než je údaj uložený v U, a všechny údaje, uložené v pravém podstromu, připojeném k U, jsou větší, než je údaj uložený v U.*

Pokud nezdůrazníme něco jiného, budeme dále pod označením *binární strom* rozumět takto uspořádaný binární strom. Základem implementace binárního stromu bude záznam *vrchol*, definovaný takto:

```
type uVrchol = ^vrchol;
     vrchol  = record
       Dt: data;
       Levý, Pravý: uVrchol;
     end;
```

Binární strom je obvykle přístupný pomocí ukazatele na kořen. Vrcholy zpravidla alokujeme dynamicky.

Také u stromů se někdy používá zarážka: všechny „prázdné“ ukazatele na následovníky obsahují místo hodnoty **nil** adresu pevně stanoveného prvku - zarážky. Jde o analogii zvláštních uzlů, kterých jsme použili při definice vnější cesty stromu; zarážka je ale jen jedna, společná pro celý strom (viz obr. 2.7). Data uložená v zarážce jsou samozřejmě bezvýznamná.

Adresu zarážky, podobně jako adresu kořene, ukládáme do zvláštní proměnné.

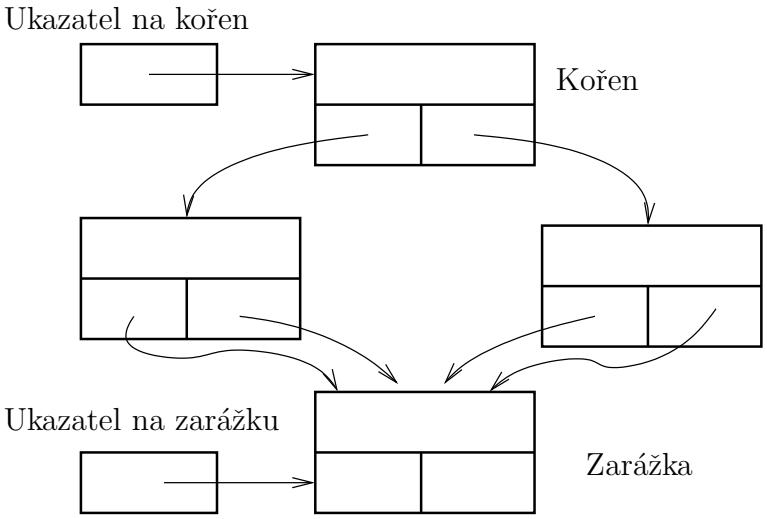
Při procházení binárního stromu se obvykle používá některá z metod, označovaných jako *přímé zpracování* (anglicky *preorder*), *vnitřní zpracování* (*inorder*) a *zpětné zpracování* (*postorder*). Při přímém zpracování nejprve zpracujeme data, uložená v kořeni, pak zpracujme levý podstrom a nakonec pravý podstrom. Při vnitřním zpracování nejprve projdeme levý podstrom, pak zpracujeme kořen a nakonec projdeme pravý podstrom; při zpětném zpracování postupujeme v pořadí levý podstrom, pravý podstrom, kořen.

Předpokládejme, že pro zpracování dat v uzlu s adresou $t$ použijeme proceduru $P(t)$. Potom procedura pro zpracování celého stromu přímou metodou bude mít tvar

```
procedure preorder(kořen: uVrchol);
begin
  if t <> nil then begin
     P(t);                {zpracování kořene}
     preorder(t^.Levý);   {zpracování levého podstromu}
     preorder(t^.Pravý);  {zpracování pravého podstromu}
  end;
end;
```

Je zřejmé, že procedury, založené na zbývajících dvou metodách, se budou lišit pouze pořadím příkazů, označených komentáři. Čtenář je jistě snadno sestaví sám.

[6] Tím vlastně definujeme uspořádání podstromů, a to bez ohledu na data, která obsahují. Poznamenejme, že při grafickém znázornění budeme opravdu kreslit levý podstrom vlevo a pravý vpravo.




Obr. 2.7: Binární strom se zarážkou

### Základní operace s binárním stromem

V tomto oddílu se seznámíme jen s nejelementárnějšími operacemi s binárními stromy; použití těchto struktur věnujeme později samostatnou kapitolu. Zde si povíme, jak vytvořit binární strom, jak do něj přidat vrchol, jak zjistit, zda je ve stromě vrchol se zadanou hodnotou, jak zrušit nalezený vrchol a jak zrušit celý strom.

Tyto algoritmy formulujeme pro binární stromy bez zarážky; jejich tvar pro stromy se zarážkou si jistě odvodí čtenář sám.

### Vytvoření binárního stromu

Vytvoříme samozřejmě prázdný binární strom. Strom se skládá ze záznamů (struktur) typu vrchol; v programu tedy musíme mít ukazatel na kořen stromu, proměnnou typu ukazatel na vrchol. Vytvoření prázdného stromu potom spočívá v přiřazení hodnoty **nil** (**NULL** v C++) této proměnné.

### Příklad 2.4

Také *strom* deklarujeme jako objektový typ; tentokrát ale použijeme jazyk C++. Deklarace typu *vrchol* vznikne přepisem pascalské deklarace, kterou jsme si už uvedli:

```
typedef struct vrchol *uVrchol;
struct vrchol {
  data Dt;
  uVrchol Levý, Pravý;
};
```

Objektový typ (třída) *strom* bude mít jediný soukromý atribut, ukazatel na kořen stromu *uKořen*. (Pokud bychom chtěli strom se zarážkou, přibyl by ukazatel za ni.)

Rozhraní třídy *strom* se bude skládat z veřejně přístupného konstruktoru, destruktoru (je nezbytný, neboť v instancích alokujeme dynamickou paměť a destruktor se musí postarat o její uvolnění) a metody pro vložení vyhledání a smazání vrcholu. Další pomocné metody deklarujeme jako soukromé, tj. budou je smět používat pouze metody třídy *strom*.

```
class strom {
  uVrchol uKořen;  // ukazatel na kořen stromu
  // pomocné soukromé metody
  uVrchol novýVrchol(data &dd);
  void vložDoStromu(uVrchol &t, data &dd);
  uVrchol najdi(data &dd, uVrchol t, uVrchol &d);
  void smažList(uVrchol t, uVrchol předch);
```

```
  void smažVeVětvi(uVrchol t, uVrchol předch);
  void smažStrom(uVrchol &t);
 // rozhraní třídy strom
 public:
  strom();            // konstruktor
  ~strom();           // destruktor
  void vlož(data &dd); // vložení vrcholu se zadanými daty
  uVrchol hledej(data &dd, uVrchol &d);  // vyhledání vrcholu
  void smaž(data &dd); // smazání vrcholu se zadanými daty
};
```

Nový strom vytvoříme buď deklarací nebo tím, že alokujeme dynamickou proměnnou typu *strom*. V obou případech se automaticky zavolá konstruktor. Jeho jediným úkolem bude vložit do ukazatele na kořen hodnotu **NULL**, která informuje uživatele, že strom je prázdný:

```
strom::strom() {
  uKořen = NULL;
}
```

### Zrušení binárního stromu

Algoritmus pro smazání celého binárního stromu lze odvodit - podobně jako u seznamu - z definice: strom je buď prázdný (a není co mazat), nebo je to vrchol, ke kterému je připojen levý a pravý podstrom (což je strom). Protože jde o rekurzivní popis, jistě nebudeme překvapeni, že i algoritmus bude rekurzivní. Známe-li ukazatel $t$ na kořen, můžeme strom smazat takto:

1. Je-li strom, na který $t$ ukazuje, prázdný, skončíme.
2. Smažeme levý podstrom, připojený k vrcholu, na který ukazuje $t$.
3. Smažeme pravý podstrom, připojený k vrcholu, na který ukazuje $t$.
4. Smažeme vrchol, na který ukazuje $t$.

### Příklad 2.4 (pokračování)

Smazání celého stromu je typická úloha pro destruktor. Protože ale nelze vyloučit, že budeme chtít smazat celý strom i jindy, naprogramujeme tuto operaci jako samostatnou metodu, kterou bude destruktor volat.

```
strom::~strom() {
 smažStrom(uKořen);
}
```

Metoda *smažStrom* implementuje popsaný rekurzivní algoritmus; umožňuje smazat nejen celý strom, ale i libovolný podstrom. Parametrem, který mazaný (pod)strom určuje, je ukazatel na jeho kořen. Do tohoto ukazatele vloží metoda *smažStrom* hodnotu **NULL**; proto jej předáváme odkazem. V definici třídy *strom* jsme tuto metodu uvedli jako soukromou; možná, že se později rozhodneme ji zveřejnit - přemístit ji do sekce **public**.

```
void strom::smažStrom(uVrchol &t) {
 if(t) {
    smažStrom(t->Levý);
    smažStrom(t->Pravý);
    delete t;
    t = NULL;
 }
}
```

### Vložení nového vrcholu

Dostali jsme data *dd* a chceme je zařadit do stromu. Pokud tam takový údaj ještě není, přidáme do stromu nový vrchol, jinak není třeba provádět vůbec nic. Přidáváme-li vrchol do prázdného stromu, alokujeme pro něj paměť, uložíme do něj potřebná data a adresu vrcholu uložíme do ukazatele na kořen.

V neprázdném stromě vyjdeme od kořene a porovnáme *dd* s hodnotou v něm. Je-li *dd* menší než uložená hodnota, zařadíme nový vrchol do levého podstromu, jinak jej zařadíme do pravého podstromu.

**Příklad 2.4 (pokračování)**

Vytvoření nového vrcholu budeme potřebovat na několika místech. Navíc obsahuje operaci, která se nemusí podařit (alokaci paměti). Proto bude rozumné definovat ji jako samostatnou funkci, která bude vracet adresu nového vrcholu.

```
uVrchol strom::novýVrchol(data &dd) {
 uVrchol q = new vrchol;
 if(!q){
  Chyba();
  return NULL;
 } else {
  q -> Dt = dd;
  q -> Levý = q -> Pravý = NULL;
  return q;
 }
}
```

Pokud se alokace paměti nepodaří, voláme funkci *Chyba* (); její definici si můžete doplnit podle potřeby.

Při vkládání nového uzlu do stromu potřebujeme strom rekurzivně prohledat. Přitom vyjdeme od ukazatele na kořen a budeme pokračovat přes ukazatele na levý či pravý podstrom. Pokud v některém z vrcholů najdeme hodnotu *dd*, skončíme - náš údaj tam již je a není třeba přidávat další uzel.

Pokud v daném uzlu náš údaj není, je třeba jej vložit do levého nebo pravého podstromu, určeného opět ukazatelem na kořen.

Je zřejmé, že procedura pro vložení údaje do stromu bude rekurzivní. Jejími parametry budou jednak vkládaná hodnota a jednak ukazatel na kořen (pod)stromu, do kterého ji chceme vložit.

Přitom vycházíme od ukazatele na kořen celého stromu - tedy od soukromého atributu. Na druhé straně není třeba, aby uživatel věděl cokoli o kořeni stromu; jeho zájem je vložit do daného stromu určitou hodnotu a tím to končí.

Proto v rozhraní třídy definujeme (veřejně přístupnou) metodu *vlož*, jejímž jediným parametrem bude vkládaná hodnota *dd*. Ta zavolá soukromou metodu *vložDoStromu*, která teprve implementuje algoritmus vkládání.

```
void strom::vlož(data &dd) {
  vložDoStromu(uKořen, &dd);
}
```

Oba parametry metody *vložDoStromu* budeme předávat odkazem. Adresu kořene proto, že ji v některých případech chceme měnit, a data kvůli úspoře místa na zásobníku při rekurzivním volání.

```
void strom::vložDoStromu(uVrchol &t, data &dd) {
  if(!t) t = novýVrchol(dd);  // je-li strom prázdný
  else {
   if(dd == t->Dt) return; // dd tam už je
   if(dd < t->Dt) vložDoStromu(t->Levý, dd);
   else vložDoStromu(t->Pravý, dd);
  }
}
```

**Vyhledání údaje ve stromu**

Chceme zjistit, zda daný strom obsahuje hodnotu *dd*. To znamená, že opět rekurzivně projdeme daný strom.

1. Je-li strom prázdný, hledaný údaj v něm není; konec.
2. Jinak porovnáme *dd* s hodnotou v kořeni; jsou-li si rovny, nalezli jsme hledaný údaj; konec.

3. Jinak je-li *dd* menší než hodnota, uložená v kořeni, prohledáme levý podstrom, připojený ke kořeni - provedeme s ním tento algoritmus a skončíme.

4. Jinak prohledáme pravý podstrom a skončíme.

**Příklad 2.4 (pokračování)**

Vyhledávání dané hodnoty ve stromu implementujeme jako funkci, která v případě úspěchu vrátí adresu nalezeného uzlu. V případě neúspěchu (žádný uzel neobsahuje *dd*) vrátí **NULL**.

Často je třeba znát také adresu předchůdce nalezeného uzlu (to oceníme zejména při rušení uzlů). To algoritmus nijak nezkomplikuje - než se dostaneme do hledaného uzlu, musíme projít přes jeho předchůdce. Stačí si tedy jeho adresu zapamatovat v pomocné proměnné. Proto bude mít funkce *strom::hledej* také parametr *d*, předávaný odkazem, ve kterém bude vracet právě adresu předchůdce.

Poznamenejme, že kořen nemá předchůdce. Je-li hledaný údaj uložen v kořeni celého stromu, vrátí funkce *strom::hledej* v d hodnotu **NULL**.

Při dalším rozboru zjistíme, že k prohledání stromu potřebujeme adresu jeho kořene. Z podobných důvodů jako při vkládání do stromu proto bude veřejně přístupná metoda *hledej* volat soukromou metodu *najdi*, která bude mít o jeden parametr navíc. Metoda *hledej* také zajistí inicializaci pomocného parametru *d*.

```
uVrchol strom::hledej(data &dd, uVrchol &d) {
  d = NULL;
  return najdi(dd, uKořen, d);
}
```

Metoda *najdi* teprve implementuje popsaný algoritmus:

```
uVrchol strom::najdi(data &dd, uVrchol t, uVrchol &d) {
  if(!t) return NULL;  // prázdný strom: není tam
  if(t->Dt == dd) return t;  // našli jsme ji
  d = t;  // jdeme dál: adresa předka
  if(dd < t->Dt) return najdi(dd, t->Levý, d);  // prohledej
  else return najdi(dd, t->Pravý, d);           // podstromy
}
```

**Zrušení vrcholu**

Zrušení jednotlivého vrcholu, jeho odstranění ze stromu, je patrně nejkomplikovanější ze základních operací nad binárním stromem. Budeme totiž muset rozlišit několik případů (viz též obr. 2.8 a 2.9):

1. *Rušíme list.* V takovém případě uvolníme dynamickou paměť, přidělenou tomuto uzlu, a odstraníme odkaz na rušený vrchol v jeho předchůdci (nebo v ukazateli na kořen, jestliže měl strom jen jeden vrchol).

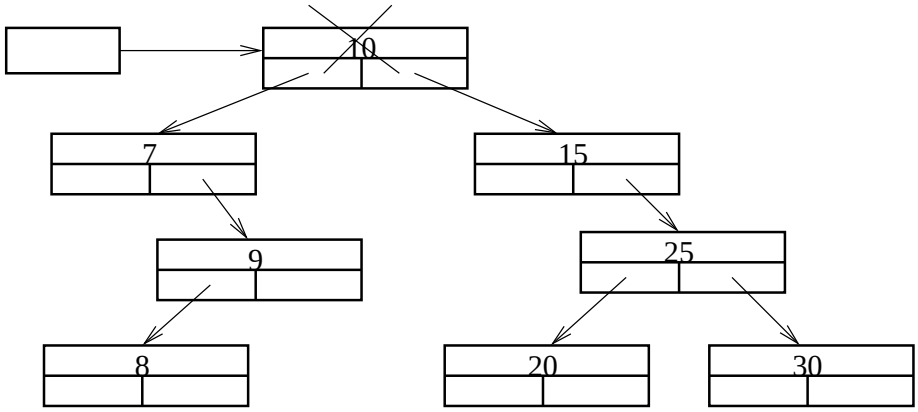
2. *Rušíme vrchol, který má jen jednoho následovníka.* Jde o podobnou situaci jako při rušení prvku seznamu. Adresu rušeného prvku uložíme do pomocné proměnné, odkaz v předchůdci upravíme tak, aby ukazoval na následovníka a vrchol zrušíme.

3. *Rušíme vrchol, který má dva následovníky.* Zde vzniká problém, čím zrušený vrchol nahradit. Kromě hodnoty *dd*, která nás již nezajímá, obsahuje totiž také odkazy na své následovníky, které je třeba uchovat. Navíc výsledkem této operace musí být opět uspořádaný binární strom.

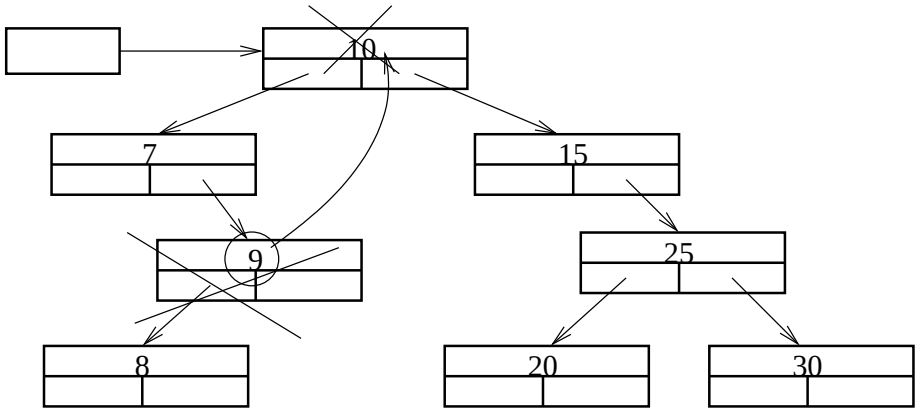
V tomto případě se používá následujícího triku:

1. Najdeme nejpravější vrchol levého podstromu, připojeného k rušenému vrcholu (označíme jej $Q$; to je vrchol, v němž je uložena největší z hodnot, menších než v rušeném vrcholu). Nalezený vrchol má nejvýše jednoho následovníka (jinak by nemohl být nejpravější).

2. Hodnotu z vrcholu $Q$ přeneseme do vrcholu, který chceme zrušit.




Obr. 2.8: Rušení listu a vrcholu s jediným následovníkem




Obr. 2.9: Rušení vrcholu se dvěma následovníky (zde kořene)

3. Zrušíme vrchol $Q$.

Místo nejpravějšího vrcholu v levém podstromu (tedy největší hodnoty, menší než je rušená) můžeme také použít nejlevějšího vrcholu v pravém podstromu (tedy nejmenší hodnoty, která je větší než hodnota v rušeném vrcholu).
Lze samozřejmě navrhnout i jiné postupy; ten, který jsme zde uvedli, však patří k nejčastěji používaným.

**Příklad 2.4 (pokračování)**

Nyní navrhneme metodu pro smazání vrcholu ve stromě. Použijeme v ní tři pomocné proměnné, neboť v nejhorším případě budeme potřebovat ukazatel na rušený vrchol, na nejpravější vrchol levého podstromu a na jeho předchůdce.

Metoda nejprve vyhledá prvek, který chceme zrušit. Pokud jej nenajde, skončí.

Dále zjistí, zda jde o list nebo zda má jediného následovníka a pokud je některá z těchto podmínek splněna, zavolá pomocnou soukromou metodu *smažList* resp. *smažVeVětvi*.

Pokud má dva následovníky, vyhledá nejpravější vrchol levého podstromu a smaže jej. K tomu použije metody pro smazání listu nebo smazání vrcholu s jediným následovníkem.

```
void strom::smaž(data &dd) { // smaže vrchol
  uVrchol t,d,q;             // pomocné proměnné
  t = hledej(dd, d);         // hledání mazaného vrcholu
  if(!t) return;             // pokud tam není, konec
  // je to list
  if(t->Levý==NULL && t->Pravý == NULL) smažList(t, d);
```

```
  else
    // má jen jednoho následovníka
    if(t->Levý == NULL || t->Pravý == NULL) smažVeVětvi(t, d);
    else {
      // má 2 následovníky
      // najdi nejpravějšího levého následovníka
      q = t->Levý;
      d = t;
      while(q->Pravý) {
         d = q;
         q = q->Pravý;
      }
      // přesuň data z *q do *t
      t->Dt = q->Dt;
      // smaž *q;
      if(q->Levý || q->Pravý) smažVeVětvi(q, d);
      else smažList(q, d);
    }
}
```

Při hledání vrcholu, který doopravdy smažeme, jsme se tentokrát vyhnuli rekurzi a použili jsme cyklu **while**.

Metoda pro smazání listu musí rozlišit případ, že mažeme kořen, a případ, že mazaný list má předchůdce. K tomu nám poslouží adresa předchůdce, kterou nám poskytla metoda *hledej*; je uložena v parametru *předch*. (Připomeňme si, že v případě kořene vrátí funkce *strom::hledej* jako adresu předchůdce hodnotu **NULL**.)

```
// smaže list stromu; t je adresa mazaného vrcholu,
// predch je adresa předchůdce
void strom::smažList(uVrchol t, uVrchol předch) {
  if(předch) {   // není to kořen
     if(předch->Levý == t) předch ->Levý = NULL;
     else předch ->Pravý = NULL;
  } else {       // je to kořen
     uKořen = NULL;
  delete t;
  }
}
```

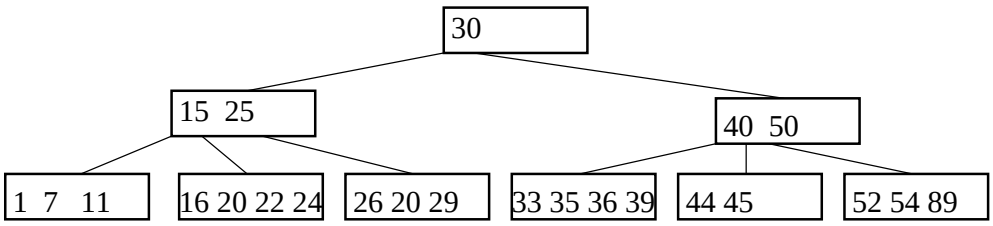
Metoda pro smazání vrcholu s jedním následovníkem je jen nepatrně složitější než metoda pro smazání prvku seznamu. Musí rozlišit, zda byl mazaný prvek levým nebo pravým následovníkem svého předchůdce. Také v případě, že jde o kořen, se bude postup poněkud lišit.

```
void strom::smažVeVětvi(uVrchol t, uVrchol předch) {
  uVrchol q;       // q je adresa následovníka
  q = (t->Levý)?t->Levý:t->Pravý;
  if(!předch) uKořen = q; // je to kořen?
        else (předch->Levý == t?      // ***
              předch->Levý: předch->Pravý) = q;
  delete t;
}
```

V pomocné proměnné *q* jsme si uložili adresu následovníka.[7] Podívejme se ještě na jednoduchý příklad použití objektového typu *strom*:

```
void pokusy() {
  strom S;   // vytvoří prázdný strom
```

[7]Poznámka pro čtenáře, kteří neznají jemnosti jazyka C++: Operátor podmíněného výrazu : vytváří v C++ l-hodnotu. To znamená, že jej můžeme použít i na levé straně přiřazovacího výrazu - jak jsme si to dovolili v předposledním příkazu této metody, označeném třemi hvězdičkami. Význam: Hodnotu **q** uložíme do ukazatele na levého nebo pravého následovníka, podle toho, který z nich byl nenulový.




Obr. 2.10: B-strom druhého řádu
Operace s b-stromem $n$-tého řádu

```
  S.vlož(10); S.vlož(5); S.vlož(3);  // vložíme nějaké prvky
  S.vlož(7);  S.vlož(17);S.vlož(17);
  S.smaž(10);  // a zase je smažeme
}
```

Destruktor se v C++ volá automaticky při zániku instance. To znamená, že při ukončení funkce *pokusy* se celý strom smaže, aniž se o to musíme starat.

## 2.3 Další odvozené datové struktury

### 2.3.1 B-strom

B-strom (*b-tree*) je datová struktura podobná stromu; jeho vrcholy se nazývají *stránky*. Pro každý b-strom *řádu n* platí, že:

1. Každá stránka obsahuje maximálně $2n$ položek (uložených údajů).
2. Každá stránka - kromě kořenové - obsahuje minimálně $n$ položek. Kořenová stránka obsahuje alespoň jednu položku.
3. Každá stránka je buď listovou stránkou, tj. nemá žádné následovníky, nebo má $m$+1 následovníků, kde $m$ je počet položek v ní uložených.
4. Všechny listové stránky jsou na stejné úrovni.

Z této definice plyne, že všechny větve b-stromů budou - na rozdíl od „klasických“ stromů - stejně dlouhé. To znamená, že práce s b-stromy může být podstatně efektivnější než práce s klasickými stromy. Navíc lze často volit velikost stránky tak, aby odpovídala jednomu sektoru na disku, a tak lze podstatně zefektivnit využití diskového prostoru a zrychlit práci s uloženými daty.

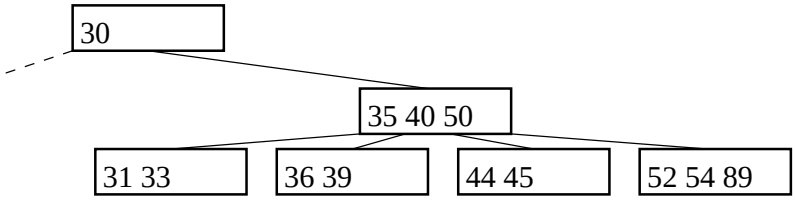
Poměrně často se setkáme s b-stromy 1. řádu, jejichž stránky obsahují 1 nebo 2 položky. Takovéto b-stromy se často nazývají 2-3-stromy, neboť každá stránka kromě listů má 2 - 3 následovníky. Setkáme se také s názvy *binární b-stromy* nebo *bb-stromy*.

Podívejme se na stránku $S$, která obsahuje $m$ položek a která není listová. K ní je připojeno $m$+1 podstromů[8], které označíme jako nultý, první,..., $m$-tý. Data, uložená v b-stromu, jsou uspořádána obvykle tak, že všechny hodnoty, uložené nultém podstromu, jsou menší, než první hodnota uložená ve stránce $S$. Všechny hodnoty, uložené v prvním podstromu, jsou větší, než první prvek ve stránce $S$ a menší, než druhý prvek v $S$,..., a všechny hodnoty, uložené v $m$-tém podstromu, jsou větší, než $m$-tý prvek stránky $S$.
Na obrázku 2.10 vidíme b-strom druhého řádu. Jeho stránky tedy obsahují 2 - 4 údaje.

#### Přidání údaje do b-stromu

Přidání nové položky do b-stromu je v podstatě přímočaré. Podobně jako ve stromu vyhledáme listovou stránku, do které nový údaj patří. Pokud je v ní volné místo (obsahuje méně než $2n$ údajů), vložíme nový údaj a skončíme.

[8]Budeme hovořit o *podstromech*, i když jde o b-strom. Termín *pod-b-strom* by byl sice přesnější, mně se ale naprosto nelíbí.




Obr. 2.11: Po vložení čísla 31 do b-stromu na obr. 2.10

Je-li však stránka již zaplněná, obsahovala by po přidání nového prvku $2n+1$ prvků. Nový prvek tedy zařadíme na správné místo, takto vzniklou „přeplněnou“ stránku rozdělíme na dvě po $n$ prvcích a prostřední prvek přesuneme do předchůdce. Jestliže stránka předchůdce nemá, vytvoříme ho.

V předchůdci se může situace opakovat. Odtud plyne, že b-strom roste vlastně pouze tak, že se rozdělí kořenová stránka.

**Příklad 2.5**

Do stromu na obr. 2.10 vložíme údaj 10. Snadno zjistíme, že patří do nejlevějšího listu mezi položky 7 a 11. Protože v této stránce je volné místo, vložíme jej a skončíme.

Dále chceme přidat číslo 31. To zřejmě patří do 4. listu zleva před číslo 33. Tento list je ale již plný. Jestliže sem číslo 31 přesto formálně přidáme, dostaneme stránku s prvky 31, 33, 35, 36, 39. Tu rozdělíme na dvě stránky tak, že do jedné dáme hodnoty 31 a 33, do druhé 36 a 39 a číslo 35 přesuneme do předchůdce, o úroveň výše.

Výsledek vidíte na obr. 2.11.

**Odstranění prvku z b-stromu**

Také odstraňování prvků z b-stromu je v podstatě jednoduché - i když detailní provedení vypadá složitě. Musíme rozlišit dvě situace, podle toho, zda nežádoucí prvek leží nebo neleží v listové stránce.

*Jestliže odstraňovaný prvek neleží v listové stránce*, vyhledáme (podobně jako u stromů) nejbližší menší prvek, tedy nejpravější prvek v levém podstromu připojeném k tomuto prvku. Přesněji: odstraňujeme-li $m$-tý prvek ve stránce $S$, vyhledáme největší prvek v $(m-1)$-tém podstromu, připojeném k $S$.

Tento „náhradní“ prvek bude určitě ležet v listové stránce. Jeho hodnotu přesuneme na místo mazaného prvku a smažeme „náhradní“ prvek v listu.

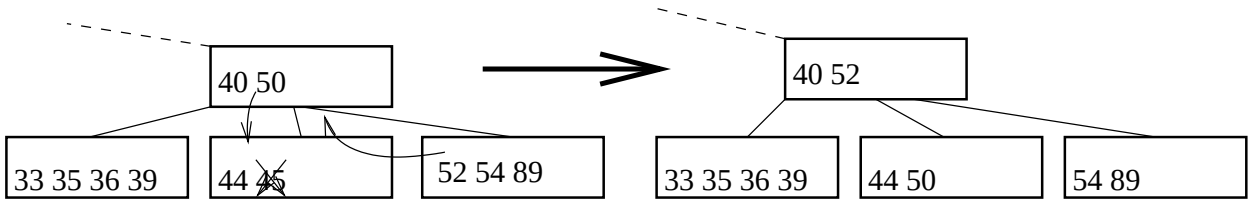
*Odstranění prvku z listu*: Poté, co prvek z listu $S$ vyjmeme, se musíme přesvědčit, kolik prvků v něm zůstalo, nebot' kromě kořene žádná stránka b-stromu nesmí mít méně než $n$ prvků. Pokud v listu $S$ zůstalo alespoň $n$ prvků, skončíme.

Pokud bude list $S$ obsahovat $n-1$ prvků, pokusíme se „vypůjčit si“ potřebný prvek ze sousední stránky (se společným předchůdcem). To je možné, jestliže sousední stránka obsahuje alespoň $n+1$ prvků.

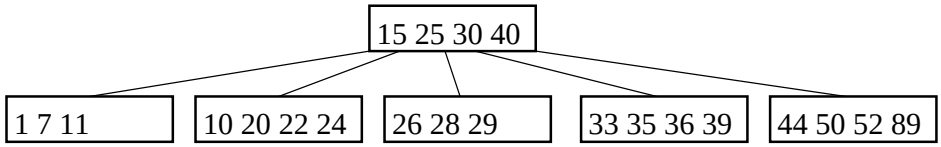
Předpokládejme, že si půjčujeme ze stránky $T$, která leží vpravo od $S$. V takovém případě do stránky $S$ převedeme prvek z předchůdce, který leží mezi $S$ a $T$, a nahradíme jej nejlevějším prvkem z $T$. Budeme-li si vypůjčovat ze stránky, která leží vlevo od $S$, bude postup podobný.

Jestliže však sousední stránka obsahuje pouze $n$ prvků, nelze si od ní „půjčit“ a musíme tyto dvě stránky sloučit. To můžeme, nebot' obě tyto listové stránky mají dohromady pouze $2n-1$ prvků.

Předpokládejme, že slučujeme list $S$ s listem $T$, který leží vpravo od $S$ a který má s $S$ společného předchůdce. Potom do stránky $S$ převedeme prvek z předchůdce, který leží mezi $S$ a $T$, spolu s ním všechny prvky z $T$ a zrušíme stránku $T$. Tím vznikl jeden list s $2n$ prvky a v jeho předchůdci ubyl jeden prvek (ten, který ležel mezi $S$ a $T$).




Obr. 2.12: Mazání s vypůjčením v části b-stromu z obr. 2.10




Obr. 2.13: Strom z obr. 2.10 po vymazání prvků 45 a 54

Přitom se může stát, že v předchůdci zbude pouze $n-1$ prvků. Opět si tedy buď vypůjčíme od souseda nebo ho s některým sousedem sloučíme (zde ale již nesmíme zapomenout na to, že musíme také přesunout připojené podstromy).

Sloučením dvou uzlů na druhé úrovni může vzniknout nový kořen. To je jediný způsob, jak se může zmenšit počet úrovní b-stromu.

**Příklad 2.6**

Vezmeme strom na obr. 2.10 a vymažeme položku 30 (kořen). Nejpravější prvek v levém podstromu je 29; tuto hodnotu tedy přesuneme do kořene a hodnotu 29 v listu smažeme. Protože v tomto listu zbyly ještě 2 prvky, můžeme skončit.

Dále smažeme hodnotu 45. Ta je v listu, který má pouze dva prvky. Proto si vypůjčíme z pravého souseda. Hodnotu 50 z předchůdce převedeme na místo smazané 45 a nahradíme ji číslem 52 ze souseda (viz obr. 2.12).
Nyní smažeme ve vzniklém stromě hodnotu 54 (poslední list vpravo). Zde si již vypůjčit nemůžeme, takže poslední stránku sloučíme s jejím levým sousedem. Tím vznikne list, který bude obsahovat hodnoty 44, 50, 52 a 89; předchůdce bude ale mít jen jediný prvek, 40. Protože si zde opět nemůžeme vypůjčit od souseda, musíme sloučit obě stránky na úrovni 2. Přitom se výška stromu sníží o jednu úroveň. Výsledek vidíte na obr. 2.13.

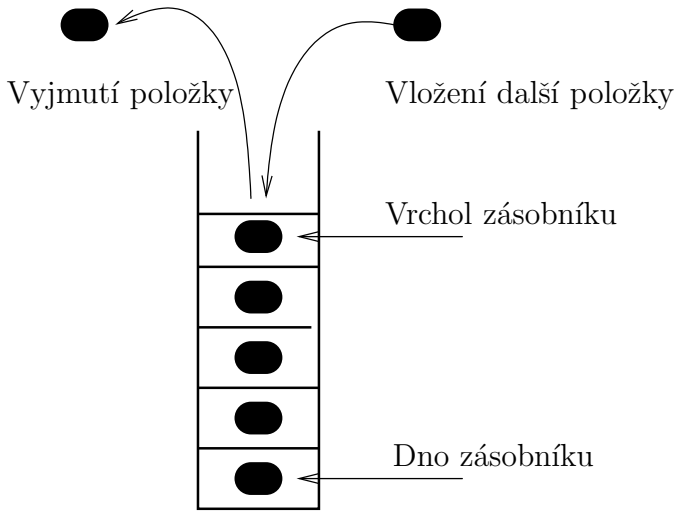
### 2.3.2 Zásobník

Zásobník (anglicky *stack*[9]) je datová struktura, do které „odkládáme“ data v průběhu zpracování. Data ze zásobníku můžeme vybírat pouze v pořadí obráceném, než v jakém jsme je do něj vložili. To znamená, že položku, vloženou jako poslední, musíme vyjmout jako první, položku, vloženou jako předposlední, vyjmeme jako druhou apod.

O poslední vložené položce říkáme, že *leží na vrcholu zásobníku*.
Položku, která neleží na vrcholu zásobníku, nelze vyjmout (museli bychom nejdříve vyjmout všechny položky, které leží nad ní). Pokud ale známe její polohu v zásobníku, můžeme ji přečíst, získat její hodnotu, aniž ji ze zásobníku vyjmeme.

Zásobník lze implementovat pomocí pole nebo pomocí seznamu. Implementujeme-li zásobník pomocí pole, musíme si neustále pamatovat index posledně vloženého prvku, který je na vrcholu zásobníku. Použijeme-li seznamu, je nejjednodušší definovat hlavu seznamu jako vrchol zásobníku.

[9] V literatuře se lze také setkat s názvem *push down storage* nebo *LIFO*, což je zkratka z označení *Last In First Out*, (poslední dovnitř, první ven - tj. objekt, který se do zásobníku vloží jako poslední, se z něj vyjme jako první).




Obr. 2.14: Schéma zásobníku

**Příklad 2.7**

Na osobních počítačích řady PC je zásobník standardní součástí uspořádání paměti libovolného programu pod DOSem. Tento zásobník je definován jako pole bytů v RAM o délce maximálně 64 KB (neboť DOS používá reálného režimu procesoru); pamět' vyhrazená pro zásobník se označuje jako „zásobníkový segment".

Adresa počátku zásobníku (její segmentová část) je uložena v registru SS; relativní adresa (tj. ofsetová část adresy) vrcholu zásobníku vzhledem k začátku zásobníku je uložena v registru SP. To znamená, že úplná adresa vrcholu zásobníku je SS:SP.

Pro usnadnění orientace na zásobníku se používá ještě registr BP[10], který vždy obsahuje relativní adresu jistého význačného bodu na zásobníku - viz dále.

Zásobník v PC má „dno" na nejvyšší adrese zásobníkového segmentu a „rozrůstá se směrem dolů", tj. každá další položka je vždy na nižší adrese než byla položka předchozí.

Pro manipulace se zásobníkem máme k dispozici řadu instrukcí asembleru. Např. instrukce PUSH XX vloží do zásobníku obsah registru XX, tj. okopíruje hodnotu z něj na vrchol zásobníku a od obsahu registru SP odečte 2 (připomeňme si, že zde zásobník „roste dolů"). Instrukce POP XX vyjme obsah dvou bytů na vrcholu zásobníku a vloží je do registru XX. Také instrukce CALL (volání podprogramu) a RET (návrat z podprogramu) využívají zásobník.

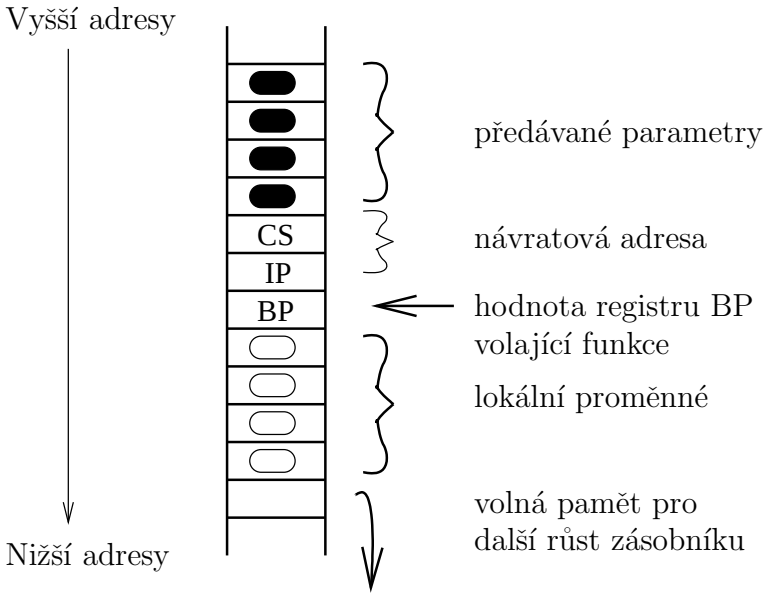
Programy využívají zásobníku při např. předávání skutečných parametrů procedurám a funkcím nebo při vytváření lokálních proměnných. Jako příklad si ukážeme, co se děje při volání funkce[11] v Borland C/C++. Příkaz

```
f(10.1, 'a');
```

- způsobí, že
  - se vypočtou hodnoty skutečných parametrů (zde není co počítat) a uloží se na zásobník; nejprve se vloží 8B, představujících hodnotu 10.1, a pak dva byty, obsahující znakovou konstantu 'a';
  - provede se instrukce CALL $f$; ta uloží na zásobník návratovou adresu a skočí na první instrukci funkce $f$;
- zavolaná funkce nejprve uloží na zásobník obsah registru BP;
- pak přesune do BP obsah registru SP;

[10] Označení registrů jsou zkratky názvů *Stack Segment*, *Stack Pointer* a *Base Pointer*.

[11] Přesně půjde o volání vzdálené funkce, která používá céčkovskou volací konvenci. Podrobné informace o tom, jak vypadá v Borland C++ volání různých typů funkcí při různých volacích konvencích, najdete v [17], kap. 6.




Obr. 2.15: Standardní ošetření při vstupu do vzdálené funkce (v Céčku na PC)

- nakonec se od SP odečte tolik, kolik bytů je třeba na lokální proměnné (tím se pro ně vyhradí místo a tak se vytvoří).

Všimněme si triku s registrem BP. Ten nyní obsahuje adresu, na které je uložen obsah BP ve volající funkci. Formální parametry mají vzhledem k tomuto místu kladnou relativní adresu, lokální proměnné zápornou. Úplná adresa lokální proměnné může být vyjádřena např. výrazem SS:[BP-10]. Registr BP představuje jakýsi počátek lokálních souřadnic na zásobníku. Viz také obrázek 2.15.

Při ukončení volané funkce se provedou tyto akce:

- do registru SP se přesune obsah registru BP (tím se ze zásobníku odstraní všechny lokální proměnné, neboť vrchol zásobníku se přesune do místa, kde je uložen obsah registru BP volající funkce);
- položka na vrcholu zásobníku se vyjme a uloží do registru BP; tím se obnoví obsah registru BP z volající funkce;
- provede se instrukce RET, která vyjme ze zásobníku návratovou adresu a skočí na ni; tím se řízení vrátilo do volající funkce;
  - volající funkce odstraní ze zásobníku parametry; tím se zásobník uvede do stejného stavu jako byl před voláním funkce $f$.

  Poznámka: Operace, uvedené symbolem „•“, se obvykle označují jako *standardní ošetření zásobníku* nebo *standardní rámec zásobníku* (*standard stack frame*).

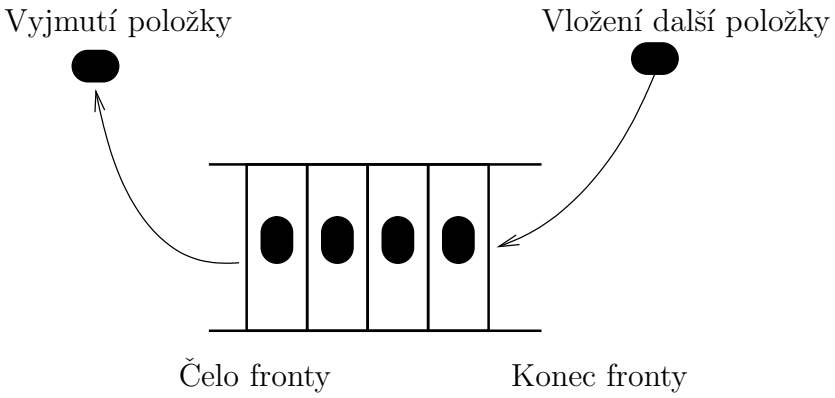
### 2.3.3 Fronta

Fronta[12] (obr. 2.16) je datová struktura podobná zásobníku. Její vnitřní organizace je ale odlišná. Prvky do fronty vkládáme na jedné straně (konci) a vybíráme je na straně druhé (čele). Ve frontě jsou uloženy v pořadí, ve kterém byly do fronty zařazeny, takže je z fronty vybíráme ve stejném pořadí, v jakém jsme je do ní vložili.

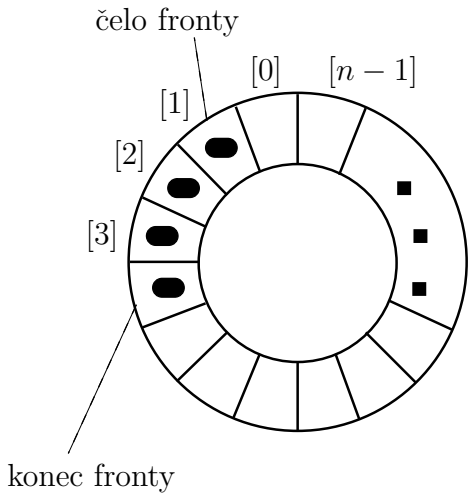
Podobně jako u zásobníku lze prvek z fronty vyjmout pouze v případě, že „ je na řadě“; jeho hodnotu můžeme ale přečíst kdykoli (prvek tam zůstane).

Frontu lze, podobně jako zásobník, reprezentovat buď pomocí pole nebo pomocí seznamu.

[12] Vedle názvu *fronta* (anglicky *queue*) se často setkáme s termínem FIFO, což je zkratka anglického označení *First In, First Out* (první dovnitř, první ven).




Obr. 2.16: Schéma fronty




Obr. 2.17: Kruhová fronta

**Fronta s prioritami**

Priorita prvku je nějaká funkce hodnoty v prvku uložené. Fronta s prioritami (s předbíháním) se liší od „obyčejné“ fronty tím, že prvky sice ukládáme v pořadí, ve kterém přišly, vybíráme je ale v pořadí závislém na jejich prioritě (nejprve prvek s nejvyšší prioritou).

Priority lze uplatňovat i při vkládání prvků do fronty. Fronty s prioritami se obvykle implementují pomocí seznamů.

**Kruhová fronta**

Kruhová fronta představuje jednu z obvyklých implementací fronty. Vezmeme pole $Q[0, \ldots, n-1]$ a budeme s ním zacházet, jako kdyby bylo kruhové, tedy kdyby po prvku $Q[n-1]$ následoval prvek $Q[0]$.

Pro obsluhu kruhové fronty potřebujeme ukazatel $f$ na čelo a ukazatel $r$ na konec. Při každé operaci vkládání nebo výběru z kruhové fronty kontrolujeme, zda náhodou nenastala rovnost $f = r$. Nastane li tato situace po výběru z fronty, znamená to, že fronta je prázdná. Nastane-li tato rovnost po vložení nového prvku, znamená to, že je fronta již plná.

### 2.3.4 Tabulka

*Tabulka symbolů* je datová struktura, která umožňuje rychle zjistit, zda se v ní někde nějaký prvek vyskytuje. Umožňuje také snadno a rychle vložit nový prvek nebo nějaký prvek vyjmout. Takovéto tabulky se často používají v překladačích při lexikální analýze: jakmile narazíme v překládaném programu na identifikátor, je třeba zjistit, zda byl již deklarován, a pokud ne, zařadit jej spolu s dalšími informacemi do tabulky.

Roli tabulky symbolů mohou docela dobře hrát binární stromy. Jestliže však nemáme žádné apriorní informace o statistickém rozdělení vkládaných prvků, nebudou vytvořené stromy zpravidla příliš optimální. Proto se obvykle používají *hešové tabulky*[13]. Ukládání hesel do těchto tabulek, *hešování*, je v principu velice jednoduché a účinné. Na druhé straně modstraňování hesel z hešové tabulky může být obtížné - dále uvidíme, proč.

**Hešová tabulka**

Hešová tabulka je vyhrazená souvislá oblast paměti, která slouží pro ukládání prvků - hesel. Tabulka je rozdělena na tzv. *koše*, kterých je řekněme $k$ a které označíme $K(1)$, $K(2)$,..., $K(k)$. V každém z košů může být s hesel (každý z košů má $s$ pozic). Často se volí $s = 1$.

Hešovou tabulku lze implementovat např. jako pole košů nebo pole hesel.

Při vyhledávání v binárním stromu jsme postupně porovnávali hledaný prvek s prvky v jednotlivých uzlech. Naproti tomu polohu (adresu, index koše) prvku $X$ v hešové tabulce vypočteme pomocí *hešovací funkce* $f(X)$. Hodnotu $f(X)$ označujeme jako *hešovou adresu* prvku $X$.

Hešovací funkce zobrazuje množinu všech možných hesel na množinu

$$\hat{n} = \{1, 2, \ldots, k\}.$$

Označíme $T$ mohutnost *prostoru hesel*, tj. počet všech možných hesel. Poměr $n/T$, kde $n$ je počet hesel, uložených v tabulce, označíme jako *hustotu hesel*, a poměr $n/(sk)$ jako *zaváděcí faktor*.

Podívejme se např. na tabulku, do které budeme ukládat identifikátory v programu napsaném ve Fortranu. Fortranské identifikátory obsahují nejvýše 6 znaků, první musí být písmeno anglické abecedy, další mohou být písmena nebo číslice. Na velikosti písmen nezáleží.

To znamená, že velikost prostoru hesel , tj. počet možných identifikátorů, je

$$T = 26 \sum_{i=0}^{5} 36^i \approx 1,6.10^9.$$

Počet identifikátorů v programu bývá ovšem o několik řádů menší.

Předchozí příklad ukazuje, počet hesel, vkládaných do tabulky, bude zpravidla podstatně menší než $T$. Proto se také počet košů, $k$, volí podstatně menší než $T$. Z toho ale plyne, že může nastat kolize: hešovací funkce může zobrazit dvě různá hesla do stejného koše, tj. pro dvě různá hesla $H_1 \neq H_2$ může platit $f(H_1) = f(H_2)$. (Taková hesla označujeme jako *synonyma* vzhledem k dané hešovací funkci *f*. )

Synonyma se ukládají po řadě do následujících pozic ve stejném koši. Jestliže hešovací funkce zobrazí nové heslo do koše, který už neobsahuje žádnou volnou pozici, nastane *přeplnění*.

**Hešovací funkce**

Hešovací funkce zobrazuje heslo (zpravidla identifikátor) na adresu koše. Je jasné, že potřebujeme funkci, kterou půjde snadno a rychle vypočítat a která přitom bude minimalizovat počet kolizí (a tím i počet přeplnění). Pravděpodobnost, že náhodně zvolené heslo $X$ z prostoru hesel padne do kteréhokoli koše, by měla být pro všechny koše stejná, rovná $1/k$. Hešovací funkce by tedy měla vést k rovnoměrnému rozdělení hesel v tabulce.

Velmi jednoduchou, nikoli ovšem dobrou možnost představuje hešovací funkce založená na abecedním třídění.

Uvažujme např. tabulku identifikátorů se šestadvaceti koši, $k = 26$. Hešovací funkce bude identifikátorům, začínajícím A, přiřazovat první koš, identifikátorům, začínajícím B druhý koš atd. V programech se ovšem často vyskytují identifikátory, které začínají stejně - např. $A1$, $A2$, $A3$ apod. Na druhé straně některá počáteční písmena může programátor z různých důvodů diskriminovat.

[13] Anglicky *hash tables*, což znamená asi tak "rozsekané tabulky". V češtině můžete setkat s označením "tabulka z rozptýlenými hesly" nebo "rozptýlená tabulka".

To znamená, že prosté abecední řazení nebude příliš výhodné.

Jednoduchou a efektivní možnost představuje použití operace modulo. Např. interpretujeme heslo jako celé číslo[14], které vydělíme vhodným číslem $M$. Zbytek po dělení pak použijeme jako hodnotu hešovací funkce,

$$f_D(X) = X \bmod M$$

Velikost hešové tabulky pak musí být v tomto případě alespoň $M$.

Rozumnost této funkce závisí na hodnotě $M$. Pokud bychom například zvolili $M$ rovno nějaké mocnině dvou, bude hodnota $f$ určena pouze posledními několika bity hesla.

Kdybychom např. zvolili $M$=256, představovala by hodnota $f_D(X)$ poslední byte hesla. Taková funkce ale rozhodně nebude rovnoměrná; snadno nahlédneme, že je daleko pravděpodobnější např. identifikátor, končící číslicí **1**, než identifikátor, končící číslicí **9**.

Lze ukázat, že nejvýhodnější je použít jako $M$ prvočíslo. Pokud lze totiž $M$ zapsat jako součin několika čísel, budou se hesla, která vzniknou jedno z druhého permutací znaků, chovat často jako synonyma. D. Knuth dále ukázal [4], že vhodná jsou prvočíselná $M$, která nejsou děliteli čísel $r^t + a$ nebo pro $r^t - a$ malá $t$ a $a$ ($r$ je základ použité číselné soustavy).

Praxe ukazuje, že stačí volit $M$, které nemá prvočíselné dělitele menší než 20 [2].

Jiná často používaná hešovací funkce se označuje jako „střed druhé mocniny“. Předpokládáme, že heslo lze uložit v jednom počítačovém slovu. Umocníme jej na druhou a vezmeme odpovídající počet bitů z prostředka výsledku.

### Přeplnění

Přeplnění nastane, jestliže se pokusíme uložit nové heslo do plného koše. Podívejme se na některá možná řešení této situace.

Asi nejjednodušší řešení může být vyhledání prvního nezaplněného koše. Je-li koš $f(x)$ zaplněn, zkusíme následující; pokud je plný i ten, vezmeme další apod. Podobně postupujeme při vyhledávání hesla. Jestliže heslo nenajdeme v koši $f(x)$ a tento koš je plný, hledáme v koši následujícím atd.

Zpravidla se ovšem neuvažuje hned následující koš, ale koš $f(x)+m$ pro nějaké $m > 1$. Tím obvykle dosáhneme rovnoměrnějšího rozložení obsazených košů. Čísla $m$ a $k$ ale nesmí být soudělná, jinak bychom nemohli v případě potřeby projít celou tabulku.

Tato metoda se nazývá *lineární otevřené adresování*. Není příliš účinná, neboť v nejhorším případě při ní vyzkoušíme $k$ - 1 košů. To může být podstatně horší než při prohledávání binárních stromů.

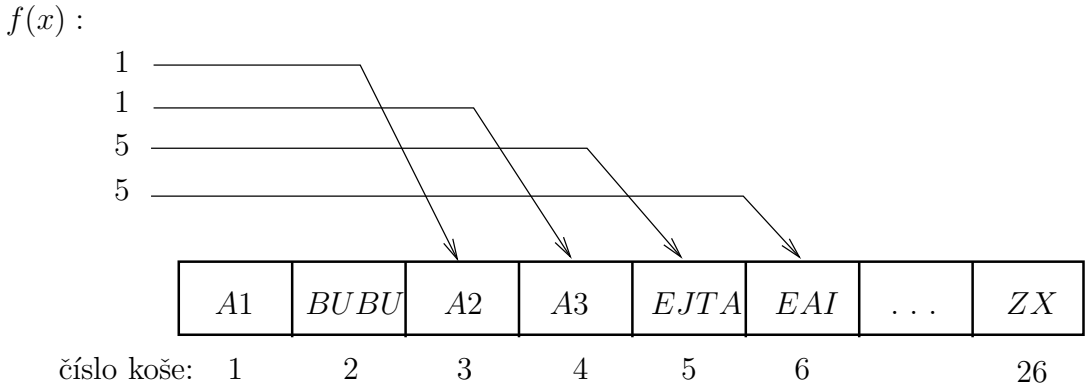
### Příklad 2.6

Uvažujme hešovou tabulku s 26 koši a s jednou pozicí v každém koši. Chceme do ní vložit identifikátory *A1*, *BUBU*, *A2*, *A3*, *EJTA*, *EAI*, *ZX*, *EEE*, *B1* (v tomto pořadí). Použijeme otevřené lineární adresování s $m = 1$ a hešovací funkci, jejíž hodnota bude rovna prvnímu písmenu hesla.

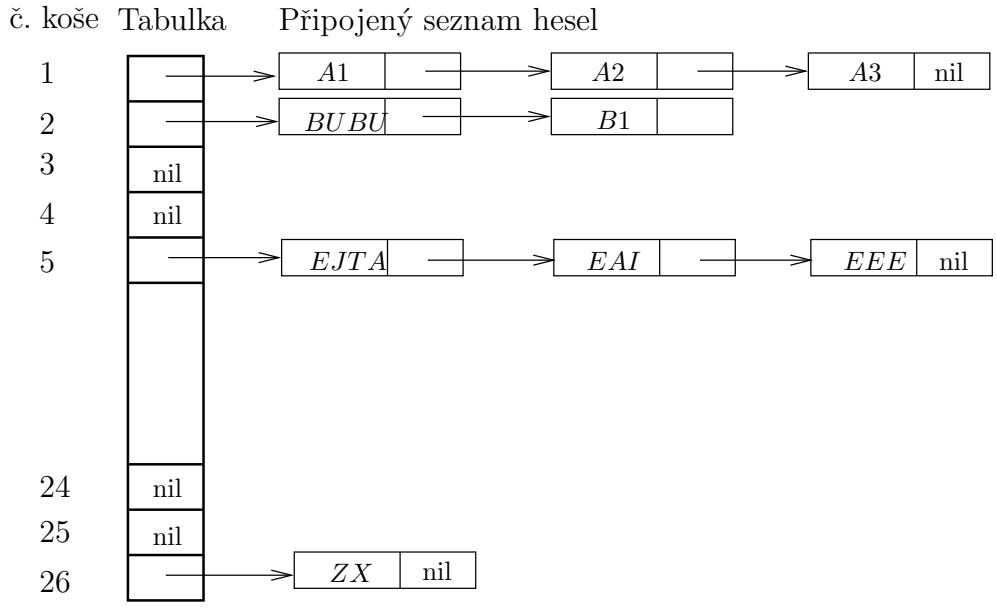
První dvě hesla přijdou do prázdných košů. Třetí heslo, *A*2, patří do prvního koše, nejbližší volný je ale až třetí. Podobně *A*3 přijde do čtvrtého koše. Heslo *EJTA* patří do pátého koše, který je volný... atd. Viz obr. 2.18.
Problémy s přeplněním lze odstranit také tím, že hesla nebudeme ukládat do přímo košů, ale do seznamů, připojených ke košům. Koš bude obsahovat pouze ukazatel na hlavu seznamu hesel, která jsou v něm uložena.

Schéma takovéto tabulky vidíte na obr. 2.19.

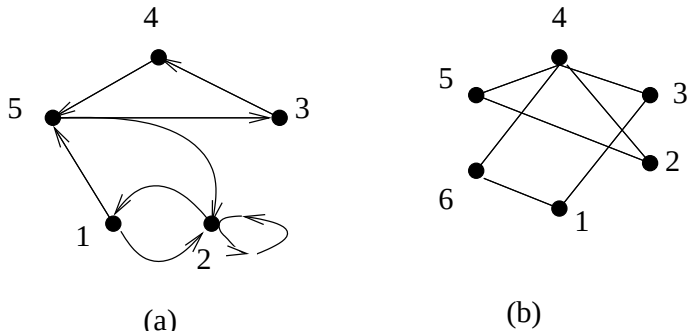
[14] Takováto funkce bude samozřejmě silně závislá na způsobu ukládání znakových řetězců v daném systému.




Obr. 2.18: Vkládání hesel do hešové tabulky




Obr. 2.19: Hešová tabulka se seznamy připojenými ke košům




Obr. 2.20: (a) orientovaný graf, (b) neorientovaný graf

### 2.3.5 Grafy

*Orientovaný graf* $G$ definujeme jako dvojici $G = \{U, H\}$ ,kde $U$ je konečná množina uzlů, kterou zpravidla ztotožňujeme s množinou $\hat{n} = \{1, 2, \ldots, n\}$ a $H \subset U \times U$ je množina orientovaných hran. O orientované hraně $e = \langle i, j \rangle$ kde $i$ a $j$ jsou uzly, řekneme, že jde z uzlu $i$ do uzlu $j$. Orientovaný graf může obsahovat i hranu $\langle i, j \rangle$. (Místo $\langle i, j \rangle$ se pro orientované hrany také používá zápis $i \rightarrow j$.)

Jestliže ztotožníme hranu $\langle i, j \rangle$ s hranou $\langle j, i \rangle$, dostaneme *neorientovaný* graf.

Grafy se obvykle znázorňují obrázky podobnými jako je 2.20.
Někdy je vhodné definovat zobrazení $h : H \rightarrow R$ které jednotlivým hranám přiřazuje číselné hodnoty. Pak hovoříme o *ohodnoceném grafu*.

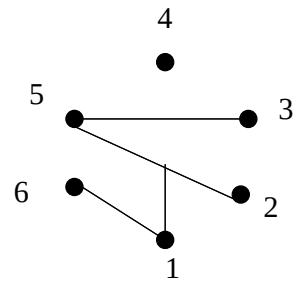
Posloupnost orientovaných hran $\langle i, i_1 \rangle, \langle i_1, i_2 \rangle, \ldots, \langle i_k, j \rangle$označujeme jako cestu, vycházející z uzlu $i$ a končící v uzlu $j$. Na obrázku 2.20 (a) vidíme např. cestu $\langle 1, 5 \rangle$, $\langle 5, 3 \rangle$, $\langle 3, 4 \rangle$ z uzlu 1 do uzlu 4. Snadno zjistíme, že pokud graf $G$ obsahuje cestu z uzlu $i$ do $j$ a z $j$ do $k$, obsahuje také cestu z $i$ do $k$.

Cestu, jejíž počáteční uzel je roven uzlu koncovému, označujeme jako *cyklus*.

Podobně jestliže v neorientovaném grafu existuje posloupnost neorientovaných hran $\langle i, i_1 \rangle, \langle i_1, i_2 \rangle, \ldots, \langle i_k, j \rangle$, označíme ji jako cestu, spojující uzly $i$ a $j$. Uzly, mezi kterými existuje cesta, označíme jako *sdružené*. Není těžké ukázat, že množina všech uzlů, sdružených s uzlem $i$, tvoří třídu ekvivalence. Množina těchto uzlů spolu s hranami, které je spojují, se nazývá *komponenta grafu*.

Z této definice plyne, že pokud uzly $i$ a $j$ leží v téže komponentě, existuje mezi nimi cesta. Jestliže naopak leží ve dvou různých komponentách grafu, cesta mezi nimi neexistuje. Komponenty tedy představují vlastně samostatné grafy.

Neorientovaný graf, který se skládá z jediné komponenty, označujeme jako *souvislý*. Obrázek 2.21 ukazuje neorientovaný graf, složený ze dvou komponent.
Orientované grafy se často reprezentují pomocí *incidenčních matic*. Incidenční matice grafu $G$, který má $n$ uzlů, bude typu $n \times n$ a její prvek $a_{ij}$ bude roven 1, jestliže graf obsahuje hranu $\langle i, j \rangle$, a 0 v případě, že ji neobsahuje.




Obr. 2.21: Graf složený ze dvou komponent: 1-4-6 a 2-3-5.

Chceme-li reprezentovat ohodnocený graf, doplníme do něj hrany, které se v něm nevyskytují, a přidělíme jim např. ohodnocení $+\infty$ (nebo jakoukoli jinou hodnotu, která se nemůže vyskytnout jako ohodnocení existující hrany). Graf pak reprezentujeme maticí, jejíž prvky obsahují ohodnocení jednotlivých hran.

Bude-li se u grafu měnit průběžně počet hran či jejich ohodnocení, budeme prostě měnit prvky incidenční matice. Jestliže se ale může měnit i počet uzlů, je výhodnější reprezentovat graf pomocí seznamu uzlů. K záznamu o každém z uzlů připojíme seznam hran, které z něj vycházejí (nebo hran, které do něj vcházejí).

### 2.3.6 Množiny

Množiny s prvky ordinálních typů

Nejčastěji se setkáváme s množinami prvků některého z ordinálních typů (znaky, intervaly, výčtové typy). Protože ordinální typy mají pouze konečný počet hodnot, můžeme odpovídající množinu reprezentovat pomocí bitových příznaků.

Jestliže má bázový typ, tj. typ prvků množiny $M$, celkem $n$ různých hodnot, můžeme tyto hodnoty očíslovat, přiřadit jim čísla $0, 1, \ldots, n-1$. Nultý bit proměnné $M$ bude roven 1, právě když množina $M$ obsahuje prvek s pořadovým číslem 0, první bit $M$ bude indikovat přítomnost prvku s pořadovým číslem 1 atd.

Snadno se přesvědčíme, že obecně přítomnost $k$-té hodnoty bázového typu (při číslování od nuly) indikuje bit číslo $k - [k/8]\,8 = k \bmod 8$ v bytu číslo $[k/8] = k \operatorname{div} 8$.

Prázdná množina bude mít všechny bity nulové.

Podívejme se na proměnnou *paleta*, definovanou jako množinu *barev*:

```
type barvy = (červená, žlutá, zelená, modrá, bílá, oranžová
              fialová, černá);
var paleta: set of barvy;
```

Protože bázový typ *barvy* má pouze osm prvků, stačí k reprezentaci jakékoli množiny barev - tedy i proměnné *paleta* - jeden byte. Obsahuje-li *paleta* zelenou barvu, bude 3. bit roven 1, jinak bude nulový.

Základní operace s množinami implementujeme jako bitové operace. Přidání prvku do množiny $M$ znamená nastavení příslušného bitu na hodnotu 1, vyjmutí prvku z množiny $M$ odpovídá nastavení příslušného bitu na 0. Test přítomnosti prvku (v Pascalu operátor **in**) v množině $M$ je testem hodnoty bitu.

Doplněk $C(M)$ množiny $M$ do množiny všech prvků bázového typu vytvoříme pomocí bitové negace. Průnik dvou množin $M$ a $N$ se stejným bázovým typem získáme pomocí bitové konjunkce (prvek je v průniku $M \cap N$, je-li zároveň v $M$ i v $N$), sjednocení pomocí operace bitové disjunkce (prvek je ve sjednocení $M \cup N$„ je-li alespoň v jedné z množin $M$ nebo $N$).

Jen nepatrně složitější je rozdíl množin: množina $M - N$ obsahuje prvky, které jsou v $M$ a nejsou v $N$ (jsou v doplňku $N$), takže $M - N = M \cap C\,(N)$. Výsledek tedy bude konjunkce $M$ a bitové negace $N$.

#### Množiny s obecnými prvky

Jinou možnost, jak implementovat množiny, představují seznamy objektů.

Předpokládejme, že všechny možné prvky naší množiny $M$ můžeme reprezentovat pomocí instancí objektových typů se společným předkem $T$. Pak můžeme $M$ implementovat jako seznam, jehož složkami budou ukazatele na typ $T$ a budou obsahovat adresy prvků.

Operace s takto implementovanými množinami jsou ovšem podstatně náročnější na čas, neboť znamenají opakované prohledávání seznamů.

# Kapitola 3

# Metody návrhu algoritmů

V této kapitole budeme hovořit o nejužívanějších metodách návrhu algoritmů, tedy vlastně o způsobech řešení problémů se zřetelem k tomu, že výsledkem má být program. Přehled, který zde uvedeme, samozřejmě nemůže být vyčerpávající. Musíme ovšem zdůraznit, že žádná z těchto metod nemusí vést k cíli.

Výklad v této kapitole budeme ilustrovat pouze velmi jednoduchými příklady. S dalšími aplikacemi popsaných metod se setkáme v následujících kapitolách.

## 3.1 Rozděl a panuj

*Rozděl a panuj* (anglicky *divide and conquer*) je nejběžnější metoda, se kterou se můžeme setkat. Můžeme se na ni dívat jako na aplikaci postupu *shora dolů*, se kterým jsme se seznámili v kap. 1.1.3.

Potřebujeme zpracovat množinu $V$ složenou z $n$ údajů. Toto množinu rozdělíme na k disjunktních podmnožin, které zpracujeme každou zvlášť. Získané dílčí výsledky pak spojíme, odvodíme z nich řešení pro celou množinu $V$.

Přitom se může stát, že problém zpracování dílčích podmnožin je stejného typu jako původní problém zpracování všech $n$ údajů. V takovém případě vede opakování metody *rozděl a panuj* zcela přirozeně k rekurzi.

Podívejme se pro určitost na situaci, kdy množinu vstupních dat, reprezentovanou polem $A$, rozdělíme na dvě podmnožiny a kdy zpracování takto získaných podmnožin představuje úlohu stejného typu jako byla úloha původní. V takovém případě bychom mohli metodu *rozděl a panuj* symbolicky zapsat ve tvaru následující funkce *RAP*:

```
var A: array [1..n] of data;                          {vstupní data}
function RAP(p,q: integer): výsledek;        {řešení pro i=p,..,q}
var m: integer;                                 {1 <= p <= q <= n}
begin
   if MALÉ(p,q) then RAP := G(p,q)
   else begin
      m := ROZDĚL(p,q);                                    {p < m < q}
      RAP := „SLOŽ RAP(p, m) a RAP(m + 1, q)“
   end;
end;
```

Zde předpokládáme, že *MALÉ* je booleovská funkce, která vrátí **true**, je-li interval $p..q$ dostatečně malý, aby jej nebylo třeba dále dělit. V takovém případě zpracuje prvky $A[p], \ldots, A[q]$ funkce $G$. Jinak určíme pomocí procedury *ROZDĚL* číslo $m$, které interval $p..q$ rozdělí na dva podintervaly a výsledek sestavíme z dílčích výsledků pomocí operace *SLOŽ*.

**Příklad 3.1 (binární vyhledávání)**

Je dáno setříděné pole $A$ celých čísel. Hodnoty jsou v tomto poli uloženy v neklesajícím pořadí. Naším úkolem je určit, zda toto pole obsahuje zadanou hodnotu x, a pokud ano, vrátit index prvku, ve kterém je uložena. Pokud $x$ v poli $A$ není, vrátíme 0.

Postup řešení metodou *rozděl a panuj*: Pole $A$ rozdělíme na několik úseků (např. na dva, přibližně stejně dlouhé) a budeme testovat každý zvlášť. Protože je toto pole seřazené podle velikosti prvků, snadno zjistíme, zda v něm může dané $x$ vůbec ležet.

Každý z úseků ovšem představuje opět pole typu *integer*, takže jej znova rozdělíme. Dělení skončí, když dojdeme k úsekům délky 1. Zde již porovnáním snadno zjistíme, zda jsme zadané $x$ našli.

Algoritmus, kterým zjistíme, zda úsek $A[p], \ldots, A[p]$ obsahuje hodnotu $x$, může mít následující tvar:

1. Je-li $p = q$, zjistíme, zda platí $A[p] = x$. Pokud ano, je výsledkem $p$, jinak je výsledkem 0.
2. Jinak položíme $m := (p + q) \operatorname{div} 2$.
3. Je-li $x \in \langle A[p], A[m] \rangle$, opakujeme tento postup pro $q := m$, jinak
4. je-li $x \in \langle A[m+1], A[q] \rangle$, opakujeme tento postup pro $p := m + 1$, jinak
5. hodnota $x$ v poli $A$ není, výsledek je 0.

Podívejme se, jak je to se složitostí tohoto algoritmu. Nejprve budeme předpokládat, že délka pole A je rovna $n = 2^m$ pro nějaké přirozené $m$. V prvním kroku pole rozdělíme na dva úseky délky $2^{m-1}$ a v jednom z nich budeme hledat. V následujícím kroku dostaneme úsek délky $2^{m-2}$, pak $2^{m-3}$ atd. To znamená, že po $m$ krocích dospějeme k úseku délky 1. Odtud plyne, že pro $n = 2^m$ je počet operací $K$ úměrný $m = \log_2 n$,

Dále je zřejmé, že při $n < 2^m$ nemůže být počet operací větší než při $n = 2^m$. To znamená, že počet operací při binárním vyhledávání je vždy $O(\log_2 n)$.

Poznamenejme, že prohledání pole prvek po prvku vyžaduje $O(n)$ operací.

## 3.2 Hladový algoritmus

Hladový algoritmus představuje velmi přímočarý přístup k řešení určité třídy optimalizačních úloh. Je až s podivem, jak často jej lze s úspěchem použít.

Je dána množina $V$ složená z n vstupních hodnot. Naším úkolem je najít podmnožinu $W$ množiny $V$, která vyhovuje určitým podmínkám a přitom optimalizuje (tj. minimalizuje nebo maximalizuje) předepsanou *účelovou funkci*.

Jakoukoli podmnožinu $W$, vyhovující daným podmínkám, označíme jako *přípustné řešení*. Přípustné řešení, pro které nabývá účelová funkce optimální hodnoty, označujeme jako *optimální řešení*.

Hladový algoritmus se bude skládat z kroků, které budou probírat jednotlivé prvky vstupní množiny $V$, a v každém kroku rozhodne, zda se daný prvek hodí do optimálního řešení. Prvky $V$ bude probírat v pořadí, určeném jistou výběrovou procedurou.

V každém kroku musíme ovšem dostat přípustné řešení. Prvek, který by vedl k nepřípustnému řešení, nevezmeme v úvahu.

Výběrová procedura, která určuje pořadí, v jakém budeme zpracovávat prvky $V$, bude založena na nějaké optimalizační míře - funkci, která může být odvozena od účelové funkce.

Formálně bychom mohli hladový algoritmus vyjádřit následující procedurou *HLAD*. Čtenář nám jistě promine, když tentokrát porušíme syntaktická pravidla jazyka Pascal více než obvykle - jde nám o symbolický zápis, nikoli o program.

```
var řešení: set of data;                          {hledaná množina}
function HLAD(A: set of data; n: integer): set of data;
var i: integer;                       {n je počet prvků množiny A}
    x: data;
```

```
begin                                          {prázdná množina je}
   řešení := [ ];                               {přípustné řešení}
   for i := 1 to n do begin
      x := ZVOL(A);                            {určíme další prvek}
      if PŘÍPUSTNÉ(řešení, x) then       {lze x přidat k řešení?}
         řešení := řešení + [x];        {když lze, tak ho přidáme}
   end;
   HLAD := řešení;
end;
```

Funkce *ZVOL* vybere další hodnotu a odstraní ji z množiny $A$. Booleovská funkce *PŘÍPUSTNÉ* testuje, zda přidáním $x$ vznikne přípustné řešení. Poznamenejme, že typ vstupních údajů *data* zpravidla nemůže sloužit jako bázový typ množiny v Pascalu.

**Příklad 3.2**

Na magnetické pásce počítače je třeba uložit $n$ souborů s délkami $l_1, \ldots, l_n$. Magnetická páska umožňuje pouze sekvenční přístup k souborům, takže chceme-li číst $r$-tý soubor, musíme nejprve přečíst $r-1$ souborů, zapsaných před ním. Doba čtení souboru je přímo úměrná délce souboru a pravděpodobnosti čtení jsou u všech souborů stejné. V jakém pořadí máme soubory uložit, aby byla střední doba přístupu k souborům nejmenší?

Ze zadání plyne, že doba čtení jednoho souboru je přímo úměrná jeho délce. Doba $t_k$, potřebná k nalezení a přečtení $k$-tého souboru na pásce, je tedy úměrná součtu délek prvních $k$ souborů. Jsou-li soubory uloženy na pásce v pořadí daném permutací $I = (i_1, \ldots, i_n)$, platí

$$t_k \cong \sum_{j=1}^{k} l_{i_1}$$

a střední doba čtení souboru bude úměrná veličině $(I)$, kterou zavedeme vztahem

$$t\,(I) \cong \frac{1}{n}\sum_{j=1}^{n} t_j = \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_1} = \tau\,(I) \tag{3.1}$$

Naším úkolem je najít takovou permutaci $I = (i_1, \ldots, i_n)$, pro kterou bude střední doba čtení souboru $t\,(I)$ minimální.

Tuto úlohu se pokusíme vyřešit pomocí hladového algoritmu. Všechny permutace jsou přípustná řešení. Účelovou funkcí bude minimální veličina $(I)$, zavedená vztahem (3.1) (a tedy také střední doba přístupu k souboru $t\,(I)$). Tato účelová funkce závisí na zvolené permutaci $I$.

Při konstrukci permutace $I$ budeme postupovat tak, aby $(I)$ bylo stále co nejmenší: jako další v pořadí vezmeme vždy soubor, pro který $(I)$ vzroste nejméně. To znamená, že soubory uložíme v pořadí podle rostoucí délky[1]. Ukážeme, že tak opravdu získáme optimální řešení.

Abychom si zjednodušili zápis, očíslujeme soubory tak, aby platilo $l_1 < \ldots < l_n$. Chceme dokázat, že optimální permutace je pak $(1, 2, \ldots, n)$. Vezmeme libovolnou permutaci $I = (i_1, \ldots, i_n)$. Z (3.1) plyne, že

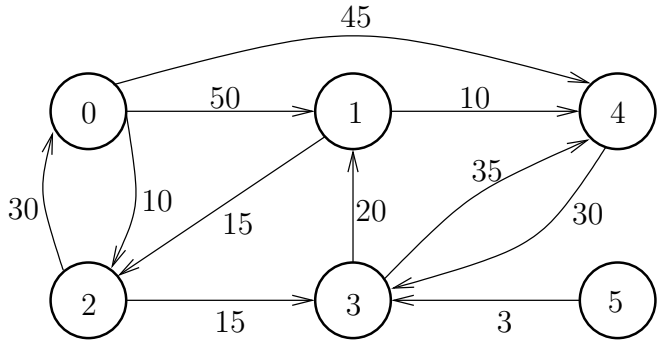
$$\tau\,(I) = \frac{1}{n}\sum_{k=1}^{n}\sum_{j=1}^{k} l_{i_1} = \frac{1}{n}\sum_{k=1}^{n}(n-k+1)\,l_{i_1},$$

neboť $l_{i1}$ je v tomto součtu $n$-krát, $l_{i2}$ je tam $(n-1)$-krát apod. Jestliže existují indexy $a < b$ takové, že $l_{ia} > l_{ib}$, dostaneme záměnou $i_a$ a $i_b$ permutaci $I'$, pro kterou platí

$$\tau\,(I') = \frac{1}{n}\sum_{k=1,\, k\neq a,\, k\neq b}^{n} (n-k+1)\,l_{i_1} + (n-b+1)\,l_{i_a} + (n-a+1)\,l_{i_b},$$

takže pro rozdíl hodnot účelové funkce pro permutace $I$ a $I'$ dostaneme

[1] Výběrová procedura se tedy řídí požadavkem, aby se přidáním dalšího souboru účelová funkce zvětšila co nejméně. To je typické použití hladového algoritmu, od kterého také pochází jeho označení: jako hladovci sáhneme po tom prvku, které poskytuje největší okamžitý prospěch. (Poznamenejme, že anglické označení *the greedy method* je ještě o něco méně učesané; *greedy* znamená *žravý*.)




Obr. 3.1: Ohodnocený graf

$$n\left[\tau\left(I\right)-\tau\left(I'\right)\right]=\left(n-a+1\right)\left(l_{i_a}-l_{i_b}\right)+\left(n-b+1\right)\left(l_{i_b}-l_{i_b}\right)=\left(b-a\right)\left(l_{i_a}-l_{i_b}\right)>0.$$

To ale znamená, že záměnou souborů $i_a$ a $i_b$ se účelová funkce zmenšila, a tedy se také zmenšila střední doba přístupu k souborům. Proto nemůže být optimální žádná permutace, při které nejsou soubory uspořádány podle velikosti v neklesajícím pořadí.

**Příklad 3.3 (problém nejkratší cesty)**

V ohodnoceném orientovaném grafu (viz např. obr. 3.1) je dán „výchozí“ uzel $v_0$. Hledáme nejkratší cesty do všech ostatních uzlů grafu, tj. takové cesty, které budou mít nejnižší součet ohodnocení hran[2].
Ze zadání je jasné, že účelovou funkcí bude součet ohodnocení hran cesty.

Náš postup bude založen na následující myšlence: Nejprve najdeme uzel $v_1$, který je nejbližší výchozímu uzlu $v_0$ (tj. vede do něj hrana, která má nejnižší ohodnocení ze všech hran, které vycházejí z $v_0$). Je zřejmé, že jsme tím našli nejkratší cestu do uzlu $v_1$. Dále projdeme všechny uzly, do kterých vede hrana z $v_0$ nebo z $v_1$, a najdeme mezi nimi ten, do kterého vede nejkratší cesta z $v_0$. Označíme jej $v_2$. Přitom nejkratší cesta z $v_0$ do $v_2$ může být buď hrana $v_0 \to v_2$ nebo cesta $v_0 \to v_1 \to v_2$. Dále projdeme všechny uzly, do kterých vede hrana z $v_0$, z $v_1$ nebo z $v_2$ atd. Takto postupně najdeme nejkratší cesty do všech uzlů grafu - nejprve cestu do uzlu, nejbližšího k výchozímu, pak do druhého nejbližšího atd.

Nyní tento postup vyjádříme formálněji a ukážeme, že je správný.

Definujeme $S$ jako množinu uzlů, do kterých již byly nalezeny nejkratší cesty z $v_0$. (Množina $S$ bude samozřejmě na počátku obsahovat jen uzel $v_0$.) Dále označíme $Dist\left(w\right)$ délku nejkratší cesty z uzlu $v_0$ do $w$, procházející pouze uzly z $S$ a končící ve $w$. Pokud taková cesta neexistuje, tj. pokud ze žádného z uzlů množiny $S$ nevede hrana do $w$, položíme $Dist\left(w\right)=+\infty$.

Přitom si povšimneme následujících skutečností:

(1) Jestliže jsme při dalším kroku našli nejkratší cestu do uzlu $u$, jde o cestu z uzlu $v_0$ do $u$ procházející pouze uzly z množiny $S$. Kdyby totiž na nejkratší cestě $v_0 \to u$ ležel uzel $w \notin S$, znamenalo by to, že se cesta $v_0 \to u$ skládá z cest $v_0 \to w$ a $w \to u$. Cesta $v_0 \to w$ je ale nutně kratší než $v_0 \to u$, takže podle našeho postupu bychom ji museli vzít před $v_0 \to u$.

(2) Z postřehu (1) plyne, že koncovým uzlem následující generované nejkratší cesty musí být uzel $u$, který má nejmenší vzdálenost $Dist\left(u\right)$ ze všech uzlů, které neleží v $S$. Pokud má více uzlů stejnou $Dist\left(u\right)$, můžeme zvolit kterýkoli z nich. Jinými slovy: cesta z $v_0$ do nově přidaného uzlu $u$ se skládá z nejkratší cesty z $v_0$ do některého z uzlů množiny $S$ a z hrany, která vede z tohoto uzlu do $u$.

(3) Jakmile zvolíme u podle (2) a vygenerujeme nejkratší cestu do tohoto uzlu, stane se $u \in S$. Přitom se pro uzly $w \notin S$ může změnit $Dist\left(w\right)$. (Všimněte si, že jakmile bude na obrázku 3.1 uzel

[2] Jestliže ohodnocení hrany $\langle i,j\rangle$ interpretujeme jako délku cesty z uzlu $i$ do $j$, pak hledáme nejkratší cestu. Ohodnocení hrany ovšem může také znamenat např. cenu přechodu z $i$ do $j$. Proto se také občas hovoří o problému *nejlacinější cesty* (a můžeme se setkat i s jinými názvy).

číslo $3 \in S$, zmenší se $Dist\,(1)$. V takovém případě bude $Dist\,(w) = Dist\,(u) + C\,(u, v)$, kde $C\,(u, v)$ je ohodnocení hrany $u \to v$. Z toho, že se změnila $Dist\,(w)$, totiž plyne, že existuje hrana $\langle u, v\rangle = u \to v$.

Na základě těchto úvah již můžeme formulovat algoritmus pro vyhledání nejkratších cest ze zadaného počátečního uzlu do všech ostatních uzlů grafu. Zapíšeme jej v Pseudopascalu, nebot' podrobná formulace by věc spíše zatemnila.

```
type uzel = 1..n;                        {n je počet uzlů v grafu}
var Dist: array [uzel] of real;                {při <i,j>∉U bude}
    Cena: array [uzel, uzel] of real;             {Cena[i,j] =∞ }
procedure NC(v,n: uzel);               {výchozí uzel a počet uzlů}
var S: set of uzel;
    u, num, i, w: uzel;
begin
   for i := 1 to n do Dist[i] = Cena[v, i];        {inicializace}
   S := [v];                          {na počátku obsahuje S pouze v}
   Dist[v] := 0;
   for num := 2 to n do begin              {*1* n-1 cest z uzlu v}
```

„zvol $u$ tak, aby $Dist\,[u] = \min\{Dist\,[w]\}$, minimum přes všechna $w \notin S$“;

```
          S := S + u;                                    {přidej v do S}
```

„ulož nalezenou nejkratší cestu z uzlu $v$ do uzlu $u$“;

```
      for (všechna w, not(w S)) do                       {aktualizace vzdáleností}
        Dist[w] := min(Dist[w], Dist[u]+Cena[u,v]);
   end;
end;
```

Všimněme si ještě časové náročnosti tohoto algoritmu. Cyklus **for** v řádce, označené *1*, se provádí $n$-1-krát. Při hledání minimálního u a při aktualizaci vzdáleností musíme v cyklu projít všechny uzly, které nejsou v $S$. Vnořené cykly tedy budou obsahovat $n-1, n-2, \ldots, 1$ iterací.

Protože $n-1\ +\ n-2\ +\cdots+\ 1 = (n-1)\,n/2$, bude časová náročnost popsaného algoritmu $O\left(n^2\right)$.

## 3.3 Dynamické programování[3]

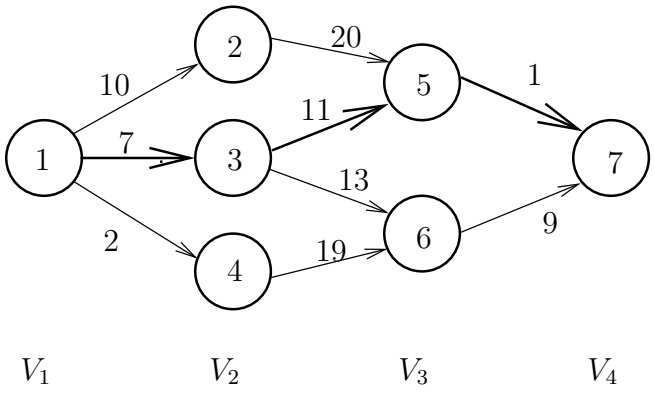
Dynamické programování představuje způsob návrhu algoritmu pro řešení optimalizačního problému, který lze uplatnit v případě, že na řešení daného problému můžeme nahlížet jako na posloupnost rozhodnutí.

Při řešení problémů metodou dynamického programování se opíráme o *princip optimality*. Ten říká, že *optimální posloupnost rozhodnutí má tu vlastnost, že at' je počáteční stav a rozhodnutí jakékoli, musí být všechna následující rozhodnutí optimální vzhledem k výsledkům rozhodnutí prvního.*

Na rozdíl od hladového algoritmu, kde generujeme vždy jen jedinou posloupnost rozhodnutí, může použití dynamického programování vést ke generování více posloupností. Zkoumáme všechny posloupnosti, které by mohly být optimální, a snažíme se vyloučit ty, o kterých je jasné, že optimální nebudou.

Nejjednodušší, ale nejméně účinnou metodou pro řešení problémů, na které se můžeme dívat jako na posloupnost rozhodnutí, je *metoda hrubé síly*: Vygenerujeme všechny možné posloupnosti a mezi nimi vyhledáme posloupnost optimální. Použitím dynamického programování můžeme významně zredukovat počet zkoumaných posloupností.

[3]Termín "programování" se používá mj. jako označení některých optimalizačních úloh. Můžeme se setkat např. s lineárním programováním, celočíselným programováním apod.




Obr. 3.2: Úrovňový graf se 4 úrovněmi

**Příklad 3.4 (použitelnost dynamického programování na problém nejkratší cesty)**

Vezměme ohodnocený orientovaný graf $G = (V, U)$. Úkolem je najít nejkratší cestu z uzlu $i$ do uzlu $j$. Ukážeme si, že tento problém bychom mohli zkusit řešit i metodou dynamického programování.

Řešení tohoto problému představuje posloupnost rozhodnutí: Je-li $i$ prvním uzlem cesty, je třeba určit, který uzel bude druhý, který bude třetí atd.

Ukážeme, že v této úloze platí princip optimality: Je-li $i, i_1, \ldots, i_s, i_{s+1}, \ldots, j$ nejkratší cesta z uzlu $i$ do uzlu $j$, musí být $i, i_1, \ldots, i_s$ nejkratší cesta z uzlu $i$ do uzlu $i_s$ a $i_s$, $i_{s+1}, \ldots, j$ nejkratší cesta z uzlu $i_s$ do uzlu $j$. Pokud by totiž bylo možné najít kratší cestu např. z uzlu $i$ do $i_s$, řekněme $i, l_1, \ldots, i_s$ představovala by posloupnost $i, l_1, \ldots, i_s, i_{s+1}, \ldots, j$ kratší cestu z $i$ do $j$ než je $i, i_1, \ldots, i_s, i_{s+1}, \ldots, j$.

Při hledání nejkratší cesty z $i$ do $j$ budeme postupovat takto: Označíme si $P_j$ množinu všech uzlů sdružených s koncovým uzlem $j$ ($P_j$ je tedy množina všech uzlů, ze kterých vede orientovaná hrana do $j$, tj. $k \in P_j$ právě když $\langle k, j \rangle \in E$). Pro každý uzel $k \in P_j$ bude $\Gamma_k$ nejkratší cesta z $i$ do $k$ (tedy z výchozího uzlu do předposledního, sdruženého s posledním). Podle principu optimality bude nejkratší cesta z $i$ do $j$ nejkratší z cest tvaru $\{\Gamma_k, j \,|k \in P_j\}$. Stojíme tedy před úlohou najít nejkratší cesty do všech uzlů z množiny $P_j$.

Princip optimality zpravidla aplikujeme rekurzivně, takže dynamické programování nás přivede k rekurentním vztahům pro veličiny, které popisují optimální řešení.

Při řešení úloh dynamického programování rozlišujeme *dopředný* a *zpětný* přístup. Nechť $x_1, x_2, \ldots, x_n$ jsou proměnné, o jejichž hodnotách budeme při řešení rozhodovat. Při dopředném přístupu rozhodujeme o hodnotě $x_l$ na základě optimálních rozhodnutí pro $x_{1+1}, \ldots, x_n$, zatímco při zpětném přístupu rozhodujeme o hodnotě $x_l$ na základě posloupnosti optimálních rozhodnutí pro $x_1, \ldots, x_{l-1}$.

**Příklad 3.5: Problém nejkratší cesty v úrovňovém grafu**

*Úrovňový graf s k úrovněmi* je orientovaný graf, jehož množina uzlů $U$ je rozdělena do disjunktních podmnožin (úrovní) $V_1, \ldots, V_k$. Úrovně $V_1$,a $V_k$ obsahují pouze 1 uzel, ostatní jich mohou obsahovat více. Uzel $s$, který leží na úrovni $V_1$, označujeme jako *počáteční*, uzel $t$, který leží na úrovni $V_k$, jako *koncový*.

Hrana, která vychází z uzlu na úrovni $i$, musí končit v uzlu na úrovni $i + 1$. To znamená, že každá cesta z počátečního do koncového uzlu ($s \to t$) musí projít postupně všemi úrovněmi grafu.

Úrovňové grafy mohou popisovat např. varianty technologického procesu a jejich ceny. Příklad jednoduchého úrovňového grafu vidíme na obr. 3.2.

Naším úkolem bude sestavit algoritmus pro nalezení nejkratší cesty z $s$ do $t$. Na rozdíl od obecného grafu zde víme, že nejkratší cesta se bude skládat z $k-1$ hran, přičemž každá hrana bude spojovat uzly na dvou různých úrovních. Pokusíme se toho využít při řešení naší úlohy.

Přitom budeme používat následující označení: $P(i,j)$ bude nejkratší cesta z uzlu $j$ na úrovni $V_i$, $Cena(i,j)$ bude její cena a $c(j,l)$ je ohodnocení hrany $\langle j,l\rangle$. O uzlech grafu předpokládáme, že jsou očíslovány tak, že uzly na úrovni $V_i$ mají nižší pořadová čísla než uzly na úrovni $V_{i+1}$.

Každá cesta z $s$ do $t$ v $k$-úrovňovém grafu je výsledkem $k-2$ rozhodnutí; $i$-té rozhodnutí spočívá v určení, který uzel z úrovně $V_{i+1}$ bude součástí cesty. Naše úloha je zvláštním případem problému nejkratší cesty, o kterém jsme hovořili v příkladu 3.4; proto i zde platí princip optimality. Z něj pro $Cena(i,j)$ dostaneme

$$Cena(i,j) = \min_{l \in V_{i+1}, <l,j> \in U} \{c(i,1) + Cena(i+1,1)\}, \tag{3.2}$$

tj. cenu nejkratší cesty z uzlu $j$ na úrovni $V_i$ do $t$ najdeme jako minimum součtu ceny cesty z uzlu $l$ na úrovni $V_{i+1}$ a ohodnocení hrany $\langle i,l\rangle$. Toto minimum počítáme přes všechny uzly $l$ na úrovni $V_{i+1}$, sdružené s $j$.

Pro úroveň $k-1$ platí $Cena(k-1,j) = c(j,t)$, pokud $\langle j,t\rangle \in U$; jinak položíme $Cena(k-1,j) = +\infty$.

Naši úlohu můžeme řešit tak, že nejprve vypočítáme na základě vztahu (3.2) $Cena(k-2,j)$ pro všechny uzly $j$ $V_{k-2}$, pak $Cena(k-3,j)$ pro všechny uzly $j \in V_{k-2}$ atd. Nakonec dospějeme k $Cena(1,s)$. Podívejme se, jak bychom pomocí tohoto postupu hledali nejkratší cestu v grafu na obr. 3.2.

Ceny cest ze třetí úrovně (uzly 5 a 6) do cílového uzlu 7 jsou $Cena(3,5) = 1$, $Cena(3,6) = 9$. Ceny nejkratších cest ze druhé úrovně (uzly 2, 3 a 4) jsou $Cena(2,2) = 21$ (zde existuje pouze jediná cesta),

$$Cena(2,3) = \min_{l \in \{5,6\}} \{c(3,1) + Cena(3,1)\} = \min\{c(3,5) + Cena(3,5)\,; c(3,6) + Cena(3,6)\} = 12,$$

$Cena(2,4) = 28$ (zde existuje opět pouze jediná cesta).
Pro nejkratší cestu z uzlu 1 do 7 nakonec dostaneme

$$\begin{aligned} Cena(1,j) &= \min_{l \in V_2} \{c(1,1) + Cena(2,1)\} = \min\{c(1,2) + Cena(2,2)\,; \\ &\quad ; c(1,3) + Cena(2,3)\,; c(1,4) + Cena(2,4)\} = 19 \end{aligned}$$

Tato cesta je na obr. 3.2 vyznačena silnějšími čarami.

Na závěr vyjádříme algoritmus pro hledání nejkratší cesty v úrovňovém grafu ve tvaru procedury, kterou opět zapíšeme v Pseudopascalu.

V proceduře $NC$ budeme do pole $D$ ukládat hodnotu $l$, která minimalizuje výraz $c(j,l) + Cena(i+1,l)$ v rovnici (2). To znamená, že $D[i,j]$ bude obsahovat index $l$ uzlu na úrovni $i+1$, přes který vede nejkratší cesta z uzlu $j$ na $i$-té úrovni.

Vzhledem ke způsobu, jakým jsou uzly očíslovány, můžeme vynechat indexy úrovní. Vstupní parametry procedury $NC$ budou: množina hran grafu, počet $n$ uzlů grafu a počet $k$ úrovní grafu. Výstupním parametrem bude pole, obsahující indexy uzlů, které tvoří nejkratší cestu.

```
procedure NejkratšíCesta(U: set of hrana; k,n: integer;
             var P: array [1..k] of integer);
var CENA: array [1..n] of real;
    D:    array [1..n] of integer;
    r,j:  integer;
begin
  for j:=n-1 downto 1 do begin                  {výpočet CENA[j]}
```

„najdi uzel $r$ takový, že $\langle j,r\rangle \in U$ a zároveň $c(j,r) + CENA[r]$ je minimální“;

```
    CENA[j] := c(j,r) + CENA[r];            {cena cesty z uzlu j}
    D[j] := r;                        {optimální cesta vede přes r}
  end;
  {nalezení nejkratší cesty}
  P[1] := 1; P[k]= n;             {první a poslední uzel je jasný}
  for j := 2 to k-1 do
    P[j] := D[P[j-1]]       {v D byl výsledek posunutý o 1 index}
end;
```

Pokusme se ještě odhadnout složitost tohoto algoritmu. V úrovňovém grafu s $k$ úrovněmi je celkem $n$ uzlů. Na první úrovni je 1 uzel, ne druhé úrovni je $n_1$ uzlů, ... na úrovni $k-1$ je $n_{k-1}$ uzlů, na $k$-té -úrovni je 1 uzel. Budeme předpokládat, že z každého uzlu na úrovni $i$ vede cesta do každého z uzlů na úrovni $i+1$.

V uvedeném algoritmu postupně hledáme nejkratší cestu do $t$ z uzlů úrovní $k-1$, $k-2$ atd. Nalezení nejkratší cesty z úrovně $k-1$ znamená projít všech $n_{k-1}$ uzlů této úrovně, tedy $n_{k-1}$ kroků. Nalezení nejkratší cesty z úrovně $k-2$ znamená projít všech $n_{k-2}$ a pro každý z nich zkoumat, přes který z $n_{k-1}$ uzlů úrovně $k-1$ vede nejkratší cesta do $t$, to znamená $n_{k-2}.n_{k-1}$ operací atd. Nalezení nejkratší cesty z úrovně 1 znamená projít všech $n_2$ uzlů úrovně 2 a zjistit, přes který z nich vede hledaní nejkratší cesta.

Celkem tedy dostaneme

$$p = n_2 + n_{k-1} + \sum_{i=2}^{k-1} n_i n_{i-1}$$

operací. Pokud budou např. na všech úrovních stejné počty uzlů, bude pro $i = 2, \ldots, k-1$ počet uzlů na jednotlivých úrovních roven $n_i = (n-2)/(k-2)$. V takovém případě bude

$$p = 2\frac{n-2}{k-2} + (k-2)\left(\frac{n-2}{k-2}\right)^2 = 2\frac{n-2}{k-2} + \frac{(n-2)^2}{k-2}$$

Počet operací vyjde tedy $O\left(n^2\right)$.

## 3.4 Metoda hledání s návratem (backtracking)

Metoda *hledání s návratem* je jednou z metod, založených na prohledávání tzv. stavového stromu problému. Další označení, pod kterými můžeme tuto metodu najít, jsou např. *metoda pokusů a oprav, metoda zpětného sledování, metoda prohledávání do hloubky.*

### 3.4.1 Úvod

Metodu hledání s návratem můžeme použít v případě, že řešením problému je vektor ($n$-tice) $(x_1, \ldots, x_n)$, jehož jednotlivé složky vybíráme z množin $S_i$, $x_i \in S_i$. Zpravidla potřebujeme najít $n$-tici, která minimalizuje nebo maximalizuje nějakou účelovou funkci $P(x_1, \ldots, x_n)$. Můžeme ale také hledat všechny $n$-tice, které splňují podmínku $P(x_1, \ldots, x_n)$.

Označme symbolem $m_i = |S_i|$ mohutnost množiny $S_i$. Potom z nich lze sestavit celkem $M = \prod_{i=1}^{n} m_i$ $n$-tic. Nejjednodušší metoda řešení takovéhoto problému, založená na „hrubé síle“, spočívá v tom, že projdeme všech $M$ možností a z nich vybereme správné řešení.

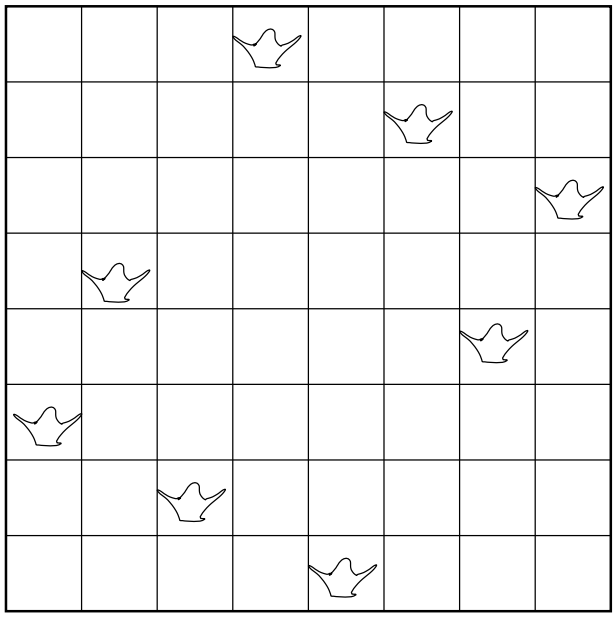
Rozhodneme-li se použít metodu hledání s návratem, vytváříme $n$-tice jednu složku po druhé. Přitom používáme účelovou funkci (případně vhodnou pomocnou funkci, odvozenou od účelové funkce) a pro každou nově vytvořenou složku testujeme, zda by taková $n$-tice vůbec mohla být optimální nebo splňovat dané podmínky.

Jestliže pro nějaké $x_i$ zjistíme, že žádný vektor, začínající $(x_1, \ldots, x_i)$, nemůže být optimální (resp. nemůže splňovat dané podmínky), nebudeme už žádný takový vektor testovat a vezmeme další možnou hodnotu $i$-té složky. Tím vyloučíme z testování dalších $m_{i+1} \times m_{i+2} \times \cdots \times m_n$ vektorů. Jestliže jsme vyčerpali všechny možné hodnoty $i$-té složky, vrátíme se o jednu úroveň zpět a budeme zkoušet další možnou hodnotu $x_{i-1}$. (To je onen „návrat“, o kterém se hovoří v názvu metody.)

Jako *řešení* označíme $n$-tici, která vyhovuje daným omezením (příp. optimalizuje účelovou funkci).
Poznamenejme, že počet složek vektoru řešení nemusí být pro všechna řešení stejný; to znamená, že jedno řešení může obsahovat $n_1$ složek, jiné $n_2$ složek.

**Příklad 3.6 (problém osmi dam)**

Problém 8 dam je klasická kombinatorická úloha, kterou formuloval již kolem r. 1850 německý matematik K. F. Gauss. Úkolem je postavit na šachovnici 8 dam tak, aby žádná z nich podle šachových pravidel neohrožovala jinou (přitom na rozdíl od „normálního“ šachu předpokládáme, že všechny dámy jsou rovnocenné a všechny



Obr. 3.3: jedno z možných řešení problému 8 dam

jsou navzájem nepřátelské, tedy každá z nich může ohrožovat kteroukoli ze zbývajících). Jedno z 92 možných řešení vidíte na obr. 3.3.

Dále se setkáme s obecnějším problémem $n$ dam, ve kterém je dána šachovnice s $n \times n$ poli, na kterou je třeba umístit $n$ dam, aby se navzájem neohrožovaly. Pro $n = 1$ tato úloha nemá význam; snadno ukážeme, že pro $n = 2, 3$ tato úloha nemá řešení. Nejmenší hodnota, pro kterou se s ní budeme zabývat, bude $n = 4$. Nyní se však vrátíme k původnímu zadání pro $n = 8$.

Obecně lze 8 dam umístit na šachovnici $\binom{64}{8} = 4426165368$ způsoby. Je ale jasné, že každá z dam musí být v jiném sloupci šachovnice. To znamená, že řešení naší úlohy můžeme vyjádřit jako osmici $(x_1, \ldots, x_8)$, kde $x_i$ budou čísla řádků. Např. řešení, které vidíme na obr. 3.3, lze vyjádřit vektorem (4,6,8,2,7,1,3,5).

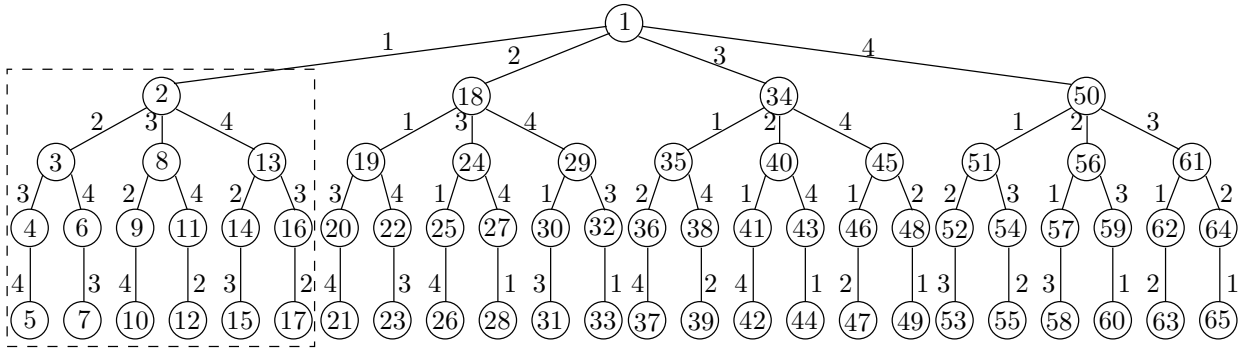
Odtud plyne podmínka: $S_i = \hat{8} = \{1, \ldots, 8\}$ pro každé $i = \hat{8}$ . To omezuje množinu možných osmic na pouhých 16777216.

Z pravidel šachu plynou další omezení: Žádné dvě dámy nesmějí ležet ve stejné řádce nebo na stejné diagonále. První z těchto podmínek zmenšuje prostor možných osmic na $8! = 40320$.

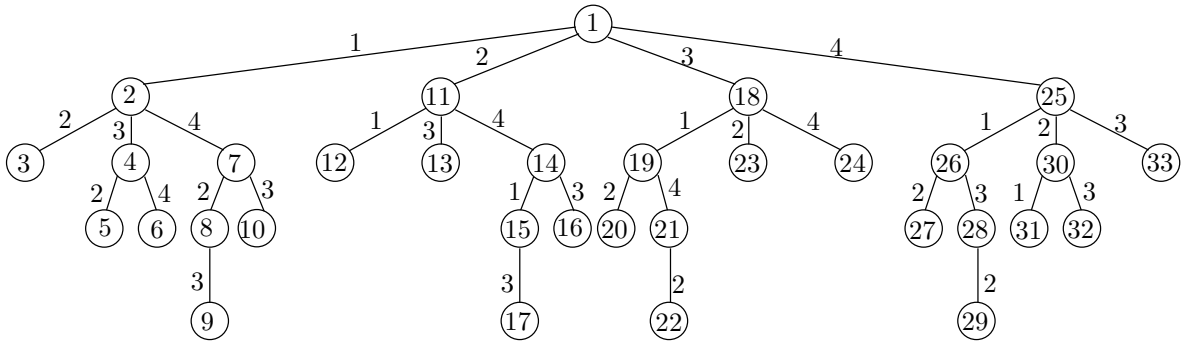
Podívejme se na postup řešení. Začneme tím, že umístíme první dámu do prvního řádku; vektor řešení tedy bude začínat $(1, \ldots)$. Z podmínky, že žádné dvě dámy nesmějí ležet na téže řádce nebo na téže diagonále, dostáváme, že nemá smysl dále zkoumat žádnou $n$-tici, která by začínala $(1, 1, \ldots)$ nebo $(1, 2, \ldots)$. Zkusíme tedy $(1, 3, \ldots)$ a pokusíme se umístit další dámu do třetího sloupce. Podobně uvažujeme i při obsazování ostatních pozic vektoru řešení.

Postup při použití metody hledání s návratem lze snadno znázornit pomocí stavového stromu problému. Kořen stromu odpovídá počátečnímu stavu; každá z hran, které vycházejí z kořene, odpovídá jedné možné volbě první složky řešení $x_1$. Obecně vrcholy stromu na $i$-té úrovni odpovídají stavům problému ve chvíli, kdy známe $(x_1, \ldots, x_{i-1})$, a hrany vycházející z vrcholu na $i$-té úrovni odpovídají možným hodnotám $x_i$. Všechny cesty od kořene do ostatních vrcholů popisují *stavový prostor problému*.

Metodu hledání s návratem pak můžeme popsat jako postupné procházení stavového stromu problému a jeho postupné „prořezávání“: Jakmile o nějaké větvi zjistíme, že nemůže vést k optimálnímu řešení, přestaneme ji procházet.
Na obrázku 3.4 vidíme stavový strom problému 4 dam, ve kterém bereme v úvahu podmínku, že řešení musí být permutací množiny $\{1, 2, 3, 4\}$.



Obr. 3.4: Úplný stavový strom problému 4 dam



Obr. 3.5: Stavový strom problému 4 dam, generovaný při hledání s návratem

Čárkovaně je označen jeden z podstromů. Vrcholy jsou očíslovány tak, jak by se procházely při úplném prohledávání. Hrany jsou označeny hodnotami, které bychom v odpovídajících krocích volili. Metoda hledání s návratem umožňuje vyhnout se prohledávání některých podstromů a tak zrychlit řešení problému. Na obr. 3.5 vidíte týž strom, „prostříhaný“ metodou hledání s návratem. Obsahuje všechny generované stavy; stavy, které představují řešení, jsou vyznačeny tučně.

Poznamenejme, že v pro některé problémy nemusí být stavový strom symetrický.
Z toho plyne, že řešení jsou ty stavy $S$, pro které cesta od kořene k $S$ definuje $n$-tici vyhovující všem omezením, případně optimalizující účelovou funkci (na obrázku 3.4 jsou to pouze listy).

Strom stavového prostoru (stavový strom) problému se skládá z podstromů; tomu odpovídá skutečnost, že stavový prostor problému lze rozdělit na podprostory. Přitom každému prvku prostoru řešení odpovídá nějaký vrchol ve stavovém stromě.

### 3.4.2 Podrobnější formulace pro zvláštní případ

V tomto odstavci sestavíme obecnou formulaci metody hledání s návratem pro případ, kdy hledáme všechny stavy, které vyhovují daným omezením.

Buď $(x_1, \ldots, x_i)$ cesta od kořene k nějakému vrcholu $x_i$ ve stavovém stromě. Označíme symbolem $T(x_1, \ldots, x_i)$ množinu všech možných hodnot $x_{i+1}$ takových, že $(x_1, \ldots, x_i)$ je také cesta pro nějaký stav problému.

Dále předpokládáme, že omezující podmínky můžeme vyjádřit jako predikáty (logické funkce) $B_i$ takové, že $B_{i+1}(x_1, \ldots, x_{i+1})$ je nepravdivé pro cestu $(x_1, \ldots, x_{i+1})$, kterou nelze prodloužit tak, abychom dostali řešení. To znamená, že kandidátem na obsazení pozice $i + 1$ ve vektoru řešení $X$ budou ty hodnoty $x_{i+1}$, které vygeneruje $T$ a které splňují $B_{i+1}$.

Proceduru, popisující metodu hledání s návratem, lze formulovat jak rekurzivně tak i nerekurzivně. Ukážeme si obě možnosti. V obou budeme předpokládat, že $X$ je globální pole, ve kterém se konstruují jednotlivá řešení. Jakmile procedura dospěje k řešení, uloží je (vytiskne, zakresí ...) a pokračuje dál.

```
var X: array [1..n] of stav;            {deklarace pro obě verze}
procedure BT;                                {nerekurzivní verze}
var k: integer;
begin
   k := 1;
   while k > 0 do begin
     if(„zbývá nevyzkoušené X [k] ∈
T (X [1] , . . . , X [k − 1]) takové, že B_k (X [1] , . . . , X [k − 1]) = true“)
     then begin
       if(„X [1] , . . . , X [k − 1] tvoří odpověď“) then Ulož(X) else
       k := k+1                          {vezmi následující možnost}
     end else
       k := k-1;                      {vrat' se k předchozí možnosti}
   end;
end;
```

Rekurzivní verze je jednodušší:

```
procedure BT(k: integer);                        {rekurzivní verze}
begin
  „pro všechna X [k] taková, že X [k] ∈ T (X [1] , . . . , X [k − 1]) ∧B_k (X [1] , . . . , X [k]) = true“ do
      begin
        if („X [1] , . . . , X [k] tvoří odpověď“) then Ulož(X) else BT(k+1)
      end;
end;
```

Poznámky k rekurzivní verzi: Jednotlivá volání procedury $BT$ najdou vždy nejvýše jednu složku vektoru řešení. V cyklu „pro všechna $X[k]\ldots$“ se vypíše výsledek, pokud se nějaký našel; jinak se rekurzivně zavolá procedura $BT$ s hodnotou $k+1$, tj. bude se hledat další složka. Pokud vhodné $X[k]$ neexistuje, nestane se vůbec nic a toto rekurzivní volání procedury $BT$ skončí, čímž se automaticky vrátíme o úroveň zpět.

Rekurzivní formulace vychází přirozenější (podobně jako většina algoritmů, které se nějak týkají stromů). Časová náročnost algoritmu závisí struktuře stavového stromu, na náročnosti generování nových stavů a testování podmínek a především na úspěšnosti a na včasnosti „odřezávání“ větví stavového stromu, které nevedou k řešení - tedy na „síle“ podmínek $B_i$.

## 3.5 Obecné metody prohledávání stavového stromu

Z výkladu v předchozím odstavci je zřejmé, že metoda hledání s návratem představuje jen jednu z možností, jak procházet stavový strom daného problému. Podívejme se tedy na některé další možnosti.

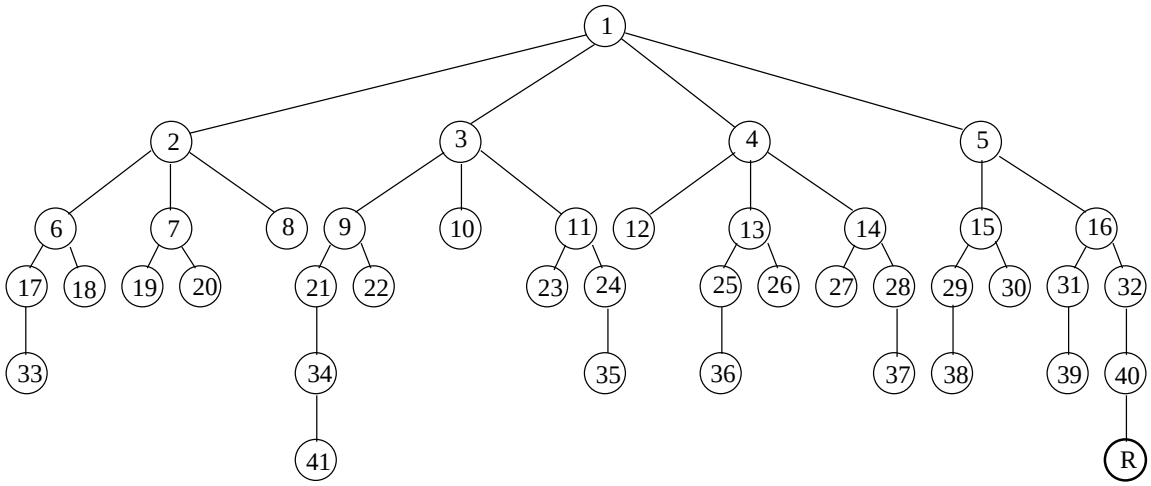
Nejprve zavedeme několik názvů.

Jako *živý* označíme vrchol stavového podstromu, který jsme již generovali (tj. generovali jsme odpovídající stav problému), ale od kterého dosud nebyli generováni všichni potomci. *Rozvíjený vrchol* je vrchol, jehož potomky právě generujeme. *Mrtvý vrchol* je vrchol, jehož všichni potomci již byli generování nebo který již nebude dále rozvíjen z jiných důvodů.

Všechny metody řešení daného problému, založené na prohledávání stavového stromu, vyžadují, abychom si uchovávali seznam živých vrcholů. Při procházení stromu můžeme postupovat různými způsoby.

První možný přístup představuje metoda hledání s návratem (backtracking) a spočívá v tom, že jakmile vygenerujeme potomka $P$ právě rozvíjeného vrcholu $R$, stane se $P$ rozvíjeným vrcholem. Po vyčerpání všech potomků vrcholu $P$ se stane $R$ znovu rozvíjeným uzlem. (Odtud pochází alternativní označení *prohledávání do hloubky*).

Druhou krajnost představuje *metoda prohledávání do šířky*. Při ní zůstane rozvíjený vrchol rozvíjeným vrcholem až do úplného vyčerpání všech potomků. Nově vygenerované vrcholy se ukládají do fronty. Jakmile se vygenerují všichni bezprostřední potomci aktuálního rozvíjeného vrcholu, vezme se jako další rozvíjený vrchol ten, který




Obr. 3.6: Stavový strom problému s jediným řešením a jeho prohledávání do šířky

je ve frontě na řadě. Při prohledávání do šířky tedy procházíme stavový strom „po patrech“. Ve stromu na obrázku 3.6 je vyznačeno pořadí, v jakém bychom touto metodou generovali jednotlivé vrcholy.

Při prohledávání do šířky používáme - podobně jako při prohledávání do hloubky - omezujících podmínek k „zabíjení“ živých vrcholů, to znamená k určení, zda daný vrchol může vést k řešení.

Jak metoda prohledávání do šířky tak i metoda prohledávání do hloubky pevně předepisují pořadí procházení stavů problému.

Podívejme se na stavový strom na obr. 3.6 Jediné řešení daného problému je na něm vyznačeno písmenem „R“. Použijeme-li prohledávání do hloubky, bude záležet na tom, jak očíslujeme stavy (tedy zda budeme stavový strom procházet z levé nebo z pravé strany). V nejlepším případě najdeme řešení po 5 krocích, v nejhorším po 37 krocích, jako úplně poslední možnost.

Při prohledávání do šířky najdeme řešení jako poslední.

Úplné prohledávání stavového stromu může být prakticky neproveditelné. Nejznámějším příkladem problému, který takto (na současných počítačích) nelze řešit, je šachová hra. Její stavový strom je sice konečný, avšak velice rozsáhlý. Udává se, že průměrný faktor větvení tohoto stromu je cca 38 (to znamená, že po každém půltahu má soupeř průměrně 38 možností, jak pokračovat, a tedy že z každého vrcholu vychází v průměru 38 hran [21]). Úplné prohledání stavového prostoru na 15 půltahů dopředu (tedy zdaleka ne celého stavového stromu šachové hry) by pak vyžadovalo zpracovat více než $4,910^{23}$ možností. Kdybychom zvládli jeden milión možností za sekundu, potřebovali bychom k prohledání vymezeného podprostoru více než $10^{10}$ let, což je doba srovnatelná se stářím vesmíru podle dnešních odhadů.
Předchozí příklad ukazuje, že pevně předepsané pořadí prohledávání nemusí být výhodné. Proto se často hledá „rozumná“ váhová funkce $F$, definovaná na živých vrcholech, podle které bychom se mohli rozhodnout, který uzel bude nejvýhodnější rozvíjet jako následující. Hodnota $F(C)$ této funkce pro vrchol $C$ představuje například odhad pravděpodobnosti, že přes $C$ vede cesta k řešení.

Při rozhodování, který z vrcholů $C_i$ budeme rozvíjet jako následující, můžeme použít přímo hodnot $F(C_i)$. Lze-li pomocí funkce $G$ odhadnout pro vrchol $C_i$ počet operací, potřebný k dosažení odpovědi z daného vrcholu, můžeme při rozhodování o následujícím rozvíjeném vrcholu vycházet hodnoty součtu $f(F(C_i)) + G(C_i)$, kde $f$ je vhodná neklesající funkce.

V ideálním případě by $F(C_i)$ udávalo vzdálenost vrcholu $C_i$ od řešení. Taková funkce by vedla přímo k požadovanému výsledku; její nalezení je ovšem úloha ekvivalentní řešení původní úlohy. Proto se při volbě následujícího rozvíjeného vrcholu používá odhadu, při jehož konstrukci se vychází z částečných znalostí o řešení dané úlohy.

Tento postup se zpravidla označuje jako *metoda nejlacinější cesty* (*least cost path method*).

# Kapitola 4

# Rekurze

Jako *rekurzivní* označujeme strukturu, které obsahuje odkaz na sebe nebo na strukturu stejného druhu. Z reklam známe např. rekurzivní obrázky: dívka předvádí počítač, na jehož obrazovce vidíme tutéž dívku, jak předvádí týž počítač atd.

Ve 2. kapitole jsme se setkali s rekurzivními datovými strukturami: každá složka stromu nebo seznamu obsahuje odkaz (ukazatel) na složku stejného typu.

V této kapitole se budeme zabývat rekurzí v algoritmech a v odpovídajících programových strukturách - v procedurách a funkcích.

## 4.1 Rekurzivní algoritmy a podprogramy

Rekurzivní algoritmus dostaneme, jestliže při rozkladu problému na elementární kroky dojdeme k problému stejného druhu, ale (v nějakém smyslu) menšího rozsahu.

Rekurzivita algoritmu je často důsledkem rekurzivní povahy zpracovávaných dat. Připomeňme si např. algoritmus zpracování binárního stromu, který je přirozeně rekurzivní:

1. Zpracujeme údaj v kořeni.

2. Pokud má daný strom levý podstrom, zpracujeme ho.

3. Pokud má daný strom pravý podstrom, zpracujeme ho.

Ve druhém a třetím kroku se zde odvoláváme na tento algoritmus jako celek („zpracujeme podstrom“). Řešíme tedy týž problém, ovšem pro menší množinu dat.

Obecný tvar rekurzivního algoritmu $P$ bychom mohli popsat formulí

$$P \equiv C\left[S_i, P\right], \tag{4.1}$$

kde $C$ znamená kompozici kroků $S_i$, které neobsahují odkaz na $P$, a samotného algoritmu $P$. Připomeňme si ale, že i rekurzivní algoritmus musí být konečný. To znamená, že odkaz na sebe sama v (4.1) musí být vázán na splnění určité podmínky, kterou označíme $B$:

$$P \equiv C\left[S_i, \text{if } B \text{ then } P\right]. \tag{4.2}$$

V mnoha případech je rekurzivní volání vázáno na hodnotu přirozeného čísla $n$, takže podmínka (4.2) má tvar

$$P\left(n\right) \equiv C\left[S_i, \text{if } n > 0 \text{ then } P\left(n-1\right)\right].$$

Tato formulace zdůrazňuje, že počet rekurzivních volání musí být konečný.

### 4.1.1 Rekurze v programu

Rekurzi v programu realizujeme na úrovni podprogramů, tj. procedur a funkcí. Jako rekurzivní označujeme takové volání procedury (funkce), při kterém dojde k aktivaci těla podprogramu dříve, než se ukončí aktivace předchozí.

Někdy je vhodné rozlišovat *přímou* a *nepřímou* rekurzi. Jako přímou rekurzi označujeme situaci, kdy tělo podprogramu $f$ obsahuje volání sebe sama. Nepřímá rekurze nastane, jestliže podprogram $f$ volá podprogram $g_1$, podprogram $g_1$ volá podprogram $g_2, \ldots,$ podprogram $g_n$ volá $f$. Vzhledem k nepřímé rekurzi nebo k možnosti volat podprogram pomocí ukazatelů není možné prohlídkou zdrojového textu zjistit, zda se v programu může vyskytnout rekurzivní volání.

Každé volání podprogramu znamená vytvoření lokálních proměnných a přidělení místa pro skutečné parametry. Jako skrytý parametr se při volání podprogramu předává také návratová adresa. Při několikanásobném rekurzivním volání procedury se tyto lokální objekty vytvářejí zvlášť pro každou aktivaci. Vzhledem k tomu, že paměť dnešních počítačů není nekonečná (a stěží kdy bude), může se snadno stát, že rekurzivní program skončí chybou, když vyčerpá veškerou volnou paměť. Proto jestliže napíšeme rekurzivní podprogram, musíme nejen vědět, že počet rekurzivních volání bude konečný, ale také že hloubka rekurze nebude velká.

**Příklad 4.1**

Jednoduchým příkladem na rekurzi může být funkce faktoriál. Z definice

$$n! = \prod_{i=1}^{n} i$$

pro $n > 0$, $n! = 1$ pro $n = 0$
plyne vztah $n! = n\,(n-1)!$

Rekurzivní funkce pro výpočet faktoriálu nezáporného celého čísla by tedy mohla mít tvar

```
function f(n: integer): integer;
begin
   if n = 0 then f := 1
   else f := n*f(n-1)
end;
```

Na první pohled je zřejmé, že rekurzivní výpočet faktoriálu není příliš výhodný. Daleko efektivnější je vynásobit čísla $1, \ldots, n$ v cyklu.

**Rekurze a programovací jazyky**

Většina dnes používaných programovacích jazyků připouští rekurzivní volání podprogramů. Mezi výjimky patří např. Fortran.

Při nepřímé rekurzi se nelze vyhnout situaci, kdy jeden z podprogramů jiný podprogram, který jsme ještě nedefinovali. Ke správnému překladu volání podprogramu ovšem stačí, aby překladač znal rozhraní (hlavičku) volané funkce. Překladači Pascalu ji sdělíme pomocí deklarace s direktivou *forward*, v ANSI C a v C++ použijeme prototyp funkce.

Programovací jazyky, které neprovádějí kontrolu typů předávaných parametrů, nemusí předběžné informace požadovat, (jako je tomu ve starších variantách jazyka C).

### 4.1.2 Kdy se rekurzi vyhnout

Jestliže pracujeme s rekurzivními daty nebo vycházíme z rekurzivních definic, dospějeme obvykle zcela přirozeně k rekurzivním algoritmům. To ale neznamená, že rekurzivní implementace je nejlepším řešením (pokud nám ji programovací jazyk vůbec dovoluje).

Typický algoritmus, u kterého lze se rekurzi vyhnout, můžeme charakterizovat pomocí schématu

$$P = [S, \text{if } B \text{ then } P]\,. \tag{4.3}$$

(v tomto zápisu závorky [ ] předepisují sekvenční provedení operací, které jsou v nich zapsány). Takováto schémata se zpravidla objevují u výpočtů na základě jednoduchých rekurentních vztahů jako je faktoriál v příkladu 4.1.

Schéma (4.3) předepisuje provést kroky $S$, a pokud je splněna podmínka $B$, zavolat $P$ - tedy znovu provést $S$, a pokud je splněna podmínka $B$, zavolat ... atd. Je tedy zřejmé, že schéma (4.3) je ekvivalentní iterativnímu schématu

$$P = [S, \text{while } B \text{ do } S]\,. \tag{4.4}$$

které vede k efektivnějšímu programu.

Jiný, trochu složitější příklad, kdy je použití rekurze nevýhodné, představují výpočty založené na rekurentních vztazích tvaru

$$x_n = f\left(x_{n-1}, x_{n-2}, \ldots, x_{n-k}\right) \tag{4.5}$$

ve kterých známe $x_0, \ldots, x_k$. Je jasné, že při výpočtu $x_n$ můžeme postupovat „shora dolů“, tj. rekurzivně, nebo „zdola nahoru“, iterativně. Následující příklad nám však ukáže, že rekurzivní výpočet bývá v tomto případě mimořádně neefektivní.

**Příklad 4.2**

Jednoduchým příkladem rekurentní posloupnosti, zadané schématem tvaru (4.5), jsou Fibonacciova čísla $f_n$, definovaná takto:

$$f_{n+1} = f_n + f_{n-1} \quad \text{pro} \quad n > 1, f_0 = 0, f_1 = 1.$$

Podívejme se, jak to dopadne, naprogramujeme-li výpočet Fibonacciových čísel jako rekurzivní funkci:

```
function F(n: integer): integer;        {Fibonacciova čísla jako}
begin                                      {programátorský horor}
  if (n = 0) or (n = 1) then f := n
  else f := f(n-1) + f(n-2)
end;
```

Výpočet $f(n)$ pro jakékoli $n > 1$ znamená automaticky další dvě volání $f$. Označíme-li $p_n$ počet volání funkce $f$ potřebný pro výpočet $f_n$, bude pro $p_n$ platit rekurentní vztah

$$p_n = p_{n-1} + p_{n-2} + 1 \tag{4.6}$$

Protože při výpočtu $p_{n-1}$ budeme muset počítat i $p_{n-2}$, musí být $p_{n-1} > p_{n-2}$, takže ze vztahu (4.6) plyne

$$p_n > 2p_{n-2}. \tag{4.7}$$

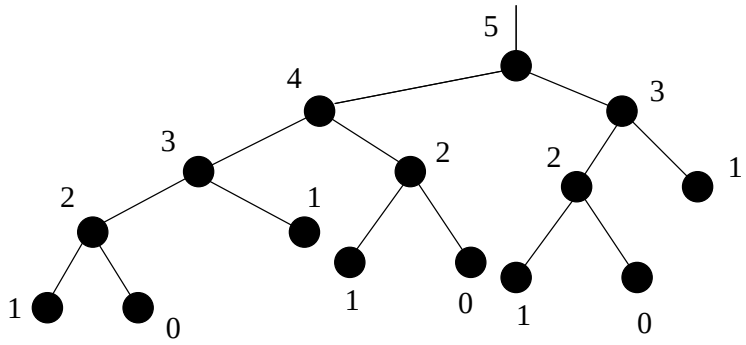
Protože $p_0 = p_1 = 1$, plyne ze (4.7), že počet volání $p_n$ roste rychleji než posloupnost $2^{n/2}$. Graf rekurzivního volání pro $n = 5$ vidíte na obr. 4.1. Všimněte si, že hodnota $f(0)$ se počítá třikrát a hodnota $f(1)$ dokonce pětkrát!

Použijeme-li nerekurzivního výpočtu, např.

```
function F(n: integer): integer; {Fibonacciova čísla }
var a,b,c,i: integer;            {nerekurzivně}
begin
  if (n = 0) or (n = 1) then f := n
  else begin
    a := 0; b := 1;
    for i:=2 to n do begin
      c := a + b; a := b; b := c;
  end;
  f := c;
end;
```

bude se každá z hodnot posloupnosti počítat pouze jednou.
S rekurentními vztahy tohoto typu se setkáme v matematice poměrně často. Např. Besselovy funkce vyhovují rekurentnímu vztahu

$$J_{v+1}(x) = \frac{2v}{x} J_v(x) - J_{v-1}(x)\,, \qquad v \in R.$$




Obr. 4.1: Graf rekurzivních volání při výpočtu $f(5)$

Podobné vztahy platí i pro další speciální funkce. Jiným příkladem mohou být kombinační čísla, která splňují rekurzivní vztah

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}.$$

## 4.2 Jak odstranit rekurzi

Rekurzivní algoritmy, jak už víme, vznikají často jako přirozený důsledek postupu shora dolů při návrhu algoritmu. Někdy je ovšem nezbytné transformovat algoritmus do nerekurzivní podoby - např. proto, že použitý programovací jazyk ji nedovoluje.

Ukážeme si jednoduchý postup, s jehož pomocí odstraníme z podprogramu rekurzivní volání (jde o přímou rekurzi, která se vyskytuje podstatně častěji než rekurze nepřímá). Následující postup je založen na využití zásobníku. Pro jednoduchost v něm budeme pod označením „procedura“ rozumět i funkci.

1. Na počátku procedury deklarujeme zásobník jako globální objekt; také ukazatel na vrchol zásobníku bude globální. Tento zásobník bude sloužit k ukládání parametrů, lokálních proměnných, návratových adres a vypočtených hodnot při rekurzivním volání.
2. Před první příkaz těla procedury vložíme návěští $L_1$.

   Dále každé rekurzivní volání dané procedury nahradíme následující posloupností příkazů:
3. Uložíme na zásobník lokální proměnné a formální parametry.
4. Vytvoříme $i$-té nové návěští $L_i, i = 2, \ldots,$ a hodnotu $i$ uložíme do zásobníku. Tuto hodnotu později použijeme ke stanovení návratové adresy. Vytvořené návěští umístíme v kroku 7.
5. Vyhodnotíme skutečné parametry nového volání a uložíme je do formálních parametrů (nikoli do zásobníku).
6. Vložíme nepodmíněný skok na počátek procedury, na návěští $L_1$.
7. Jestliže takto upravujeme funkci, umístíme návěští, vytvořené ve 4. kroku, k příkazu, kterým vyjmeme hodnotu funkce ze zásobníku, a připojíme kód, který tuto hodnotu v rekurzivní funkci využívá. V proceduře toto návěští připojíme k prvnímu příkazu bezprostředně za skokem, vloženým v 6. kroku.

   Na konci procedury provedeme tyto úpravy:
8. Je-li zásobník prázdný, skončíme.
9. Jinak vezmeme aktuální hodnoty výstupních parametrů a předáme je odpovídajícím proměnným na vrcholu zásobníku (tím vracíme vypočtené hodnoty parametrů do předchozího volání).
10. Vložíme kód, který odstraní ze zásobníku index návratové adresy (pokud tam nějaký je) a uložíme jej do nepoužité proměnné.
11. Vyjmeme ze zásobníku hodnoty lokálních proměnných a parametrů a přidělíme je odpovídajícím proměnným.

12. Je-li to funkce, vložíme instrukce pro vyhodnocení vracené hodnoty a výsledek uložíme na vrchol zásobníku.

13. Index návratové adresy použijeme ke skoku na příslušné návěští $L_i$.

**Příklad 4.3**

Vezmeme rekurzivní verzi funkce faktoriál z příkladu 4.2 a upravíme ji podle předchozího návodu. Přitom budeme předpokládat, že máme k dispozici objektový typ *zásobník* na ukládání celých čísel s konstruktorem *vytvoř* a s metodami *vlož, vyjmi a prázdný*. V následujícím výpisu odkazují čísla v komentářích na kroky předchozího návodu.

Citlivější čtenáře bych rád předem upozornil, že výsledek je v příkrém rozporu s pravidly programátorské slušnosti. Je ale jasné, že takovouto transformací získáme polotovar, který budeme dále upravovat.

```
function f(n: integer): integer;
var a: integer;
    z: zásobník;
label L1, L2;
begin
z.Vytvoř;                                                    {1}
L1:                                                          {2}
    if n = 0 then a := 1        {vypočtenou hodnotu uložíme do a}
    else begin
      z.vlož(n);                                             {3}
      n := n-1;                                              {5}
      goto L1;                                               {6}
L2:                                                          {7}
      a := z.vyjmi;       {vyjmeme výsledek z předchozího volání}
     a := a*n;                                  {a použijeme jej}
    end;
    if z.prázdný then f := a                                 {8}
    else begin
      n := z.vyjmi;                                         {11}
      z.vlož(a);                                            {12}
      goto L2;                                              {13}
    end;
end;
```

Vzhledem k tomu, že funkce $f$ obsahovala jediné rekurzivní volání, nepotřebujeme ukládat návratovou adresu na zásobník - ta je vždy stejná. Odpadá tedy téměř celý krok 4 a krok 10. Funkce nemá žádné výstupní parametry, takže odpadá i krok 9.

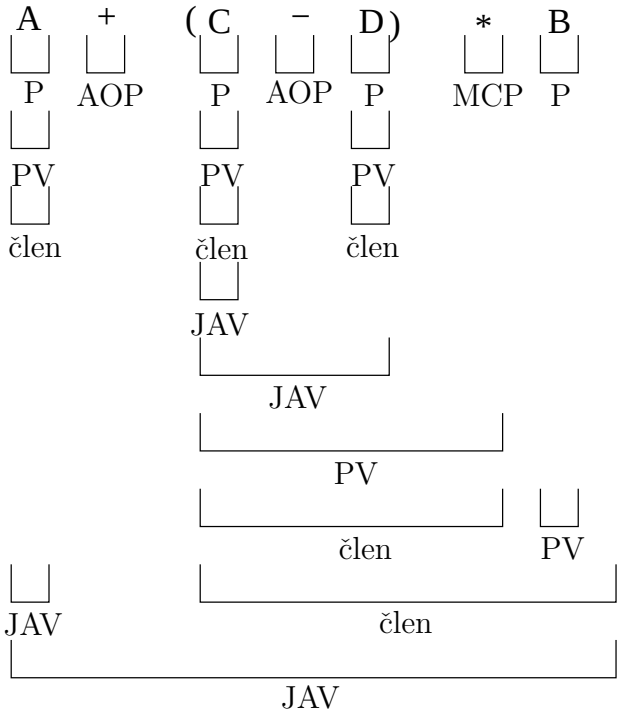
Výsledek, vypočtenou hodnotu, ukládáme do pomocné proměnné $a$. To není původní lokální proměnná, takže ji ve 3. kroku nemusíme ukládat do zásobníku.

Jak jsme již řekli v úvodu, získáme touto transformací polotovar, který můžeme (a musíme) ještě dále upravovat. Například z toho, že $f$ obsahovala jediné rekurzivní volání, také plyne, vypočtená hodnota, která je v $a$ a kterou ve 12. kroku ukládáme na zásobník, je vždy stejná jako hodnota, která je v $a$ a kterou vyjímáme ze zásobníku v 7. kroku. Můžeme tedy vypustit instrukce *z.vlož*($a$) a *z.vyjmi*. Dále se můžeme pokusit odstranit příkazy **goto** a nahradit je cykly[1] atd. Často se tak podaří dospět k rozumné iterativní verzi daného algoritmu.

## 4.3 Další příklady

Na závěr kapitoly o rekurzi si ukážeme několik příkladů.

[1] V našem případě dokonce musíme, pokud budeme uvedený program překládat překladačem, který se řídí normou ISO jazyka Pascal. Tato norma, jak známo, zakazuje skok dovnitř složeného příkazu, takže konstrukce **goto L2** je syntakticky nesprávná.




Obr. 4.2: Syntaktická analýza jednoduchého aritmetického výrazu (P znamená proměnnou)

### 4.3.1 Syntaktická analýza

Syntaktické definice programovacích jazyků jsou obvykle přirozeně rekurzivní. Proto také můžeme očekávat, že překladače budou při syntaktické analýze používat rekurzivních procedur.

Jako příklad vezmeme syntaktickou definici *jednoduchého aritmetického výrazu* (JAV). Za jednotlivými syntaktickými kategoriemi uvedeme v závorkách zkratky, pomocí kterých se na ně budeme dále odvolávat. Symbol „|“ slouží k oddělování jednotlivých alternativ, *ČBZ* znamená „celé číslo bez znaménka“.

| | |
|---|---|
| *aditivní_operátor* (*AOP*): | + \| - |
| *multiplikativní_operátor* (*MOP*): | * \| / |
| *prvotní_výraz* (*PV*): | *ČBZ* \| *proměnná* \| (*JAV*) |
| *člen*: | *PV* \| *člen MOP PV* |
| *JAV*: | *člen* \| *AOP člen* \| *JAV AOP člen* |

Při překladu potřebujeme určit všeobecnou syntaktickou třídu daného řetězce znaků. Na obrázku vidíme příklad analýzy řetězce „A + (C - D)*B“.
Analyzátor zpracovává zdrojový soubor znak po znaku. Může to být např. procedura, jejímiž vstupními parametry jsou ukazatel na znakový řetězec (vstupní data) a cíl (syntaktická kategorie). Jestliže lze znaky, ležící bezprostředně za místem, určeným ukazatelem, chápat jako instanci cíle, vrátí výsledek ANO (true) a ukazatel se posune o jeden znak dopředu, jinak vrátí NE (false) a ukazatel se nezmění.

Podívejme se, jak by probíhala analýza řetězce „A + (C - D)*B“ popsaným analyzátorem. Šipkou „↑“ označíme pozici ukazatele ve vstupním řetězci, šipkou „→“ volání analyzátoru.

```
→  A + (C - D) * B     cíl: JAV
   ↑ JAV může být člen
   →  A + (C - D) * B   cíl: člen
      ↑ člen může být prvotní výraz (PV)
      →  A + (C - D) * B     cíl: PV
         ↑  „A“ je instance kategorie „proměnná“, PV
         může být proměnná - vrať ANO, nalezen PV
      ANO, nalezen člen
   ANO, nalezen JAV
```

Nyní si ale musíme položit otázku: lze najít delší řetězec, který by též byl JAV? Podle syntaktické definice zkusíme konstrukci „JAV AOP člen“.

→ + (C - D) * B cíl: AOP
↑ ANO, nalezen AOP
→ (C - D) * B cíl: člen
↑ člen může být PV; PV může začínat „(“, musí to být konstrukce „(JAV)“
→ C - D) * B cíl: JAV
↑ „C“ je proměnná, to je PV - vrať ANO

Lze najít delší řetězec, který by byl též JAV? Zkusíme možnost „JAV AOP člen“.

→ - D) * B cíl: AOP
↑ ANO, AOP nalezen
→ D) * B cíl: člen
↑ „D“ je proměnná, tedy PV, tedy člen - ANO

Lze najít delší řetězec, který by byl též JAV?

→ ) * B cíl: AOP
↑ NE

Našli jsme nejdelší JAV. Aby to byl PV, musí končit závorkou.

→ ) * B cíl: „)“
↑ ANO

Našli jsme prvotní výraz, tedy člen. Lze najít delší člen? Jediná možná konstrukce je „člen MOP PV“.

→ * B cíl: MOP
↑ ANO, nalezen MOP
→ B cíl: PV
↑ „B“ je proměnná, tedy PV - ANO, nalezen PV

ANO, byl nalezen JAV a řetězec byl plně analyzován.

### 4.3.2 Ackermannova funkce

Na závěr si ukážeme, jak se také může chovat rekurzivně definovaná funkce. Ackermannova funkce je dána předpisem

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

Její definice je zdánlivě jednoduchá. Pokud ji budete chvíli zkoumat nebo pokud si ji naprogramujete, zjistíte, že už pro velmi nízké hodnoty parametrů se vyčerpají možnosti počítače. V následující tabulce najdete pro několik hodnot parametrů hodnotu funkce, počet rekurzivních volání, maximální hloubku rekurze a maximální hodnotu parametru $n$, se kterým je funkce volána.

| **m** | **n** | $A(m,n)$ | počet volání | max. hloubka | max. $n$ |
|---|---|---|---|---|---|
| 0 | $n$ | $n+1$ | 1 | 1 | $n$ |
| 1 | $n$ | $n+2$ | $2n+2$ | $n+1$ | $n+2$ |
| 2 | $n$ | $2n+3$ | $2n^2+7n+5$ | $2n+4$ | $2n+2$ |
| 3 | $n$ | $8.2^n-3$ | $f_n$ | $8.2^n-1$ | $8.2^n-4$ |

Tab. 4.1 Vlastnosti Ackermannovy funkce

V tab. 4.1 je $f_n$ posloupnost, rostoucí přibližně jako $4^n$, jejíž osmý člen má např. hodnotu 2785999. Tato tabulka mimo jiné ukazuje, že hloubka rekurze a počet rekurzivních volání se mohou podstatně lišit.

Poznamenejme, že při výpočtu $A(4,1)$ nestačil zásobník a program skončil chybou.

# Kapitola 5

# Třídění

Pod pojmem *třídění* budeme rozumět uspořádání zadané posloupnosti dat podle určitého klíče v neklesajícím nebo nerostoucím pořadí.

Prvky posloupnosti, kterou třídíme, mohou být jakéhokoli typu, samozřejmě všechny stejného. Obecně půjde o záznamy (struktury); *klíčem*, podle kterého budeme prvky posloupnosti porovnávat, může být buď přímo některá ze složek tohoto záznamu nebo funkce, vypočítaná na základě celého záznamu. My budeme pro jednoduchost předpokládat, že každý z prvků posloupnosti obsahuje numerickou složku jménem *klíč*. Občas budeme pro stručnost mluvit pouze o porovnávání prvků; budeme tím ovšem mít na mysli porovnávání jejich klíčů. Podobně budeme-li hovořit o „nejmenším prvku“, budeme mít na mysli prvek s nejmenším klíčem atd.

Při výkladu metod třídění budeme rozlišovat dvě základní situace:

1. Jestliže známe předem počet prvků posloupnosti a všechny prvky tříděné posloupnosti jsou uloženy ve vnitřní paměti počítače, kde k nim můžeme přistupovat v libovolném pořadí (tedy třídíme data, uložená v poli), hovoříme o *vnitřním třídění*.
2. Jestliže počet prvků tříděné posloupnosti předem neznáme a prvky tříděné posloupnosti jsou uloženy ve vnější paměti se sekvenčním přístupem (tedy třídíme soubor na magnetické pásce nebo disku), hovoříme o *vnějším třídění*.

Metody vnitřního a vnějšího třídění se podstatným způsobem liší, proto o nich budeme hovořit zvlášť.

## 5.1 Vnitřní třídění

Jedním z nejdůležitějších požadavků při třídění polí je úsporné využívání operační paměti počítače. To znamená, že bychom měli pracovat „na místě“, přímo uvnitř tříděného pole, aniž bychom deklarovali pole pomocné. Jinými slovy výchozí (zdrojová) posloupnost musí být uložena na stejném místě paměti jako posloupnost cílová.

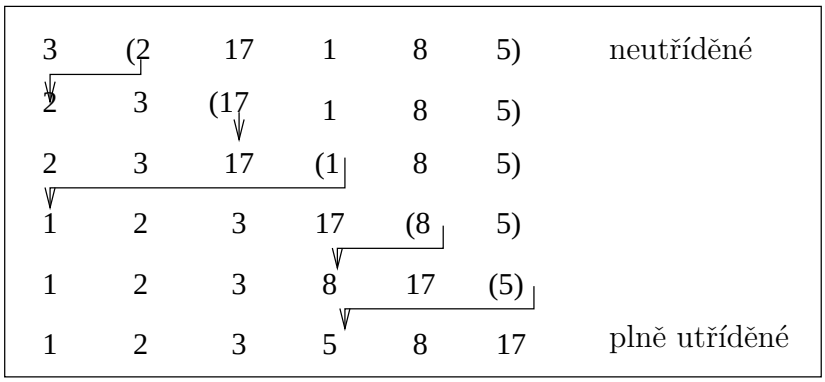
Má-li tříděné pole $n$ prvků, měli bychom tedy spotřebu paměti vyjádřit číslem $n + c$, kde $c$ je malá konstanta.

Dalším požadavkem je samozřejmě efektivita algoritmu. Vhodným měřítkem efektivity může být počet *porovnání klíčů* a *počet přesunů prvků* v poli. Obvykle je významnější počet přesunů, neboť bývají časově náročnější. Existují ale i příklady, ve kterých je porovnání složitější než přesun dat. Při našich úvahách si budeme všímat závislosti obou těchto veličin na počtu prvků $n$ tříděného pole.

V následujících odstavcích se seznámíme s několika jednoduchými metodami pro vnitřní třídění, které vyžadují řádově $n^2$ operací. Tyto algoritmy jsou velice jednoduché a vhodné pro malé objemy tříděných dat. Navíc se na nich snadno demonstrují principy třídění.

Kvalitní algoritmy pro třídění polí vyžadují řádově $n \log n$ porovnání, tedy podstatně méně. Jednotlivé operace mohou ale být složitější, takže pro malá n mohou být výhodnější metody jednoduché.

V následujících odstavcích budeme zpravidla předpokládat, že tříděná posloupnost je uložena v poli $a$, deklarovaném příkazem




Obr. 5.1: Třídění přímým vkládáním

```
var a: array [1..n] of data;
```

V některých případech toto pole doplníme zleva o několik prvků, které poslouží jako zarážky. Vedle toho budeme předpokládat deklaraci typu **index**:

```
type index = 1..n
```

### 5.1.1 Třídění přímým vkládáním

Tato metoda připomíná způsob, jakým si karetní hráči obvykle řadí karty v ruce: z balíčku rozdaných karet berou jednu kartu po druhé a zařadí ji na správné místo podle velikosti a barvy. Je-li třeba, odsunou doprava karty, které zařadili již dříve.

Při popisu třídění pole přímým vkládáním vyjdeme z představy dvou posloupností, zdrojové a cílové. V $i$-tém kroku vezmeme $i$-tý prvek zdrojové posloupnosti a zařadíme jej na správné místo v cílové posloupnosti.

Protože zdrojová i cílová posloupnost leží na témže místě paměti, budeme postupovat takto:

- První prvek pole ponecháme na místě.
- Vezmeme druhý prvek a porovnáme jej s prvním. Je-li větší, ponecháme jej na místě, jinak jej zařadíme na první místo a prvek z prvního místa odsuneme na druhé místo.
- Vezmeme třetí prvek, zjistíme, zda patří na první, druhé nebo třetí místo, zařadíme jej a prvky za ním podle potřeby odsuneme.

atd.

**Příklad 5.1**

Chceme utřídit posloupnost $(3, 2, 17, 1, 8, 5)$, viz obr. 5.1. Šipky na obrázku ukazují směry přesunů v jednotlivých krocích; v závorkách je dosud netříděná část pole.
Než se pustíme do zápisu algoritmu přímého vkládání v Pascalu, musíme si ujasnit, jak vložit prvek na správné místo v utříděné části posloupnosti.

Nejjednodušší možnost je asi tato: vkládaný prvek je na $i$-tém místě, vlevo od něj je již setříděná část pole. Porovnáme ho tedy s prvkem bezprostředně vlevo od něj. Pokud je vkládaný prvek větší nebo roven svému levému sousedu, je již na správném místě. V opačném případě tyto dva prvky prohodíme a vkládaný prvek opět porovnáme s prvkem, který je v poli bezprostředně vlevo od něj atd.

Podívejme se na vkládání prvku 1 v příkladu 5.1 (třetí řádek na obr. 5.1). Tento prvek stojí na 4. místě. Porovnáme jej tedy s prvkem na třetím místě (17); protože je menší, prohodíme je, takže dostaneme posloupnost $2, 3, 1, 17, \ldots$.

Dále porovnáme prvek 1 s prvkem na druhém místě (2). Protože 1 je menší, opět je prohodíme, takže dostaneme posloupnost $2, 1, 3, 17 \ldots$. Po dalším porovnání a prohození dostaneme posloupnost, kterou vidíme ve 4. řádku obrázku.

Hledání správného místa skončí, jestliže bude prvek vlevo menší než vkládaný prvek nebo jestliže jsme již dospěli na první místo v poli. To jsou dvě podmínky; jedné z nich se můžeme zbavit (a tak algoritmus poněkud zjednodušit), jestliže použijeme zarážku.

Pole $a$ bude místo $a\,[1]$ začínat prvkem $a\,[0]$, tj. deklarujeme je příkazem

```
var a: array [0..n] of prvek;
```

a do $a[0]$ uložíme hodnotu vkládaného prvku. Porovnávání pak skončí v nejhorším případě těsně před zarážkou, neboť vkládaný prvek bude mít stejný klíč jako zarážka.

Procedura pro třídění přímým vkládáním může vypadat takto ($n$ je počet prvků ve tříděné posloupnosti):

```
procedure PřímêVkládání;
var i,j: index;
    x:   prvek;                                  {pomocná proměnná}
begin
  for i := 2 to n do begin
     x := a[i];
     a[0] := x;                                  {definice zarážky}
     j := i-1;
     while x.klíč < a[j].klíč do begin
       a[j+1] := a[j];                  {odsouvání prvků doprava}
       dec(j)
     end;
     a[j+1] := x;                {vložení prvku na správné místo}
  end;
end;
```

**Analýza algoritmu přímého vkládání**

Při rozboru této metody se budeme opírat o proceduru *PřímêVkládání*. Je zřejmé, že nejhorší možný případ nastane, jestliže bude zdrojová posloupnost uspořádána v opačném pořadí - jako první bude největší prvek, jako poslední prvek nejmenší. Potom budeme muset v $i$-tém kroku, při vkládání $i$-tého prvku, provést $C_i = i-1$ porovnání a $M_i = i+1$ přesunů včetně uložení hodnoty vkládaného prvku do zarážky.

Celkový počet porovnání a přesunů v nejhorším případě potom bude

$$C_{max} = \sum_{i=2}^{n} C_i = \sum_{i=2}^{n} (i-1) = \frac{n^2-n}{2}, \qquad M_{max} = \sum_{i=2}^{n} M_i = \sum_{i=2}^{n} (i+1) = \frac{n^2+n-2}{2}. \tag{5.1}$$

Nejlepší možný případ nastane, když bude zdrojová posloupnost již správně uspořádána. V takovém případě potřebujeme v každém kroku pouze jedno porovnání, $C_i = 1$. Počet přesunů v popsané proceduře bude v každém kroku[1] $M_i = 3$. Celkový počet porovnání a přesunů v nejlepším případě potom bude

$$C_{min} = \sum_{i=2}^{n} 1 = n-1, \qquad M_{min} = 3\,(n-1)\,. \tag{5.2}$$

Budeme-li předpokládat, že všechny permutace zdrojové posloupnosti jsou stejně pravděpodobné, můžeme tvrdit, že průměrný počet porovnání bude $C_i = i/2$ a počet přesunů bude $M_i = C_i + 2$. Odtud pro průměrné počty operací odvodíme

$$C_{\oslash} = \sum_{i=2}^{n} C_i = \sum_{i=2}^{n} i = \frac{n^2+n-2}{4}, \qquad M_{\oslash} = \sum_{i=2}^{n} M_i = \sum_{i=2}^{n} \left(\frac{1}{2}i + 2\right) = \frac{n^2+9n-10}{4}. \tag{5.3}$$

Průměrný počet porovnání i přesunů je tedy $O\left(n^2\right)$.

[1] Algoritmus třídění přímým vkládáním lze naprogramovat tak, že tyto zbytečné přesuny odpadnou, výsledná procedura bude však složitější.

Poznamenejme, že metoda přímého vkládání je stabilní v tom smyslu, že prvky se stejnou hodnotou klíče budou v cílové posloupnosti uloženy ve stejném pořadí jako v posloupnosti zdrojové.

### 5.1.2 Třídění binárním vkládáním

Binární vkládání představuje jisté vylepšení přímého vkládání. Jestliže si uvědomíme, že zvolený prvek ukládáme do posloupnosti, která je již setříděná, můžeme zrychlit hledání místa, na které patří. Přitom nebudeme potřebovat zarážku.

Místo pro vkládaný prvek budeme zjišťovat metodou binárního vyhledávání, kterou můžeme považovat za typický příklad algoritmu vytvořeného podle modelu *rozděl a panuj*: cílovou posloupnost rozdělíme na dvě podposloupnosti a porovnáním zjistíme, do které z nich vkládaný prvek patří. Opakováním tohoto postupu nakonec určíme místo, na které prvek vložíme.

Přesně popíšeme tento algoritmus opět v Pascalu:

```
procedure BinárníVkládání;
var i,j,k,l,r,m: index;
    x:           prvek;
begin
  for i := 2 to n do begin
    x := a[i];
    l := 1; r := i-1;             {l,r jsou meze podposloupnosti}
    while l <= r do begin                       {binární hledání}
      m := (l+r) div 2;                    {rozdělíme na poloviny}
      if x.klíč < a[m].klíč then r := m-1
      else l := m+1
    end;{while}
    for j := i-1 downto l do a[j+1] := a[j];
    a[l] := x;
  end; {for i}
end;
```

#### Analýza algoritmu binárního vkládání

Místo, na které uložíme nový prvek, je popsáno podmínkou

```
a[j].klíč <= x.klíč <= a[j+1].klíč.
```

Zkoumaný interval délky $i$ přitom musíme rozdělit podle okolností buď $[\log_2 i]$-krát, kde $[z]$ označuje celou část čísla $z$, nebo $([\log_2 i] + 1)$-krát. To znamená, že celkový počet porovnání $C$ bude omezen součty

$$\sum_{i=1}^{n} [\log_2 i] \leq C \leq \sum_{i=1}^{n} ([\log_2 i] + 1) \tag{5.4}$$

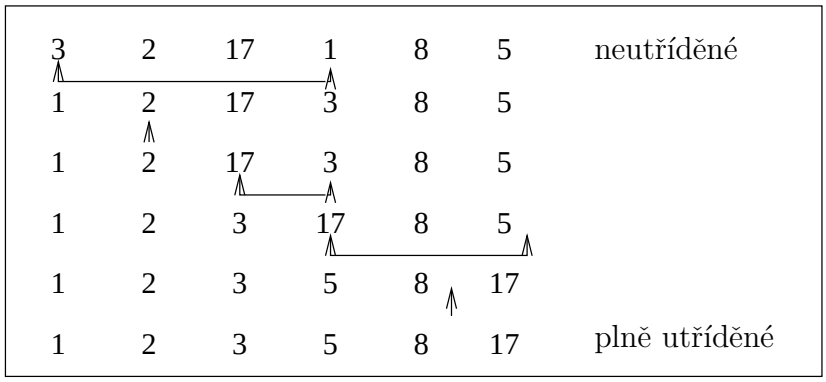
Hodnotu součtu na levé straně (5.4) můžeme přibližně odhadnout pomocí integrálu:

$$\sum_{i=1}^{n} [\log_2 i] \approx \int_1^n \log_2 x \, \mathrm{d}x = [x\,(\log_2 x - s)]_1^n = n\,(\log_2 n - s) + s, \tag{5.5}$$

(5)

kde $s = \log_2 e = 1/\ln 2 = 1,44269...$ .

Přitom počet porovnání bude prakticky nezávislý na uspořádání prvků zdrojové posloupnosti. Počet přesunů zůstane ovšem v podstatě stejný jako v případě přímého vkládání, odpadnou pouze operace se zarážkou. To znamená, že počet porovnání bude podle (5.5) jen $O\,(n \log_2 n)$, avšak počet přesunů zůstane $O\left(n^2\right)$. Vzhledem k tomu, že přesuny jsou zpravidla časově náročnější než porovnání, nemusí jít o úsporu nijak významnou.




Obr. 5.2: Třídění přímým výběrem

### 5.1.3 Třídění přímým výběrem

V metodě přímého i binárního vkládání jsme vždy brali jeden předem určený prvek zdrojové posloupnosti a ten jsme vkládali do cílové posloupnosti (tj. probírali jsme celou cílovou posloupnost).

Metoda přímého výběru je založena na opačném principu: ze zdrojové posloupnosti vybereme vhodný prvek (uvažujeme tedy vlastně celou zdrojovou posloupnost) a ten vložíme na předem přesně určené místo cílové posloupnosti.

Algoritmus přímého výběru bude založen na následující myšlence:

1. Najdeme ve zdrojové posloupnosti prvek s nejmenším klíčem.
2. Vyměníme ho s prvkem na první pozici.
3. Nyní je na první pozici nejmenší prvek z celé posloupnosti. Potřebujeme utřídit zbytek posloupnosti, proto opakujeme tyto kroky se zbylými $n-1$ prvky, dokud je $n > 1$.

**Příklad 5.2**

Na obrázku 5.2 vidíme postup třídění posloupnosti $(3, 2, 17, 1, 8, 5)$ metodou přímého výběru. Šipky naznačují výměny. Poznamenejme, že ve druhém kroku nedojde k výměně prvků, neboť nejmenší prvek ze zdrojové posloupnosti je právě na druhém místě; podobně nedojde k výměně v pátém kroku.
Procedura pro třídění přímým výběrem bývá obvykle uváděna ve tvaru

```
procedure PřímýVýběr;
var i,j,k: index;
    x:     prvek;
begin
    for i := 1 to n-1 do begin         {na první druhé,... místo}
      k := i;
      x := a[i];                                              {*}
      for j := i+1 to n do                        {najdi nejmenší}
        if a[j].klíč < x.klíč then begin
           k := j;
           x := a[j];                                        {**}
        end;
      a[k] := a[i];           {prohod' ho s prvkem na i-tém místě}
      a[i] := x;                                             {***}
    end;
end;
```

Poněkud výhodnější může být následující varianta téže procedury, ve které ušetříme několik přesunů prvků:

```
procedure PřímýVýběr1;
var i,j,k: index;
```

```
    x:     prvek;
begin
    for i := 1 to n-1 do begin         {na první druhé,... místo}
      k := i;
      for j := i+1 to n do               {najdi index nejmenšího}
        if a[j].klíč < a[k].klíč then begin
           k := j;
        end;
      x := a[k];        {prohod' nejmenší s prvkem na i-tém místě}
      a[k] := a[i];
      a[i] := x;
    end;
end;
```

**Analýza třídění přímým výběrem**

Pokud jde o počet porovnání klíčů, je tato metoda horší než přímý výběr. Počet porovnání totiž nezávisí na počátečním uspořádání prvků v poli, neboť v $i$-tém kroku vždy prohledáváme celé pole $a$ od pozice $i+1$ do konce. To znamená, že počet porovnání je

$$C = \frac{1}{2} n\,(n-1)\,.$$

Pokud jde o počet přesunů, záleží na tom, zda použijeme proceduru *PřímýVýběr* nebo *PřímýVýběr1*.

V případě procedury PřímýVýběr bude minimální počet přesunů

$$M_{min} = 3\,(n-1) \tag{5.6}$$

a to v případě, že pole je již uspořádané. Maximální počet přesunů nastane v případě, že pole je uspořádané v opačném pořadí. V tom případě se při každém průchodu vnějším cyklem provedou příkazy označené jednou a třemi hvězdičkami. Ty obsahují celkem 3 přesuny. Přiřazovací příkaz, označený dvěma hvězdičkami, se při $i = 1$ provede $(n-1)$-krát, při dalším průchodu $(n-2)$-krát, atd. (Při prvním průchodu se na poslední místo uložil největší prvek; ve druhém průchodu se začíná u druhého a skončí u předposledního prvku a přitom se na předposlední místo uloží druhý největší prvek atd.) Pro $i > n/2$ se již příkaz se dvěma hvězdičkami neprovádí, neboť pole je již srovnáno.

Při sčítání řady $n-1+n-2+\ldots$ musíme rozlišit několik případů podle hodnoty $n$. Dostaneme, že maximální počet přesunů bude

$$M_{max} = \left[\frac{n^2}{4}\right] + 3\,(n-1)\,, \tag{5.7}$$

kde $[z]$ opět znamená celou část čísla $z$. Při stanovení průměrného počtu přesunů se předpokládá, že všechny možné permutace prvků pole $a$ jsou stejně pravděpodobné. Lze ukázat, že průměrný počet přesunů bude přibližně roven $n\,(\ln n + \gamma)$, kde $\gamma$ je Eulerova konstanta, $\gamma = 0,577\ldots$ Viz [1], str. 101.

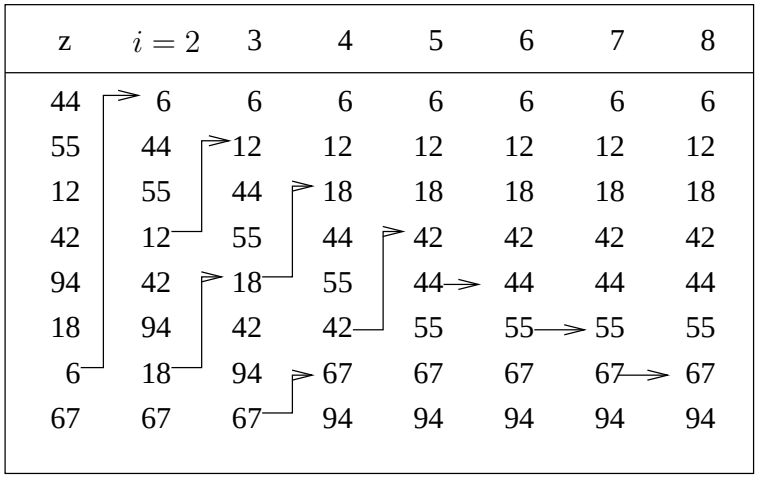
Použijeme-li proceduru *PřímýVýběr1*, bude třeba v každém kroku 3 přesuny bez ohledu na uspořádání prvků, tedy celkem $3\,(n-1)$ přesunů.

### 5.1.4 Bublinkové třídění a třídění přetřásáním

Tyto dvě metody bychom mohli shrnout pod označení „třídění přímou výměnou“, neboť výměny prvku jsou jejich dominantním rysem. Podobně jako většina metod vnitřního třídění jsou tyto dvě metody jsou založeny na opakovaném procházení polem. Přitom se vždy porovnávají dva sousední prvky, a pokud nejsou uloženy ve správném pořadí, prohodí se.

Algoritmus *bublinkového třídění* (*bubblesort*) v nejjednodušší podobně můžeme vyjádřit následující pascalskou procedurou:

```
procedure Bublání;
var i,j: index;
      x: prvek;
begin
```




Obr. 5.3: Bublinkové třídění

```
  for i := 2 to n do begin              {n-1 krát projdeme polem}
    for j := n downto i do                        {průchod polem}
      if a[j-1].klíč > a[j].klíč then begin
        x := a[j-1];                          {pokud prvky nejsou}
        a[j-1] := a[j];                      {ve správném pořadí,}
        a[j] := x;                                  {prohodíme je}
      end;
  end;
end;{bublání}
```

Jestliže pole procházíme „odzadu“, jako je tomu v proceduře *Bublání*, dostane se při prvním průchodu nejmenší prvek na první místo. Ve druhém průchodu už nemusíme procházet celé pole, stačí skončit na druhém místě, kam se při druhém průchodu dostane druhý nejmenší prvek; atd.

**Příklad 5.3**

Ukážeme si efekt procedury *Bublání* při třídění posloupnosti (44, 55, 12, 42, 9, 18, 6, 67). Na obrázku 5.3 jsme ji zapsali svisle ve sloupci nadepsaném **Z**. V dalších sloupcích vidíme tříděné pole při jednotlivých průchodech cyklem s parametrem $i$.

S trochou fantazie si můžeme představovat, že v prvním průchodu „vyplave“ nejlehčí prvek na první místo (jako bublinka - odtud název), ve druhém průchodu druhý nejlehčí prvek atd.
Na první pohled je zřejmé, že algoritmus, popsaný procedurou *Bublání*, lze vylepšit. Tato procedura totiž nerozpozná utříděné pole, takže např. poslední tři průchody při třídění posloupnosti, uvedené v příkladu 5.3, byly naprosto zbytečné. To můžeme napravit např. tak, že budeme při každém průchodu zaznamenávat počet výměn. Jestliže nedojde k žádné výměně, je třídění skončeno. Cyklus **for** s parametrem $i$ nahradíme cyklem **repeat** s podmínkou „v minulém průchodu došlo k alespoň jedné výměně“.

Dalšího vylepšení můžeme dosáhnout tím , že si zapamatujeme i index prvku, kterého se poslední výměna týkala. Za tímto prvkem již nebyly žádné další výměny, takže prvky zde jsou již uspořádány, a proto tento úsek již příště nemusíme procházet.

Vedle toho je zřejmá nesymetrie algoritmu bublinkového třídění: na obrázku 5.3 je vidět, že nejmenší prvek (zde 6) „vybublá“ na své místo ihned při prvním průchodu, i když byl původně na předposlední pozici, zatímco největší prvek (94) „klesá ke dnu“ velmi pomalu, při každém průchodu jen o jednu pozici.

Kdybychom procházeli tříděnou posloupnost v obráceném pořadí, tj. od prvního k poslednímu prvku, chovaly by se prvky obráceně. Největší prvek by se dostal na své místo ihned, zatímco nejmenší by postupoval pomalu. Odtud lze usuzovat, že bychom mohli dosáhnout dalšího vylepšení, kdybychom pravidelně střídali směry průchodu.

Tomuto vylepšení algoritmu bublinkového třídění se říká *třídění přetřásáním* (*shakesort*). jeho algoritmus můžeme v Pascalu zapsat ve tvaru následující procedury:

```
procedure Třesení;
var j,k,l,r : index;
      x:      prvek;
begin
  l := 2; r := n; k := n;
  repeat
    for j := r downto l do                    {průchod od konce}
      if a[j-1].klíč > a[j].klíč then begin
        x := a[j-1];                                  {výměna}
        a[j-1] := a[j]; a[j] := x;
        k := j;                           {index poslední výměny}
      end;
     l := k+1;                       {příště procházíme jen od l}
     for j := l to r do
      if a[j-1].klíč > a[j].klíč then begin
        x := a[j-1];                                  {výměna}
        a[j-1] := a[j]; a[j] := x;
        k := j;                           {index poslední výměny}
      end;
     r := k-1;                       {příště procházíme jen od r}
  until l > r;                    {když se l a r se setkají, konec}
end; {Třesení}
```

**Analýza bublinkového třídění a třídění přetřásáním**

Počet porovnání při bublinkovém třídění v nejjednodušší podobě nezávisí na uspořádání pole a je vždy roven

$$C = \sum_{i=1}^{n-1} (n-i) = \frac{1}{2}\left(n^2 - n\right). \tag{5.8}$$

Minimální počet přesunů je zřejmě $M_{min} = 0$, neboť pokud je pole již uspořádané, nebude nikdy splněna podmínka v příkazu **if** v proceduře *Bublání*.

Maximální počet přesunů dostaneme pro pole uspořádané obráceně. V tomto případě bude docházet k výměně prvků při každém porovnání v každém z průchodů; vezmeme-li v úvahu, že na jednu výměnu prvků potřebujeme 3 přesuny, dostaneme podobně jako ve vztahu (5.8)

$$M_{max} = \frac{3}{2}\left(n^2 - n\right). \tag{5.9}$$

Průměrný počet přesunů bychom pak mohli odhadnout hodnotou

$$M_{\oslash} = \frac{3}{4}\left(n^2 - n\right).$$

Podívejme se nyní na algoritmus třídění přetřásáním, popsaný procedurou *Třesení*. Je-li tříděné pole již uspořádané, bude mít proměnná $k$ po skončení prvního cyklu **for** hodnotu $n$, neboť podmínka v příkazu **if** nebude splněna ani jednou. Proměnná $l$ tím získá hodnotu $n+1$ a tělo druhého cyklu **for** se neprovede již ani jednou. Proměnná $r$ pak dostane hodnotu $n-1$, cyklus **repeat** skončí. Nejmenší počet porovnání proto bude $C_{min} = n - 1$.

Pokud jde o průměrný počet výměn, D. Knuth ukázal, že je roven

$$C_{\oslash} = \frac{1}{2}\left[n^2 - n\left(r + \ln n\right)\right],$$

kde $r$ je jistá konstanta [1]. Počet výměn zůstává u třídění přetřásáním stejný jako u bublinkového třídění. To znamená, že ve většině případů nebudou mít uvedená vylepšení metody bublinkového třídění velký význam.

### 5.1.5 Shellovo třídění (třídění se zmenšováním kroku)

Příčinou neefektivnosti některých z předchozích algoritmů, např. třídění přímým vkládáním nebo bublinkového třídění, je, že v nich vyměňujeme pouze sousední prvky. Pokud je například nejmenší prvek uložen na konci pole, potřebujeme $n$ výměn, aby se dostal na správné místo. Proto se některé algoritmy snaží preferovat výměny prvků „na velké vzdálenosti“. Jedním z takových algoritmů je i Shellovo třídění (*Shellsort*), navržené D. L. Shellem.

Základní myšlenkou Shellova třídění je přeuspořádat pole tak, že když vezmeme každý $h$-tý prvek, dostaneme setříděné pole (takovéto pole se nazývá *setříděné s krokem* $h$). Pole setříděné s krokem $h$ představuje vlastně $h$ proložených nezávislých polí.

Při třídění pole s krokem $h$ můžeme přesunovat prvky na větší vzdálenosti (nejméně $h$) a tak dosáhnout menšího počtu výměn při třídění s menším krokem. Při Shellově třídění tedy utřídíme pole s krokem $h_1$, pak s krokem $h_2 < h_1$ atd.; nakonec je utřídíme s krokem 1 a dostaneme plně utříděné pole. Při každém z následujících průchodů třídíme pole, které je již částečně setříděné, a proto můžeme očekávat, že bude třeba jen málo výměn.

Dosud jsme neurčili, jak utřídíme pole s krokem $h$. Použijeme např. metody přímého vkládání. V oddílu 5.1.1. jsme při formulaci algoritmu přímého vkládání použili zarážku, abychom zjednodušili podmínku pro ukončení průchodu polem. Budeme-li chtít použít zarážek i v Shellově třídění, musíme jich zavést více. Je-li $h_t$ největší z kroků, se kterými dané pole $a$ třídíme, musíme je deklarovat příkazem

```
var a: array [-ht..n] of prvek;
```

Pro větší hodnoty $h_t$ to nemusí být výhodné. V následující proceduře *Shell* zarážky nepoužijeme. Čtenář jistě dokáže napsat proceduru, využívající zarážek sám (viz též [1], str. 107). V této proceduře předpokládáme, že $t$ je globální konstanta, udávající počet různých kroků pro třídění.

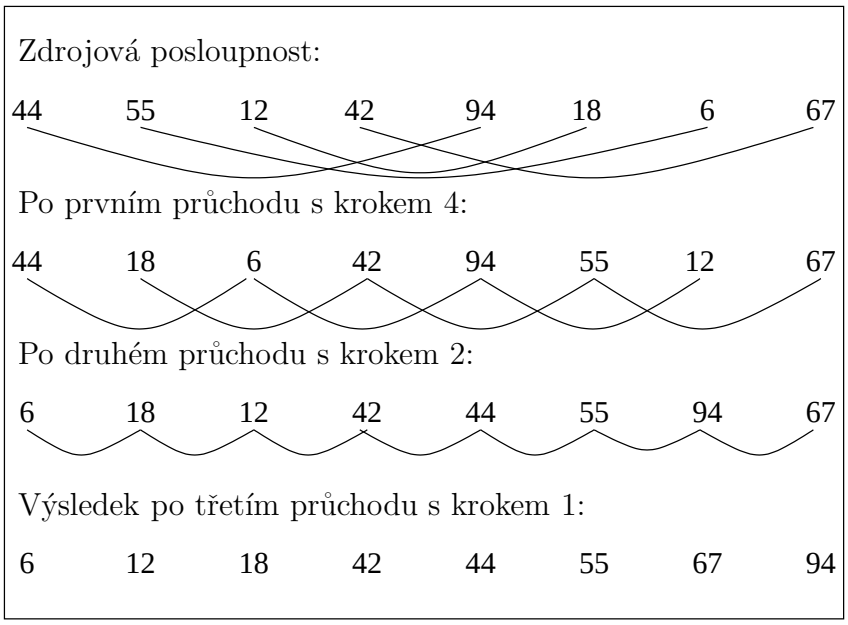
```
procedure Shell;
var i,j,k: index;
    x: prvek;
    m: 1..t;
    h: array [1..t] of integer;                {pole délek kroků}
begin
  h[t] := 1;                           {výpočet posloupnosti kroků}
  for i := t-1 downto 1 do h[i] := 3*h[i+1]+1;
  for m := 1 to t do begin{třídění s krokem h[m]}
     k := h[m];                                              {krok}
     for i := k+1 to n do begin                  {přímé vkládání}
        x := a[i];
        j := i-k;
        while (x.klíč < a[j].klíč) and (j >= k) do begin
           a[j+k] := a[j];                          {bez zarážky}
           j := j-k;
        end;
        a[j+k] := x;
     end;
  end;
end;
```

**Příklad 5.4**

Podívejme se opět na posloupnost $(44, 55, 12, 42, 9, 18, 6, 67)$. Zkusíme ji utřídit pomocí Shellova algoritmu; přitom zvolíme $t = 3$ a kroky $h_1 = 4$, $h_2 = 2$ a $h_3 = 1$. Jak se dále dozvíme, nejde o příliš výhodnou volbu; pro náš příklad ale postačí.

**Analýza Shellova třídění**

Analýza Shellova třídění vede ke komplikovaným matematickým problémům, které přesahují rámec tohoto textu. Uvedeme proto pouze některé výsledky; podrobnější informace najde čtenář např. v [5].




Obr. 5.4: Shellovo třídění (třídění se zmenšováním kroku)

První z problémů, o kterých se zmíníme, je volba posloupnosti kroků $h_i$. Otázka optimální volby není, jak se zdá, dosud uspokojivě vyřešena. Je však známo, že obecně horší výsledky dostaneme, jestliže bude $h_i$ násobkem $h_{i-1}$. Např. posloupnost $\ldots, 8, 4, 2, 1$ nevede k příliš efektivnímu třídění, neboť při této volbě porovnáme prvky na sudých místech s prvky na lichých místech až v posledním průchodu.

Na druhé straně je známo, že poměrně dobré výsledky dostaneme pro posloupnost kroků definovanou vztahy $h_t = 1$, $k_{k-1} = 3h_k + 1$; $t$ volíme podle vztahu $t = [\log_3 n] - 1$, kde $[z]$ opět označuje celou část čísla $z$. Poslední prvky této posloupnosti jsou $1, 4, 13, 40, 121, \ldots$.

Lze dokázat, že pro tuto volbu kroků nepřesáhne počet porovnání hodnotu $n^{3/2}$ [5].

Jiná doporučovaná posloupnost má tvar $h_t = 1$, $h_{k-1} = 2h_k + 1$; v tomto případě určíme $t$ ze vztahu $t = [\log_2 n] - 1$. Tato posloupnost končí čísly $1, 3, 7, 15, 31, \ldots$.

Celková složitost Shellova algoritmu v tomto případě je $O\left(n^{1,2}\right)$. [1]

### 5.1.6 Stromové třídění a třídění haldou

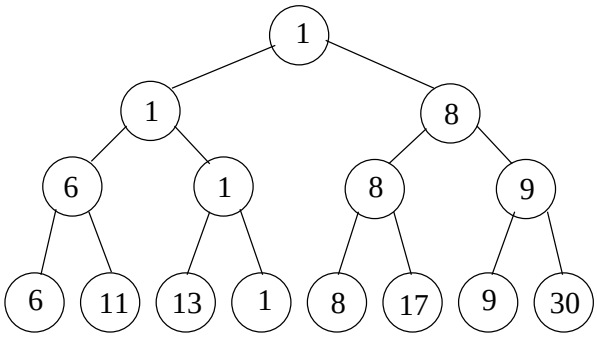
Předchozí metody byly založeny za opakovaném prohledávání pole - nejprve všech $n$ prvků, potom $n-1$ prvků atd. Pokud se nám nepodaří snížit počet prohledávání, bude počet porovnání vždy řádu $O\left(n^2\right)$. Přitom by se mohlo zdát, že stačí pouhé jedno prohledání pole, neboť při něm už získáme všechny potřebné informace. Je zřejmé, že všechny předchozí metody informací příliš optimálně nevyužívaly.

Pokusíme se tedy navrhnout metodu, která bude využívat informací poněkud efektivněji.

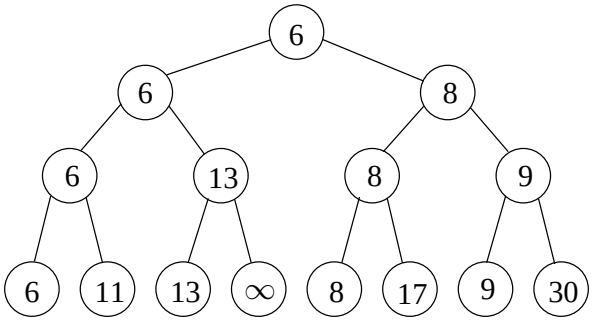
Rozdělíme-li prvky v poli na dvojice, můžeme pomocí $n/2$ porovnání určit menší prvek z každé dvojice. Pomocí dalších $n/4$ porovnání určíme menší prvek z každé čtveřice (stačí porovnat menší prvky z každé dvojice) atd. Takto můžeme pomocí $n-1$ porovnání sestrojit porovnávací strom, jehož kořen obsahuje nejmenší prvek celého pole a kořen každého podstromu obsahuje nejmenší prvek tohoto podstromu.

**Příklad 5.5**

Provnávací strom, zkonstruovaný z posloupnosti $(6, 11, 13, 1, 8, 17, 9, 30)$ vidíme na obr. 5.5.
Nyní z celého stromu odstraníme nejmenší prvek. List, který tuto hodnotu obsahoval, nahradíme vrcholem s klíčem $+\infty$, a ostatní vrcholy porovnávacího stromu, které obsahovaly nejmenší hodnotu, nahradíme vždy




Obr. 5.5: Porovnávací strom




Obr. 5.6: Porovnávací strom z obr. 5.5 po odstranění nejmenšího prvku

druhým z dvojice vrcholů, ve které se tato hodnota vyskytovala. Nyní můžeme odstranit druhý nejmenší prvek atd. Odstraněné hodnoty tvoří posloupnost seřazenou podle velikosti. *Stromové třídění* skončí v okamžiku, kdy vrcholu přiřadíme hodnotu $+\infty$.

**Příklad 5.5 (pokračování)**

V porovnávacím stromu na obr. 5.5 je nejmenší hodnota 1. List, který ji obsahoval, nahradíme listem s hodnotou $+\infty$. Předchůdce tohoto listu bude obsahovat hodnotu 13, předchůdce na úrovni 2 bude obsahovat hodnotu 6 a kořen pak taky 6. Strom, který vznikne touto úpravou, vidíme na obr. 5.6. V následujícím kroku odstraníme hodnotu 6 a do kořene se dostane 8. Další kroky jistě zvládne čtenář sám.
Na sestrojení stromu jsme potřebovali $n$ kroků; na jednotlivé výběry potřebujeme $\log_2 n$ kroků (počet těchto kroků je roven počtu úrovní stromu). Protože výběrů ze stromu je celkem $n$, dostáváme, že stromové třídění potřebuje $O(n \log_2 n)$ kroků. Pro velká $n$ tedy bude výhodnější než Shellovo třídění, které vyžaduje $O\left(n^{1,2}\right)$ kroků.

Podívejme se nyní na nevýhody stromového třídění. Tento algoritmus potřebuje celkem $2n - 1$ paměťových míst, zatímco všechny předchozí metody potřebovaly pouze $n$ míst. Navíc v závěru stromového třídění se strom zaplní vrcholy s hodnotou $+\infty$, které se naprosto zbytečně porovnávají.

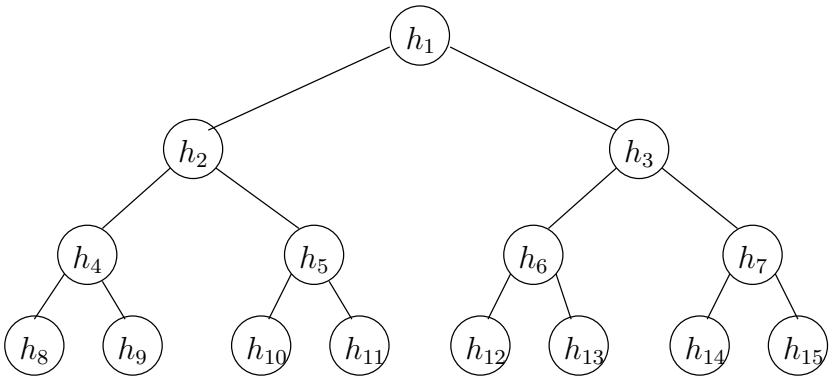
Obě tyto nevýhody se pokouší odstranit metoda *třídění haldou* (heapsort), kterou navrhl J. Williams [23].

*Halda* je zde vlastně binární strom, v němž jsou uložená data jistým způsobem uspořádána a který je uložen v poli. Formální definice haldy zní:

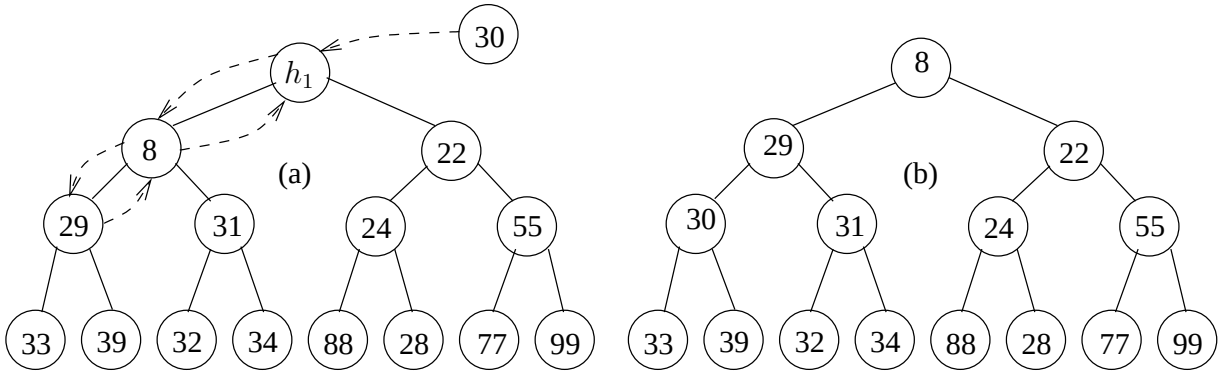
Halda je posloupnost prvků $h_l, h_{l+1}, \ldots, h_r$ takových, že pro všechny indexy $i = l, \ldots, r/2$ platí nerovnosti

$$h_i \leq h_{2i} \quad , \quad h_i \leq h_{2i+1} \tag{5.10}$$

Uvažujme haldu $h_1, h_2, \ldots, h_n$. Pak je $h_1$ kořenem stromu a $h_2$ a $h_3$ jsou jeho levý a pravý následovník. Vrchol $h_2$ má následovníky $h_4$ a $h_5$, vrchol $h_3$ má následovníky $h_6$ a $h_7$. Viz též obr. 5.7
Vezměme nyní prvky $a_l, \ldots, a_n$ tříděného pole $a$. Je-li $l \geq n/2$, tvoří podle definice haldu, neboť ve tříděném poli pro žádný index $i = n/2 \ldots n$ neexistuje $a_{2i}$ nebo $a_{2i+1}$. Tyto prvky představují vlastně nejnižší úroveň binárního stromu. Abychom toho mohli využít, potřebujeme ukázat, jak přidat do haldy nový prvek.




Obr. 5.7: Halda z patnácti prvků, uložených v poli $h_1, h_2, \ldots, h_{15}$




Obr. 5.8: (a) Přidávání prvku s hodnotou 30 do haldy; (b) výsledek

Přidejme k haldě $a_{l+j}, \ldots, a_n$ prvek $a_l$. Nově přidaný prvek v posloupnosti $a_l, \ldots, a_n$ je na místě kořene (nemusí jít o kořen celého stromu, ale nějakého podstromu). Podíváme se, zda pro něj platí podmínky (5.10). Pokud ano, je vše v pořádku a posloupnost $a_l, \ldots, a_n$ stále tvoří haldu. Pokud ne, musíme tento prvek prohodit s menším z jeho následovníků tak, aby byly podmínky (5.10) splněny. Nyní opět zkontrolujeme, zda platí podmínky (5.10); pokud ne, musíme nově vložený prvek opět zaměnit s některým z následovníků atd.

Nově vložený prvek tedy posunujeme z původní polohy v kořeni stromu na nižší úrovně tak, aby platily podmínky (5.10), které definují haldu.

**Příklad 5.6**

Uvažujme haldu z obr. 5.8 (a), kterou tvoří vlastně dva samostatné podstromy. Do této haldy chceme přidat prvek s hodnotou 30. Nejprve jej vložíme na pozici $h_1$, takže se stane kořenem stromu. Protože ale nesplňuje podmínky (5.10), musíme jej prohodit s menším z jeho následovníků , který má hodnotu 8. Ani zde nejsou podmínky (5.10) splněny, proto jej opět prohodíme s menším z následovníků (má hodnotu 29). Výsledek vidíme na obr. 5.8 (b).

Jakmile umíme přidat do haldy nový prvek, umíme vytvořit haldu z prvků celého pole:

1. Vyjdeme od prvků $a_l, \ldots, a_n$ tříděného pole $a$ pro $l = [n/2] + 1$; tyto prvky tvoří haldu.
2. K části haldy, která je již hotová, postupně přidáme prvky $l-1, l-2, \ldots, 1$.

Proceduru, která přidá jeden prvek do haldy, nazveme výstižně *VytvořHaldu*, neboť s její pomocí haldu opravdu vytvoříme.

```
procedure VytvořHaldu(l,r: integer);
label 13;
var i,j: index;
    x:   prvek;
```

```
begin
  i := l;
  j := 2*i;
  x := a[i];                                    {přidávaný prvek}
  while j <= r do begin                 {V tomto cyklu se x zařadí}
    if j < r then                             {Porovnat s menším z}
       if a[j].klíč > a[j+1].klíč then inc(j);     {následovníků}
     if x.klíč <= a[j].klíč then goto 13;    {x je větší - konec}
     a[i] := a[j]; i := j; j := 2*i;
  end;
  13: a[i] := x;
end;{vytvoř haldu}
```

Haldu vytvoříme opakovaným voláním této procedury pro $l = [n/2], [n/2]-1, \ldots, 1$.

Ve vytvořené haldě bude na prvním místě nejmenší prvek ze tříděného pole; další prvky však již seřazené nejsou. Proto budeme celý postup opakovat pro prvky $a_2, \ldots, a_n$: vytvoříme z nich haldu, která bude mít na prvním místě (tj. ve druhé pozici tříděného pole) nejmenší hodnotu zbylé části posloupnosti. V dalším kroku pak vytvoříme haldu z prvků $a_3, \ldots, a_n$ atd.

Zde ovšem vytváříme haldu z $n$ prvků, pak haldu z $n-1$ prvků atd. Výhodnější bude, jestliže po prvním kroku, po vytvoření haldy z $n$ prvků, uklidíme nalezený nejmenší prvek na konec pole, tj. vyměníme první prvek s posledním. Snadno se přesvědčíme, že pokud prvky $a_1, \ldots, a_n$ tvořily haldu, tvoří ji i prvky $a_2, \ldots, a_{n-1}$. Proto nyní bude stačit přidat k této haldě an a dostaneme haldu tvořenou $a_2, \ldots, a_n$. Druhý nejmenší prvek pole, který bude v kořeni této haldy, opět „uklidíme na konec“, vyměníme ho s předposledním prvkem pole atd.

**Příklad 5.7**

**Uvažujme posloupnost**

$$(3, 5, 8, 2, 1, 4, 7, 6)\,.$$

Základem haldy budou prvky s indexy 5 - 8. k nim postupně přidáme prvky s indexy 4, 3, 2 a 1 a vytvoříme tak haldu

$$(1, 2, 4, 3, 5, 8, 7, 6)\,.$$

Nyní „uklidíme“ nalezený nejmenší prvek na konec pole, tj. prohodíme ho s posledním prvkem. Tak dostaneme posloupnost

$$(6, 2, 4, 3, 5, 8, 7, 1)\,.$$

Protože prvky $2, 4, 3, 5, 8, 7$ tvoří haldu, stačí k ní přidat prvek 6. (Nyní vytváříme samozřejmě haldu pouze z prvních $n-1$ prvků.). Po zařazení prvku 6 do haldy pomocí procedury *VytvořHaldu* dostaneme posloupnost

$$(2, 3, 4, 6, 5, 8, 7, 1)\,.$$

Ve zbylé posloupnosti je tedy nejmenší hodnota 2. To opět „uklidíme“ na konec, vyměníme ji za předposlední prvek a dostaneme

$$(7, 3, 4, 6, 5, 8, 2, 1)\,.$$

Je tedy třeba přidat k haldě tvořené prvky $a\,[2] \ldots a\,[6]$ prvek 7... atd. Výsledkem bude pole, seřazené obráceně:

$$(8, 7, 6, 5, 4, 3, 2, 1)\,.$$

Popsaný postup vede k poli, seřazeném v obráceném pořadí. Pokud si přejeme pole, seřazené od nejmenšího prvku k největšímu (jako ve všech předchozích metodách), stačí prostě obrátit nerovnosti v proceduře *VytvořHaldu* (tedy vlastně obrátit nerovnosti v definici haldy ve vztazích (5.10)).

Vyložený postup zachycuje procedura TříděníHaldou. Jejím jádrem je procedura *VytvořHaldu* se změněnými nerovnostmi.

```
procedure VytvořHaldu(l,r: integer);
label 13;
var i,j: index;
    x:   prvek;
begin
```

```
  i := l;
  j := 2*i;
  x := a[i];
  while j <= r do begin                          {Zařadí x na místo}
    if j < r then                               {Porovnání s větším}
     if a[j].klíč < a[j+1].klíč then inc(j);              {prvkem}
    if x.klíč >= a[j].klíč then goto 13;    {Jsou-li násl. menší}
    a[i] := a[j]; i := j; j := 2*i;                    {tak konec}
  end;
  13: a[i] := x;

end;{vytvoř haldu}
procedure TříděníHaldou;
var l, r: index;
    x:    prvek;
begin
  l := (n div 2) + 1;                         {Prvky s indexy l..r}
  r := n;                                        {již tvoří haldu}
  while l > 1 do begin
    dec(l);                                {Přidávání prvků k haldě}
    VytvořHaldu(l,r)
  end;
  while r > 1 do begin                  {Nalezený nejmenší prvek dej}
     x := a[1];                                    {na konec pole}
     a[1] := a[r];
     a[r] := x;                                 {a ze zbytku zase}
     dec(r);
     VytvořHaldu(1,r)                               {vytvoř haldu}
  end;
end;{třídění haldou}
```

**Analýza třídění haldou**

Proceduru *VytvořHaldu* budeme volat při počátečním vytvoření haldy $n/2$-krát. Počet míst, přes které se budou při jednotlivých voláních přemísťovat vkládané prvky, bude postupně $[\log_2(n/2)], [\log_2(1+n/2)], \ldots, [\log_2(n-1)]$. Celkem jde tedy o méně než $[(n/2)\log_2 n]$ přesunů při konstrukci haldy.

Vlastní třídění (cyklus **while** $r > 1$ v proceduře *TříděníHaldou*) vyžaduje $n-1$ průchodů. Jednotlivá volání procedury *VytvořHaldu* v tomto cyklu vyžadují nejvýše $[\log_2(n-1)], [\log_2(n-2)], \ldots, 1$ přesunů; celkem půjde přibližně o $(n-1)(\log_2(n-1)-s)$ přesunů, kde $s = \log_2 e = 1,44269\ldots$ (viz vztah (5.5) v oddílu o třídění binárním vkládáním). Vedle toho potřebujeme při každém průchodu tímto cyklem 3 přesuny na přemístění nalezeného prvku na konec pole, tedy celkem dalších $3(n-1)$ přesunů.

Z toho plyne, že v celkový počet přesunů je $O(n \log_2 n)$. To je výsledek lepší než u Shellova algoritmu.

Na druhé straně je první pohled je zřejmé, že jednotlivé kroky třídění haldou jsou složitější než tomu bylo u předchozích metod. Např. největší prvek se po vytvoření haldy dostane na první místo a teprve pak jej odsuneme na správné místo na konci pole. Z toho plyne, že třídění haldou bude výhodné zejména pro rozsáhlá pole, kde se výrazně uplatní malý počet operací.

### 5.1.7 Rychlé třídění (quicksort)

Nejčastější označení, pod kterým se s tímto algoritmem setkáme, je *quicksort*. Vedle českého překladu *rychlé třídění* se také setkáme s označením *třídění rozdělováním*. Algoritmus pro rychlé vnitřní třídění publikoval Hoare [24] a jak se bystrý čtenář podle názvu jistě dovtípil, je (alespoň ve většině případů) velmi rychlý.

Myšlenka tohoto 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, můžeme je utřídit po-

mocí pouhých $n/2$ výměn. Porovnáme prvky na obou koncích pole a zaměníme je; pak postoupíme o jedno pole směrem ke středu a zopakujeme porovnání.

Pokud pole není seřazené v obráceném pořadí, nemůže být náš postup tak přímočarý. Vyjdeme z principu *rozděl a panuj*, o kterém jsme hovořili v kap. 3.1.

Zvolíme náhodně prvek $x$ a uspořádáme pole $a$ tak, aby vlevo od $x$ ležely prvky s klíči menšími než je klíč $x$ a vpravo prvky s klíči většími než je klíč $x$. Tím jsme pole rozdělili na tři části. Prvek $x$ je již na svém místě, vlevo od něj jsou prvky menší a vpravo od něj jsou prvky větší.

Pokud pole obsahuje pouze 3 prvky a za $x$ jsme zvolíme prostřední (prostřední pokud jde o velikost, tedy medián), získáme tímto procesem uspořádané pole. Pokud má pole délku 2, můžeme za $x$ zvolit kterýkoli z nich - výsledkem bude opět uspořádané pole.

Odtud je již zřejmý algoritmus rychlého třídění:

- Zvolíme $x$ a rozdělíme popsaným způsobem tříděné pole na úseky $a[1], \ldots, a[s], \ldots, a[n]$. Přitom platí, že $a[s] = x$, prvky $a[1], \ldots, a[s-1]$ jsou menší nebo rovny $x$ a prvky $a[s+1], \ldots, a[n]$ jsou větší nebo rovny $x$.
- Proces rozdělení opakujeme pro úseky $a[1], \ldots, a[s]$ a $a[s+1], \ldots a[n]$ a dále pak pro jejich části, až dospějeme k úsekům délky 1. Ty jsou již automaticky utříděné.

### Rozdělení pole na dvě části

Podívejme se nyní podrobněji na proces rozdělení pole na dvě části.

Z toho, co jsme si řekli v předchozím oddílu, plyne: vyjdeme od prvního prvku pole $a$ a budeme hledat prvek, který je větší než $x$ (má větší klíč). Zároveň budeme pole $a$ prohledávat od konce, až narazíme na prvek, který je menší než $x$. Pak tyto dva prvky zaměníme. Až se oba směry prohledávání setkají uprostřed pole, skončíme.

Vzhledem k tomu, že se s takovýmto dělením pole na dvě části setkáme také v následujícím oddílu, věnovaném hledání mediánu, formulujeme je jako samostatnou proceduru:

```
type pole = array [1..n] of prvek;
procedure Rozděl(var a: pole, x: prvek);
var w:   prvek;
    i,j: index;
begin
  i := 1; j := n;
  repeat                          {Tyto dva cykly while hledají}
    while a[i].klíč < x.klíč do inc(i);         {prvky, které se}
    while x.klíč < a[j].klíč do dec(j);         {navzájem vymění}
    if i <= j then begin
      w := a[i]; a[i] := a[j]; a[j] := w;               {Výměna}
      inc(i); dec(j);
    end;
  until i > j;                    {Až do setkání uprostřed pole}
end;
```

### Příklad 5.8

Uvažujme posloupnost

$$(17, 12, 15, 11, 20, 13).$$

Za rozdělovací prvek $x$ zvolíme hodnotu 15. První cyklus **while** najde prvek 17, který je větší než $x$ a leží vlevo od $x$. Druhý cyklus **while** najde prvek 13, který je menší než $x$ a leží vpravo od $x$. Proto se tyto dva prvky zamění. Tak dostaneme posloupnost

$$(13, 12, 15, 11, 20, 17).$$

Dále první cyklus **while** skončí u hodnoty 15 (tedy u prvku $x$) a druhý cyklus **while** u hodnoty 11. Po výměně těchto prvků bude mít naše posloupnost tvar

$$(13, 12, 11, 15, 20, 17)\,.$$

Tím procedura rozdělování skončí. Všimněte si, že se přemístila i zarážka $x$.

Poznamenejme, že v algoritmu pro rychlé třídění použijeme lehce upravenou verzi této procedury.

### Vlastní algoritmus rychlého třídění

Vlastní implementace algoritmu quicksort bude již velice jednoduchá. Základem bude rekurzivní (nebo iterativní) volání lehce upravené rozdělovací procedury.

V následující proceduře budeme pro změnu předpokládat, že pole třídíme na základě hodnot funkce *Pořadí(X: prvek):integer*. Pokud bychom chtěli třídit podle klíčů, stačí výrazy tvaru $Poad\,(a\,[i])$ nahradit výrazy tvaru $a\,[i]\,.kli$.

Procedura *Třídění*, vnořená do procedury Quicksort, je vlastně stará známá procedura *Rozděl*, která má jako vstupní parametry počáteční a koncový index tříděného úseku pole. Jako $x$, prvek, podle kterého se pole rozděluje, se volí vždy prostřední prvek. Na konec této procedury jsme doplnili rekurzivní volání: jestliže po rozdělení vzniknou úseky delší než 1, zavolá se na ně opět procedura *Třídění*.

Vlastní tělo procedury *Quicksort* obsahuje pouze volání procedury *Třídění* pro celé pole, tedy pro úsek od indexu 1 do $n$.

```
procedure Quicksort;                              {Rekurzivní verze}
      {procedura Třídění rozděluje pole na úseky a ty uspořádává}
                               {třídí úsek pole a od a[l] do a[r]}
  procedure Třídění(l,r: index);
  var w,x: prvek;
      i,j: index;
  begin {Třídění}
      i := l; j := r;
      x := a[(l+r) div 2];              {Zarážka: prvek uprostřed}
      repeat
        while Pořadí(a[i]) < Pořadí(x) do inc(i);
                                        {Indexy prvků pro výměnu}
        while Pořadí(a[j]) > Pořadí(x) do dec(j);
        if i <= j then begin                                {výměna}
           w := a[i]; a[i] := a[j]; a[j] := w;
           inc(i); dec(j);
        end;
      until i>j;
      if l < j then Třídění (l,j);         {prvek x na svém místě}
      if i < r then Třídění (i,r);              {třídí se ostatní}
  end; {třídění}
begin {Quicksort}
    Třídění(1,n);
end;{Quicksort}
```

Použití rekurze v algoritmu rychlého třídění je přirozeným důsledkem použití principu *rozděl a panuj*. Ve skutečnosti ale nemusí být rekurzivní tvar procedury *Quicksort* výhodný. Můžeme použít např. postupu, se kterým jsme se seznámili v kap. 4.2., a převést *Quicksort* do nerekurzivní podoby. Často je ovšem výhodnější vyjít z rozboru algoritmu a pokusit se dospět k nerekurzivní verzi přímo.

V našem případě je zřejmé, že při každém průchodu, tj. při každém volání procedury *Třídění*, se pole rozdělí na dva úseky. Jeden z nich můžeme ihned zpracovat, meze druhého si musíme uschovat. K tomu použijeme zásobník.

Na počátku vložíme do zásobníku meze celého pole, tj. 1 a $n$. Procedura *QuicksortN* (nerekurzivní quicksort) si tyto hodnoty ze zásobníku vezme, úsek pole rozdělí, jednu dvojici mezí uloží do zásobníku a druhou ihned zpracuje.

Zpracování bude probíhat v cyklu **repeat**, který skončí, jakmile nebude v zásobníku žádná další dvojice mezí ke zpracování.

Zásobník bude pole, složené ze struktur, obsahujících dvojice indexů. Definujeme jej jako objektový typ:

```
type  zásobník = object
       z: array [1..m] of record          {Pole na ukládání dat}
          l,r: index end;
       s: 0..m;                    {Ukazatel na vrchol zásobníku}
       constructor Vytvoř;
       procedure Vlož(i,j: index);
       procedure Vyjmi(var i,j: index);
       function Prázdný: boolean;
      end;
```

Konstruktor *Vytvoř* uloží do atributu **s** hodnotu 0 (zásobník je prázdný). Metody *Vlož* resp. *Vyjmi* umožňují vložit do zásobníku dvojici indexů resp. vyjmout ji ze zásobníku. Booleovská funkce *Prázdný* vrátí **true**, bude-li zásobník prázdný, tj. bude-li $s = 0$. Implementace těchto metod je triviální a přenechávám ji čtenáři.

V proceduře deklarujeme zásobník jako lokální proměnnou z typu *zásobník*. Při třídění se budeme - podobně jako v rekurzivní verzi - opírat o funkci *Pořadí*.

```
procedure QuicksortN;                          {Nerekurzivní verze}
var i,j,l,r: index;
    x, w:    prvek;
    z:       zásobník;
begin
 z.Vytvoř;
 z.Vlož(1,n);                             {Inicializace zásobníku}
 repeat                             {Další úsek z vrcholu zásobníku}
   z.Vyjmi(l,r);
   repeat                              {Rozdělení úseku a[l]..a[r]}
     i := l; j := r;
     x := a[(l+r) div 2];                                 {Zarážka}
     repeat
       while Pořadí(a[i]) < x do inc(i); {Indexy prvků pro výměnu}
       while Pořadí(a[j]) > x do dec(j);
       if i <= j then begin                                {Výměna}
          w := a[i]; a[i] := a[j]; a[j] := w;
          inc(i); dec(j);
       end;
     until i>j;
     if i < r then z.Vlož(i,r);    {Ulož pravý úsek do zásobníku}
     r := j;
   until l >= r;
 until z.Prázdný;
end;
```

V implementaci nerekurzivní verze algoritmu rychlého třídění zůstal jeden nevyřešený problém: jak velké má být $m$, konstanta, která určuje velikost zásobníku? Odpověď najdeme při analýze algoritmu rychlého třídění.

### Analýza rychlého třídění

Vyjdeme od analýzy procesu rozdělování pole na úseky. Má-li tříděné pole $n$ prvků, potřebujeme k rozdělení pole nejvýše $n$ porovnání v cyklech **while**, kde porovnáme všechny prvky se zvolenou zarážkou $x$.

Dále určíme střední počet výměn. Bez újmy na obecnosti můžeme předpokládat, že tříděné pole obsahuje klíče $1, \ldots, n$. Všechny permutace klíčů, tedy všechna možná uspořádání prvků v poli, pokládáme za stejně pravděpodobné. Jako zarážku $x$ jsme zvolili prvek s klíčem $i$. Potom bude potřebný počet výměn roven součinu počtu prvků v levém úseku pole, $i-1$, a pravděpodobnosti výskytu klíče, který je větší než $i$ (a tedy jej musíme vyměnit). Tato pravděpodobnost je rovna počtu takových klíčů, lomenému $n$.

Střední počet výměn při dělení pole na úseky potom bude

$$M = \frac{1}{n} \sum_{i=1}^{n} (i-1) \frac{n-i+1}{n} = \frac{n}{6} - \frac{1}{6n} \approx \frac{n}{6} \tag{5.11}$$

Kdyby se nám podařilo při volbě rozdělovacího prvku $x$ zvolit vždy medián, tj. kdyby bylo výsledkem vždy rozdělení pole na poloviny, potřebovali bychom k utřídění pole $\log_2 n$ průchodů. To znamená, že v tomto (nejvýhodnějším) případě potřebujeme celkem $C = n \log_2 n$ porovnání a $M = (n/6) \log_2 n$ výměn, tedy $(n/2) \log_2 n$ přesunů, neboť jedna výměna prvků vyžaduje tři přesuny.

Lze ukázat, že pokud budeme rozdělovací prvek volit náhodně (rovnoměrně), bude průměrná účinnost algoritmu rychlého třídění horší o faktor $2 \ln 2$ [1],[26].

Podívejme se ještě na nejhorší případ. Jeho analýza je zajímavá nejen z hlediska účinnosti algoritmu, ale i z hlediska velikosti zásobníku (konstanty $m$) v nerekurzivní verzi.

Zřejmě nejhorší možný případ nastane, jestliže jako zarážku zvolíme nejmenší prvek z daného úseku. Potom bude mít pravý úsek délku $n-1$ prvků a levý délku 1. Pokud se nám takováto nešťastná volba podaří v každém kroku, budeme potřebovat místo $\log_2 n$ celkem $n$ rozdělení. V tomto případě bude celkový počet porovnání i přesunů $O\left(n^2\right)$ a rychlé třídění již nebude dělat čest svému jménu.

Zmíněný nejhorší případ nastane mj. tehdy, když bude pole již předem utříděné a my budeme volit jako $x$ vždy první nebo poslední prvek. Pokud bychom volili vždy prostřední prvek, představovalo by utříděné pole naopak nejlepší případ; čtenář ovšem jistě snadno zkonstruuje pole, které povede k nejhoršímu případu při volbě prostředního prvku.

Obvykle se doporučuje buď volit $x$ náhodně nebo jako medián malého vzorku (3 - 5 prvků). Taková volba sice nezmění průměrný výkon algoritmu, zlepší ale jeho chování v nejhorším případě.

Podívejme se nyní na nerekurzivní verzi algoritmu. Délka zásobníku musí vycházet z nejhoršího možného případu. V proceduře *QuicksortN* vždy zpracováváme levý úsek a meze pravého úseku uložíme do zásobníku, kde čekají na budoucí zpracování. Jestliže bude mít pravý úsek vždy délku 1 a levý $n-1$, nahromadí se nám v zásobníku až $n$ dvojic mezí. To znamená, že v nejhorším případě může být potřeba pomocné paměti $O(n)$!

Přitom pro rekurzivní proceduru nebude situace o nic lepší - naopak, spotřeba paměti bude ještě vyšší. Jak víme, budou se parametry rekurzivních volání ukládat na implicitní zásobník programu; vedle toho ovšem přibudou ještě lokální proměnné a návratové adresy.

Vraťme se ale k nerekurzivní verzi. Snadno ukážeme, že když budeme vždy zpracovávat jako první kratší úsek a delší odložíme na později, bude maximální délka zásobníku rovna $m = \log_2 n$ (to nastane, jestliže budeme za $x$ volit vždy medián). Odpovídající úpravu procedury *QuicksortN* zvládne čtenář jistě sám.

## 5.2 Hledání k-tého prvku podle velikosti

Tato úloha úzce souvisí s algoritmy pro vnitřní třídění. Máme pole $a$, mezi jehož prvky chceme najít $k$-tý podle velikosti. Nejčastěji půjde o hledání *mediánu*, tedy prvku, který je menší nebo roven jedné polovině prvků pole $a$ zároveň je větší nebo roven druhé polovině prvků pole. Např. mediánem posloupnosti (11, 23, 6, 19, 1) je číslo 11.

Občas ale potřebujeme najít obecně $k$-tý prvek podle velikosti, tedy prvek $x$, pro který platí: $k-1$ prvků pole je menších nebo rovno $x$ a $n-k$ prvků je větších nebo rovno $x$. Připomeňme si, že pokud říkáme, že prvek $x$ je větší než prvek $y$, máme tím na mysli opět nerovnost mezi jejich klíči.

První možnost, která nás napadne, je jednoduchá:

1. Pole $a$ utřídíme podle velikosti od nejmenšího prvku k největšímu.
2. Vezmeme prvek $a[k]$.

Ukážeme si ale, že existuje efektivnější metoda.

### 5.2.1 Hoarův algoritmus

Autorem tohoto algoritmu je Hoare [24] a využívá opět dělení pole na úseky, popsané v oddílu 5.1.7. Postup je následující:

Použijeme proceduru pro rozdělení pole na úseky, přičemž položíme $l = 1$ a $r = n$. Jako prvek $x$ použijeme $a[k]$. Výsledkem bude částečně uspořádané pole $a$ pro indexy $i$ a $j$, které dostaneme na konci procedury *Rozděl*, bude platit

a) Pro všechna $k < i$ platí $a[k].klíč \leq x.klíč$.

b) Pro všechna $k > j$ platí $a[k].klíč \geq x.klíč$.

c) $i > j$.

Výsledkem bude jedna z následujících tří možností:

1. $j < k < i$. Výměny skončily před dosažením zarážky $x = a[k]$. To znamená, že prvek $a[k]$ dělí pole $a$ na dva úseky tak, jak to požadujeme, a $a[k]$ je hledaným $k$-tým nejmenším prvkem.
2. $j < k$. Prvek $x$ byl příliš malý (resp. jeho klíč byl příliš malý). Operaci dělení musíme zopakovat pro úsek $a[i], \ldots, [r]$.
3. $k < i$. Prvek $x$ byl příliš velký. Operaci dělení musíme zopakovat pro úsek $a[l], \ldots, a[j]$.

**Příklad 5.9**

Podívejme se na příklady obou nepříznivých možností.

a) Je dána posloupnost $(3, 2, 1, 6, 5, 4, 0, 9, 8, 7)$. Hledáme 5. prvek podle velikosti. Jestliže za $x$ zvolíme $a[5] = 5$, dostaneme po prvním průchodu rozdělovací procedurou posloupnost $(3, 2, 1, 0, 4, 5, 6, 9, 8, 7)$ a hodnoty indexů, při kterých se prohledávání zastavilo, budou $i = 6$ a $j = 5$. To znamená, že se prvek $x = 5$ odsunul z původního místa doprava, neboť byl příliš velký. Musíme pokračovat s úsekem $a[1], \ldots, a[5]$.

b) Uvažujme opět posloupnost $(3, 2, 1, 6, 5, 4, 0, 9, 8, 7)$. Tentokrát hledáme 6. prvek jako zarážku bereme $x = a[6] = 4$. Po prvním průchodu dostaneme posloupnost $(3, 2, 1, 0, 4, 5, 6, 9, 8, 7)$ a indexy $i$ a $j$ budou $i = 6$, $j = 5$. Prvek $a[6]$ se odsunul z původního místa doleva, neboť byl příliš malý. Musíme pokračovat s úsekem $a[6], \ldots, a[10]$.

Proceduru pro vyhledání $k$-tého prvku pole můžeme v Pascalu zapsat např. takto (podobně jako v předchozím oddílu předpokládáme, že prvky pole $a$ jsou typu *prvek* a že typ *index* má rozsah $1 \ldots n$ ):

```
procedure Najdi(k: index);
var l,r,i,j:index;
    w,x:    prvek;
begin
  l := 1; r := n;                          {Na počátku celé pole}
  while l < r do begin
    x := a[k];
    i := l; j := r;
    repeat                                                {Rozděl}
      while a[i].klíč < x.klíč do inc(i);
      while a[j].klíč > x.klíč do dec(j);
      if i <= j then begin
```

```
          w := a[i];                                    {Výměna}
          a[i] := a[j]; a[j] := w;
          inc(i); dec(j);
        end;
      until i > j;
      if j < k then l := i;                          {Příliš velký}
      if k < i then r := j;                           {Příliš malý}
    end;
  end;
```

Budeme-li předpokládat, že se při každém rozdělení intervalu zkrátí na polovinu délka úseku, ve kterém je hledaný prvek, dostaneme pro potřebný počet porovnání přibližný výraz

$$C = n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 \approx 2n$$

To znamená, že v nejlepším případě je tato metoda rychlejší než úplné utřídění pomocí nejlepších algoritmů pro vnitřní třídění, neboť ty požadují $O(n \log_2 n)$ operací.

V nejhorším případě, kdy se počet prvků v úseku, ve kterém leží hledaný prvek, při každém rozdělení zmenší pouze o 1, budeme potřebovat $n - 1$ průchodů a tedy $O(n^2)$ porovnání.

## 5.3 Vnější třídění

V tomto oddílu se budeme zabývat metodami pro třídění sekvenčních souborů. Připomeňme se nejdůležitější rozdíly ve srovnání s metodami pro třídění polí.

O souboru předpokládáme, že je uložen na vnějším, zpravidla magnetickém médiu se sekvenčním přístupem. To znamená, že musíme zpracovávat jeden záznam po druhém v pořadí, v jakém jsou v souboru uloženy.

Vedle toho zde předem neznáme rozsah tříděných dat, tj. počet $n$ záznamů v souboru. Lze ale předpokládat, že je tak velký, že se soubor nevejde do operační paměti počítače.

Čtení záznamu ze souboru nebo uložení (zápis) do souboru budeme společně označovat jako *přístup do souboru*. Přístup do souboru trvá o několik řádů déle než porovnávání záznamů, takže je z hlediska efektivity algoritmů pro vnější třídění rozhodující. Proto si při rozboru algoritmů pro vnější třídění budeme všímat jen počtu přístupů do souboru.

Na druhé straně máme obvykle k dispozici dostatečné množství vnější paměti, takže s ní nemusíme příliš šetřit a budeme při třídění využívat pomocných souborů.

### 5.3.1 Přímé slučování

Algoritmus *přímého slučování* (*sort-merge*) odvodíme pomocí metody *rozděl a panuj*. Na něm se seznámíme se základními problémy vnějšího třídění. Protože si však dále ukážeme výhodnější algoritmus, nebudeme jej formulovat jako proceduru.

Budeme uvažovat takto: Protože je tříděný soubor příliš velký, než aby se vešel do paměti, kde bychom na něj mohli použít některou z metod pro vnitřní třídění, rozdělíme jej na dva stejně velké soubory. Pokud dokážeme tyto pomocné soubory (nějak) setřídit, zbývá je sloučit a jsme hotovi.

Na pomocné soubory můžeme použít tutéž úvahu. To znamená, že každý z pomocných souborů opět rozdělíme na poloviny atd.

Po přibližně $\log_2 n$ krocích dospějeme k souborům, které budou obsahovat pouze jeden záznam ($n$ je neznámý počet záznamů v původním souboru). Jenže soubor s jediným záznamem je již automaticky setříděný. To znamená, že stačí umět všechny tyto setříděné soubory sloučit.

Ve skutečnosti si samozřejmě nemůžeme dovolit vytvořit $n$ pomocných souborů (i když jsme řekli, že vnější pamětí nemusíme příliš šetřit). Místo toho ponecháme záznamy řekněme ve dvou souborech, se kterými budeme zacházet jako s posloupností souborů, nebo ještě lépe s posloupností setříděných záznamů.

Předpokládejme, že $P$ a $Q$ jsou dva vzestupně setříděné soubory. Jak je sloučíme, aby vznikl vzestupně utříděný soubor $R$? Nejjednodušší postup je tento:

1. Nejprve přečteme první záznam ze souboru $P$ a uložíme jej do proměnné $p$. Pak přečteme první záznam ze souboru $Q$ a uložíme jej do proměnné $q$.
2. Porovnáme klíče proměnných $p$ a $q$. Je-li $p.klíč \leq q.klíč$, zapíšeme do $R$ proměnnou $p$ a ze souboru $P$ přečteme další záznam. Je-li $p.klíč \geq q.klíč$, zapíšeme do $R$ proměnnou $q$ a ze souboru $Q$ přečteme další záznam.
3. Krok 2 opakujeme, dokud nenarazíme na konec jednoho ze souborů $P$ nebo $Q$.
4. Pokud jsme narazili na konec souboru $P$, okopírujeme do souboru $R$ proměnnou $q$ a zbylé záznamy ze souboru $Q$, jinak okopírujeme do souboru $R$ proměnnou $p$ a zbylé záznamy ze souboru $P$.

Výsledkem bude soubor $R$, který bude obsahovat vzestupně seřazené záznamy ze souborů $P$ a $Q$.
Nyní již můžeme formulovat základ algoritmu třídění přímým slučováním. Přitom předpokládáme, že třídíme soubor $A$. Pomocné soubory označíme $B$ a $C$.

1. Soubor $A$ rozdělíme do souborů $B$ a $C$ tak, že do každého z nich překopírujeme střídavě jeden záznam. Soubor $B$ tedy nyní obsahuje liché záznamy z $A$, soubor $C$ obsahuje sudé záznamy z $A$.
2. Každý ze souborů $B$ a $C$ chápeme jako posloupnost utříděných souborů délky 1. Z $B$ a $C$ vytvoříme soubor $A$ tak, že jednoprvkové úseky sloučíme pomocí výše popsaného postupu do dvouprvkových úseků. Soubor $A$ nyní obsahuje uspořádané dvojice.
3. Soubor $A$ rozdělíme do souborů $B$ a $C$ tak, že do každého z nich překopírujeme střídavě jednu uspořádanou dvojici.
4. Sloučením dvouprvkových úseků z $B$ a $C$ získáme soubor $A$, který bude obsahovat uspořádané čtveřice.
5. Body 3 a 4 opakujeme pro čtveřice, osmice atd., až dostaneme plně utříděný soubor.

Při slučování souborů $B$ a $C$ musíme vzít v úvahu, že počet $n$ prvků původního souboru $A$ nejspíš nebude roven žádné mocnině dvou. Pokud například bude $n$ liché, bude po rozdělení obsahovat soubor $B$ o jeden záznam více než soubor $C$. Obecně může jeden ze souborů na konci obsahovat o jednu $k$-tici méně. Při slučování nesmíme na tyto záznamy zapomenout.

**Příklad 5.10**

Uvažujme soubor (posloupnost)

$$A = (11, 8, 77, 54, 21, 64, 46, 0)\,.$$

Soubor $A$ rozdělíme v prvním kroku do souborů $B$ a $C$ takto:

$$B = (11, 77, 21, 46)\,, \qquad C = (8, 54, 64, 0)$$

Ve druhém kroku sloučíme $B$ a $C$ tak, že vznikne soubor $A$ a s uspořádanými dvojicemi,

$$A = (8, 11, 54, 77, 21, 64, 0, 46)\,.$$

Tento soubor opět rozdělíme do $B$ a $C$, tentokrát tak, že do $B$ dáme první dvojici, do $C$ druhou dvojici atd. Dostaneme

$$B = (8, 11, 21, 64)\,, \qquad C = (54, 77, 0, 46)\,.$$

Sloučením dvojic dostaneme soubor $A$, který bude obsahovat uspořádané čtveřice:

$$A = (8, 11, 54, 77, 0, 21, 46, 64)\,.$$

Tento soubor rozdělíme tak, že do $B$ přijde jedna čtveřice a do $C$ druhá:

$$B = (8, 11, 54, 77)\,, \qquad C = (0, 21, 46, 64)\,.$$

Sloučením těchto dvou souborů dostaneme úplně utříděný soubor $A$,

$$A = (0, 8, 11, 21, 46, 54, 64, 77)\,.$$

Použijeme-li tento postup při třídění souborů na magnetických páskách, budeme potřebovat 3 pásky (3 stojany). Na magnetickém disku můžeme pracovat s několika soubory zároveň, takže stačí jedno zařízení, pokud má dostatečnou kapacitu volného prostoru.

Abychom si dále usnadnili vyjadřování, zavedeme následující označení: Operace s celým souborem budeme nazývat *fáze*; budeme hovořit např. o *fázi rozdělování*, ve které rozdělujeme soubor $A$ na pomocné soubory $B$ a $C$, a o *fázi slučování*, ve které ze dvou souborů $B$ a $C$, které obsahují utříděné $k$-tice, vytvoříme soubor $A$, obsahující utříděné $2k$-tice.

Jako *kroky* budeme označovat nejmenší podprocesy při třídění.

**Analýza třídění přímým slučováním**

Předpokládejme nejprve, že $n = 2^k$ pro nějaké přirozené $k$. Potom budeme v prvním průchodu slučovat celkem $2^k$ jednoprvkových úseků, ve druhém průchodu $2^{k-1}$ dvouprvkových úseků atd. při $k$-tém průchodu sloučíme dva úseky o $2^{k-1}$ prvcích.

Dostaneme tedy celkem $k = \log_2 n$ průchodů.

Bude-li $2^{k-1} \leq n \leq 2^k$, bude počet průchodů opět roven nejvýše $k$. To znamená, že obecně můžeme počet průchodů omezit číslem $[\log_2 n] + 1$, kde $[z]$ znamená celou část čísla $z$.

Každý průchod se skládá z fáze rozdělování a fáze slučování. Fáze rozdělování představuje jedno čtení a jeden zápis do souboru pro každý záznam, tedy celkem $2n$ přístupů do souboru. Také ve fázi slučování musíme každý záznam nejprve přečíst a později jej zase zapsat do souboru. Celkem tedy každý průchod znamená $4n$ přístupů do souboru.

To znamená, že celkový počet přístupů do souborů je omezen hodnotou $4n\,([\log_2 n] + 1)$.

**Vylepšení algoritmu třídění přímým slučováním**

Fáze rozdělování představuje pouhé kopírování záznamů. Vyžaduje jedno čtení a jeden zápis do souboru pro každý záznam a přitom je v podstatě neproduktivní - že zabírá pouze čas. Bohužel ji nelze odstranit; jestliže si ale můžeme dovolit použít ještě jeden pomocný soubor, můžeme ji fázi rozdělování spojit s fází slučování a tak při každém průchodu ušetřit pro každý záznam dva přístupy do souboru. Dostaneme tak upravený algoritmus, který bude používat souborů $A, B, C$ a $D$. Postup pak vypadá takto:

Na počátku rozdělíme tříděný soubor $A$ do souborů $C$ a $D$ podobně jako v původní verzi algoritmu. Při slučování však budeme vzniklé dvojice střídavě zapisovat do souboru $A$ resp. $B$ tak, že první dvojici zapíšeme do $A$, druhou do $B$, třetí do $A$ atd. To znamená, že soubory $A$ a B budou již obsahovat utříděné dvojice.

V dalším průchodu budeme slučovat dvojice ze souborů $A$ a B a vzniklé čtveřice zapisovat střídavě do $C$ a $D$.

Snadno zjistíme, že touto úpravou zkrátíme čas v podstatě na polovinu (ovšem za cenu dalšího souboru - a to nemusí být vždy přijatelné).

### 5.3.2 Třídění přirozeným slučováním

Použijeme-li algoritmu přímého slučování, bude lhostejné, zda je soubor na počátku již částečně utříděn nebo zda v něm jsou prvky uloženy zcela náhodně. Dokonce i pro zcela setříděný soubor se provedou všechny průchody.

Ve skutečnosti i soubor se zcela náhodně uspořádanými prvky může již obsahovat úseky, v nichž prvky za sebou následují ve správném pořadí. *Třídění přirozeným slučováním* umožňuje takové úseky „přirozeným způsobem“ využívat.

Než se pustíme do dalšího výkladu, zavedeme pojem *běh*. Tak budeme označovat každou maximální uspořádanou podposloupnost v rámci tříděné posloupnosti (souboru). To znamená, že v posloupnosti $a_1, \ldots, a_n$

označíme jako běh každou podposloupnost $a_r, \ldots, a_s$, pro kterou platí nerovnosti

$$a_r \leq a_{r-1}, a_s \leq a_{s+1}, a_i \leq a_{i+1} \text{ pro } r \leq i < s. \tag{5.12}$$

Například posloupnost (5, 12, 7, 1, 9, 100) se skládá z běhů (5, 12), (7) a (1, 9, 100).

Při třídění přímým slučováním jsme vycházeli z předpokladu, že výchozí soubor se skládá z $n$ běhů délky 1, a ty jsme postupně slučovali. Při třídění přirozeným slučováním budeme předpokládat, že se výchozí soubor skládá z blíže neurčeného počtu běhů různých délek, a při kopírování a slučování budeme běhy určovat za chodu programu podle (5.12).

„Dynamické“ určování běhů přináší jisté komplikace. Především se mohou i podstatným způsobem lišit počty běhů ve slučovaných souborech. Vzhledem k tomu, že rozdělovací fáze kopíruje běhy střídavě do souboru $B$ a $C$, by se mohlo zdát, že se počty běhů v těchto souborech budou lišit nejvýše o 1. Může se ale stát, že poslední prvek $i$-tého běhu je menší než první prvek $i+2$-tého běhu (a rozdělovací fáze umístí tyto dva běhy bezprostředně za sebe). Slučovací fáze je pak bude chápat jako jediný běh.

Podívejme se na posloupnost (1, 5, 2, 9, 7, 11, 6, 22). Rozdělovací fáze zde najde 4 běhy po dvou prvcích a tuto posloupnost rozdělí do souborů $B$ a $C$ takto:

$$B = (1, 5, 7, 11)\,, \qquad C = (2, 9, 6, 22)\,.$$

V souboru $B$ se ovšem dva běhy „samovolně sloučily“, takže slučovací fáze najde v $B$ pouze jediný běh.

### Procedura pro přirozené slučování

V tomto odstavci budeme předpokládat, že se tříděný soubor skládá ze záznamů tvaru

```
type prvek = record
       klíč: integer;
       d:    data
     end;
```

Záznamy třídíme podle vzrůstající hodnoty klíče.

### Objektový typ Soubor

Při třídění slučováním je občas vhodné znát nejen hodnotu právě čteného záznamu v souboru, ale i hodnotu záznamu následujícího. V klasickém Pascalu k tomu slouží tzv. *přístupová proměnná souboru*. Turbo Pascal tuto možnost nenabízí, můžeme se ji ale snadno naprogramovat. Jedno z možných řešení představuje následující objektový typ *soubor*.

```
soubor = object
  f:file of prvek;
  pp: prvek;             {hraje roli přístupové proměnné souboru}
  def: boolean;
  procedure read(var x: prvek);
  procedure assign(s: string);
  procedure write(x: prvek);
  procedure reset;
  procedure rewrite;
  procedure close;
  function eof: boolean;
end;
```

Atribut *f* představuje proměnnou typu soubor tak, jak ji známe z Turbo Pascalu; *pp* bude hrát roli přístupové proměnné a *def* použijeme při nové definici funkce *eof*.

Při zápisu do souboru v tomto algoritmu přístupovou proměnnou nepotřebujeme. Metody *assign, rewrite, write* a *close* budou prostě volat stejnojmenné standardní procedury, např.

```
procedure soubor.assign(s: string);
begin
   system.assign(f,s);
end;
```

Metody pro čtení ovšem přístupovou proměnnou budou muset využívat. Při otevření souboru pro čtení proto ihned přečteme první záznam (pokud soubor nějaký obsahuje) a uložíme jej do *pp*. Pokud je soubor prázdný, vložíme do proměnné *def* hodnotu **false**.

```
procedure soubor.reset;
begin
   system.reset(f);
   if not system.eof(f) then begin
        system.read(f, pp);
        def := true;
   end else def := false;
end;
```

Procedura *soubor.read* vrátí hodnotu *pp*, přečte další prvek (pokud existuje) a připraví ho do *pp*. Pokud narazí na konec souboru, uloží do *def* hodnotu **false**.

```
procedure soubor.read(var x: prvek);
begin
   if def then x := pp;
   if not system.eof(f) then begin
        system.read(f, pp);
        def := true;
   end else def := false;
end;
```

Funkce *eof* prostě vrací hodnotu atributu *def*, který signalizuje, zda je něco v přístupové proměnné.

```
function soubor.eof;
begin
   eof := not def;
end;
```

### Procedura pro přirozené slučování souborů

Zde uvedeme proceduru *PřirozenéSlučování* v jednodušší podobě, s oddělenou rozdělovací a slučovací fází. Třídíme soubor *C*, který pokládáme za globální instanci typu *soubor*. Pomocné soubory jsou *A* a *B*. Tělo procedury *PřirozenéSlučování* se skládá z opakování rozdělování a slučování (procedury, které to provádějí, se jmenují výstižně *Rozděl* a *Sluč*). Přitom se v proměnné l počítají běhy, vytvořené při slučování; třídění skončí, jestliže výsledný soubor obsahuje pouze jeden běh.

Procedura *Rozděl* v cyklu volá proceduru pro zkopírování jednoho běhu, *ZkopírujBěh*, a střídavě kopíruje běhy do *A* a *B*. Procedura *Sluč* v cyklu slučuje běhy ze souborů *A* a *B* a zapisuje je do *C*; k tomu použije proceduru *SlučBěh*. Pokud obsahuje jeden ze souborů více běhů než druhý, překopíruje nakonec zbylé běhy do *C*. Běhy přitom počítá v proměnné *l*.

Procedura *ZkopírujBěh* kopíruje jednotlivé prvky (volá proceduru *ZkopírujPrvek*), dokud nenarazí na konec běhu. Procedura *ZkopírujPrvek* kopíruje jednotlivé prvky z jednoho souboru do druhého a přitom zjišťuje, zda nenastal konec běhu nebo konec souboru (což je také konec běhu).

Slučovací fáze je poněkud složitější. Procedura *Sluč* se odvolává na proceduru *SlučBěh*, která sloučí aktuální běh ze souboru A s aktuálním během ze souboru *B*. Přitom v cyklu porovnává prvky, uložené v přístupových proměnných *A* a *B* a menší z nich zapíše do *C*. Jestliže v jednom ze souborů *A*, *B* běh skončí, překopíruje zbytek běhu ze zbývajícího souboru do *C*. K tomu lze použít proceduru *ZkopírujBěh*.

```
{****** Třídění souboru metodou přirozeného slučování *****}
procedure přirozenéSlucovani;
 var l:   integer;
     kb:  boolean;                          {Indikuje konec běhu}
     A, B: soubor;
 procedure ZkopírujPrvek (var X,Y: soubor); {Kopíruje jednotlivý}
  var buf: prvek;              {prvek z X do Y a přitom zjišt'uje}
 begin                                    {konec běhu nebo souboru}
     X.read(buf);
     Y.write(buf);
     if X.eof then kb := true                {Test konce souboru}
     else kb := buf.klíč > X.pp.klíč;           {Test konce běhu}
 end;
 procedure ZkopírujBěh(var X,Y: soubor);
 begin               {zkopírování jednoho běhu ze souboru X do Y}
   repeat
     ZkopírujPrvek(X,Y)                {V cyklu kopíruje prvky,}
   until kb;                          {dokud nedojde na konec běhu}
 end;{zkopírujBěh}
 procedure Rozděl;                              {Rozdělovací fáze}
 begin                                {Ze souboru c do souborů a,b}
   repeat
     ZkopírujBěh(C,A);                   {Střídavě kopíruje běhy}
     if not C.eof then ZkopírujBěh(C,B)         {z C do A nebo B}
   until C.eof;
 end; {Rozděl}
 procedure SlučBěh;
 begin {ze souborů A,B do C}
   repeat                                  {Porovná prvky v přístu-}
     if A.pp.klíč < B.pp.klíč then begin      {pových proměnných}
        ZkopírujPrvek(A,C);                   {menší překopíruje}
        if kb then ZkopírujBěh(B,C)
     end else begin
        ZkopírujPrvek(B,C);
        if kb then ZkopírujBěh(A,C)          {Překopíruje zbytek}
     end;
   until kb;
 end; {SlučBěh}
 procedure Sluč;                                 {Slučovací fáze}
 begin                                {Ze souborů A,B do souboru C}
  while not A.eof and not B.eof do begin
    SlučBěh;                                {Slučuje běhy z A a B}
    inc(l);                                          {a počítá je}
  end{while 1};
  while not A.eof do begin           {Pokud má A více běhů než B}
    ZkopírujBěh(A,C);                          {okopíruj zbytek}
    inc(l);
  end{while 2};
  while not B.eof do begin           {Pokud má B více běhů než A}
    ZkopírujBěh(B,C);                          {okopíruj zbytek}
    inc(l);
  end{while 3};
 end{slučování};
begin {přirozené slučování}
  A.assign('data.111');
  B.assign('data.222');
  repeat                                         {Rozdělovací fáze}
    A.rewrite; B.rewrite;
```

```
      C.reset;
      Rozděl;
      A.reset; B.reset;                          {Slučovací fáze}
      C.close;
      C.rewrite;
      l:=0;
      Sluč;
    until l = 1;              {Jediný běh, tj. soubor = běh: KONEC}
  end;{přirozené slučování}
```

Je zřejmé, že účinnost tohoto algoritmu bude lepší než účinnost přímého slučování. Nejhorší případ nastane, jestliže všechny běhy v původním souboru budou mít délku 1 a při rozdělování nebo slučování nikdy nedojde k „samovolnému sloučení“ dvou po sobě následujících běhů. V takovém případě bude v proceduře Sluč třeba $O(\log_2 n)$ průchodů cyklem **repeat** a tedy celkový počet přístupů do souborů bude $O(n \log_2 n)$.

Nejlepší případ nastane, jestliže bude vstupní soubor již uspořádaný. V takovém případě najde slučovací procedura jediný běh a skončí po prvním průchodu, takže budeme potřebovat $4n$ přístupů do souboru.

### Další možná vylepšení

Metody, se kterými jsme se dosud seznámili, lze označit za základní. Existují ovšem další, propracovanější a také účinnější metody. Řekneme si základní myšlenky některých z nich. Podrobnější informace najdete např. v [1] nebo [5].

### Vícecestné slučování

Máme-li k dispozici dostatek volného místa ve vnější paměti (dostatek stojanů pro magnetické pásky), můžeme použít více pomocných souborů. Jestliže v rozdělovací fázi rozdělíme zdrojový soubor do $N$ pomocných souborů a z nich pak budeme jednotlivé běhy slučovat, dostaneme *N-cestné slučování*. Podobně, jako jsme v případě metody přímého slučování odvodili složitost O(n.log2 n), odvodíme pro N-cestné slučování složitost $O(n \log_N n)$.

Poznamenejme, že nejde o řádové zlepšení, neboť $\log_N n = \frac{\log_2 n}{\log_2 N}$.

### Polyfázové slučování

Tuto metodu navrhl L. R. Gilstadt [28]. Vychází ze snahy lépe využít pomocných souborů (to je důležité zejména při třídění na magnetických páskách, kde uzavření a znovuotevření souboru znamená zdlouhavé převíjení pásky).

Podívejme se na slučování ze souborů $A$ a $B$ do souboru $C$. Jakmile jsme při přirozeném slučování narazili na konec souboru $A$, okopírovali jsme zbytek souboru $B$ do $C$ a tím skončil průchod. Jestliže narazíme na konec souboru $A$ při polyfázovém třídění, uzavřeme ho, otevřeme ho pro zápis. Proces slučování bude pokračovat: sloučíme zbytek souboru $B$ se souborem $C$ a výsledek zapisujeme na $A$.

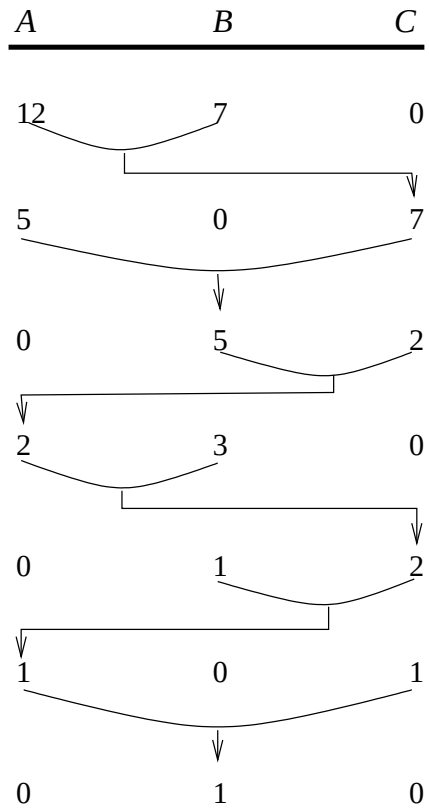
Protože se $n$ běhů ve vstupním souboru změní na $n$ běhů ve výstupním souboru, stačí si vést evidenci o počtu běhů na jednotlivých páskách. Proces skončí, až dospějeme do situace, kdy sloučíme dva vstupní soubory, obsahující po jednom běhu.

Použijeme-li více souborů, bude postup podobný.

### Příklad 5.11

Ve vstupním souboru $A$ je 12 běhů, ve vstupním souboru $B$ ke 7 běhů (viz obr. 5.9). Při prvním průchodu se 7 běhů ze souboru $B$ sloučí s 7 běhy ze souboru $A$ a vznikne 7 běhů v souboru $C$. To znamená, že soubor $B$ je nyní „prázdný“ (vyčerpali jsme všechny běhy), zatímco v souboru $A$ zbylo 5 běhů. Budeme tedy pokračovat slučováním 5 zbylých běhů ze souboru $A$ se 7 běhy z $C$ a výsledek budeme ukládat do $B$.

Ve druhém průchodu se sloučí 5 běhů z $A$ s 5 běhy z $C$. Výsledkem bude prázdný soubor $A$, v $C$ zbudou 2 běhy a $B$ bude obsahovat 5 běhů. Novým výstupním souborem bude $A$.




Obr. 5.9: Polyfázové třídění (k příkladu 5.11)

Třetí průchod sloučí 2 běhy z $C$ se dvěma běhy z $B$, dostaneme 2 běhy v $A$ a v $B$ zbudou 3 běhy. Soubor $C$ se vyprázdní a v příštím průchodu bude sloužit jako výstupní.
Při čtvrtém průchodu se sloučí 2 běhy ze souboru $A$ se dvěma běhy z $B$ a vzniknou dva běhy v $C$. Vyprázdní se soubor $A$, v $B$ zbude jeden běh.

Pátý průchod sloučí jediný běh ze souboru $B$ s jedním během v $C$. $B$ se vyprázdní, v $C$ zbude jeden běh a v $A$ je také jeden běh.

Poslední průchod sloučí jediný běh v $A$ s jediným během v $C$. V $B$ je utříděný soubor.

Další vylepšení se mohou týkat rozdělení běhů na páskách.

## 5.4 Některé další metody třídění

V tomto odstavci se seznámíme s dalšími metodami pro třídění.

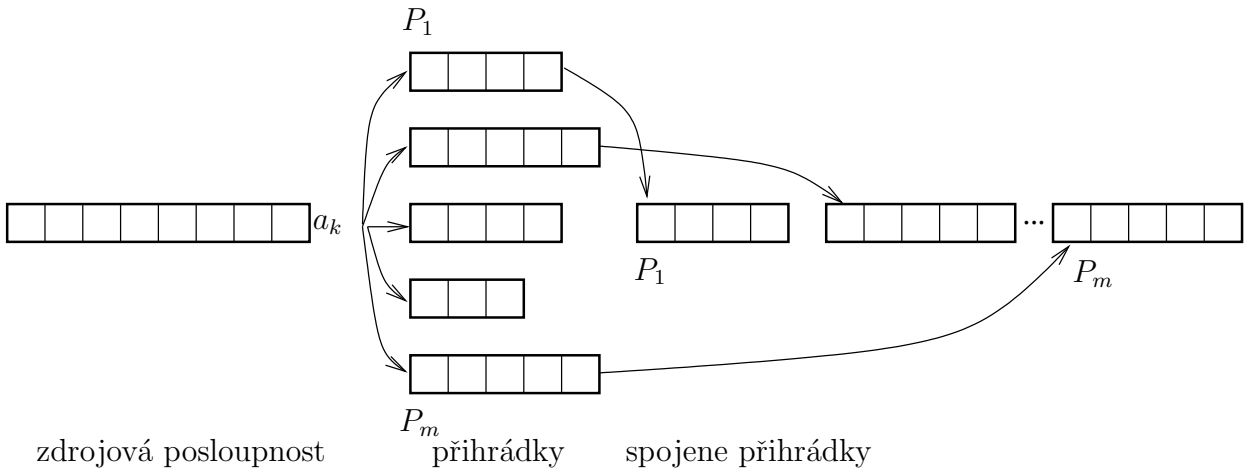
### 5.4.1 Přihrádkové třídění

*Přihrádkové třídění* (*bucketsort*) lze použít v případě, kdy klíč, podle kterého třídíme, může nabývat poměrně malého počtu hodnot. Bez újmy na obecnosti můžeme předpokládat, že pro všechny prvky $a_i$, které se mohou ve tříděné posloupnosti vyskytnout, platí

$$a_i.klíč \in \hat{M} = \{1, 2, K, M\}\,.$$

kde $M$ je malé pevné číslo. Naším úkolem je opět seřadit danou posloupnost $a_i$, $i = 1, \ldots, n$, podle rostoucího klíče.

Základní myšlenka přihrádkového třídění je velmi jednoduchá. Definujeme $N$ přihrádek $P_1, \ldots, P_M$. Nyní budeme postupně procházet posloupnost $a_i$ a jednotlivé prvky budeme rozřazovat do odpovídajících přihrádek.




Obr. 5.10: Zdrojová posloupnost se roztřídí do přihrádek a ty se pak sloučí

Prvky s klíčem 1 do přihrádky $P_1$, prvky s klíčem 2 do přihrádky $P_2$ atd. Potom přihrádky sloučíme v pořadí $P_1 + P_2 + \cdots + P_M$ (viz obr. 5.10).

Jako přihrádky mohou posloužit např. fronty nebo soubory (záleží na rozsahu tříděné posloupnosti). Důležité je, aby struktura, kterou použijeme v roli přihrádky, zachovávala pořadí, ve kterém do ní byly prvky vloženy; jinak by tento algoritmus nebyl stabilní, měnil by pořadí prvků, které mají stejnou hodnotu klíče.
Algoritmus přihrádkového třídění popisuje procedura *PřihrádkovéTřídění*. V něm předpokládáme, že máme k dispozici objektový typ *fronta*, který implementuje tuto datovou strukturu. Konstruktor *fronta.Vytvoř* vytvoří prázdnou frontu, metoda *fronta.Vlož* vloží prvek na konec fronty a metoda *fronta.Vyjmi* vyjme prvek z čela fronty. Booleovská funkce *fronta.Prázdná* vrátí **true**, je-li fronta prázdná. Tříděná posloupnost je uložena v poli $a$.

```
procedure PřihrádkovéTřídění;
var P: array [1..M] of fronta;
    i: 0..n;
    j: 1..M;
begin
  for j := 1 to M do P[j].Vytvoř;        {Vytvoří prázdné fronty}
  for i := 1 to n do
    P[a[i].klíč].Vlož(a);          {Vloží a[i] do správné fronty}
  i := 0;
  for j := 1 to M do              {Přepíše frontu zpět do pole a}
    while not P[j].Prázdná do begin
      inc(i);
      P[j].Vyjmi(a[i])       {Vyjme prvek z P[j] a uloží do a[i]}
    end;
end;
```

**Analýza přihrádkového třídění**

Nejprve se podívejme na spotřebu paměti. Pokud bychom implementovali frontu jako pole, museli bychom počítat s možností, že všechny prvky tříděné posloupnosti mohou mít stejný klíč. To znamená, že všechny fronty musí mít délku $M$, takže potřebujeme paměť pro celkem $Mn$ prvků navíc. To je zpravidla naprosto nepřijatelné.

Musíme tedy frontu implementovat jako seznam. V takovém případě budeme potřebovat navíc $n$ prvků (ve skutečnosti vzhledem k „administrativě“ seznamu o něco málo více). Můžeme ovšem očekávat, že program bude o něco pomalejší, než kdybychom použili pole - opět díky „administrativě“ při práci se seznamem.

Nyní se podíváme na počet operací. Každý prvek jednou vyjmeme z pole $a$, vypočteme jeho klíč a vložíme ho do fronty. Na konci jej z fronty vyjmeme a uložíme do pole. To znamená, že na každý prvek připadají dva

přesuny a jeden výpočet klíče; celkem tedy $2n$ přesunů a $n$ výpočtů klíče. Přihrádkové třídění tedy vyžaduje $O(n)$ operací.

### 5.4.2 Lexikografické třídění

Nejprve musíme definovat *lexikografické uspořádání*. Je-li $D$ lineárně uspořádaná množina a $m$ přirozené číslo, pak lexikografické uspořádání množiny $D^m$ (tedy množiny uspořádaných $m$-tic prvků z množiny $D$) je uspořádání definované tak, že nerovnost

$$(a_1, a_2, \ldots, a_m) \leq (b_1, b_2, \ldots, b_m) \tag{5.13}$$

platí, právě když se tyto dvě $m$-tice rovnají, $(a_1, a_2, \ldots, a_m) = (b_1, b_2, \ldots, b_m)$, nebo $a_s \leq b_s$ pro nejmenší index $s$ takový, že $a_s \neq b_s$.

**Příklad 5.12**

Podívejme se na dva příklady lexikografického uspořádání.

Je-li $D$ anglická abeceda, $D = \{a, b, c, \ldots, z\}$, s uspořádáním $a \leq b \leq \ldots \leq z$, představuje $D^m$ množinu všech řetězců (slov) o $m$ písmenech a lexikografické uspořádání znamená obvyklé uspořádání podle abecedy.

Je-li $D$ množina číslic, $D = \{0, 1, \ldots, 9\}$ s obvyklým uspořádáním, představuje $D^m$ řetězec $m$ číslic. Každý z těchto řetězců můžeme pokládat za zápis celého čísla v desítkové soustavě (připouštíme čísla, začínající nevýznamnými nulami, např. 00014). Snadno se přesvědčíme, že v tomto případě se lexikografické uspořádání je vlastně obvyklé uspořádání celých čísel.

Uvažujme nyní posloupnost $A_1, A_2, \ldots, A_n$, složenou z prvků $D^m$. Tyto posloupnost $m$-tic chceme setřídit na základě lexikografického uspořádání (5.13). Algoritmus lexikografického třídění je velmi jednoduchý, i když na první pohled nečekaný. Posloupnost $A_1, A_2, \ldots, A_n$ budeme opakovaně třídit pomocí algoritmu přihrádkového třídění podle poslední složky $m$-tice, pak podle předposlední složky, podle $(m-2)$-té složky atd.

Ze stability přihrádkového třídění plyne, že při prvním průchodu budou ve výsledné posloupnosti prvky $A_1$ v pořadí, určeném posledním znakem. Ve druhém průchodu budou ve výsledné posloupnosti prvky $A_1$ v pořadí, určeném předposledním znakem; pokud však bude u $A_k$ a $A_l$ stejný předposlední znak, budou ve stejném pořadí, jako byly po prvním průchodu - tedy v pořadí, určeném posledním znakem. Tuto úvahu můžeme zopakovat i pro další průchody; z ní plyne, že výsledkem bude lexikograficky setříděná posloupnost.

Algoritmus lexikografického třídění zapíšeme opět v Pascalu. Přitom předpokládáme, že prvek tříděného pole $a[i]$ obsahuje pole $m$ klíčů.

```
procedure LexikografickéTřídění;
var P: array [1..M] of fronta;
    i: 0..n;
    j: 1..M;
    l: 1..k;
begin
 for l := k downto 1 do begin     {Opakovaná přihrádková třídění}
  for j := 1 to M do P[j].Vytvoř;        {Vytvoří prázdné fronty}
  for i := 1 to n do
    P[a[i].klíč[l]].Vlož(a);       {Vloží a[i] do správné fronty}
  i := 0;
  for j := 1 to M do              {Přepíše frontu zpět do pole a}
    while not P[j].Prázdná do begin
      inc(i);
      P[j].Vyjmi(a[i])       {Vyjme prvek z P[j] a uloží do a[i]}
    end;
 end;
end;
```

Je-li počet klíčů $m$ a množství možných hodnot klíčů $M$ pevné, potřebujeme $m$ průchodů přihrádkového třídění. Složitost lexikografického třídění tedy je $mO(n) = O(n)$.

**Poznámka**

Z definice lexikografického třídění by se mohlo zdát, že je rozumné začít porovnáním prvních klíčů, pokud se shodují, tak porovnat druhé klíče atd. Jestliže se takový algoritmus pokusíte navrhnout, zjistíte, že bude velmi komplikovaný.

**Poznámka o abecedním třídění**

Jako nejdelší české slovo se obvykle uvádí *nejnedoobhospodařovávatelnějšími*, které má 32 písmen. Mohlo by se tedy zdát, že na abecední třídění abecední třídění českých slov stačí vzít řetězce dlouhé řekněme 60 znaků (abychom pokryli i výrazy složené z několika slov) a použít algoritmu lexikografického třídění.

Situace je bohužel poněkud komplikovanější, a to nejen díky spřežce *ch*, která se chápe jako jedno písmeno. Postup při abecedním řazení v češtině v češtině a slovenštině je předepsán státní normou [29], alespoň v Česku stále platnou. V tomto odstavci si povíme o nejdůležitějších zásadách abecedního řazení, které z této normy plynou.

Za primární třídicí znaky se pokládají písmena tzv. *české a slovenské standardizované unifikované abecedy*

a b c č d e f g h ch i j k l m n o p q r ř s š t u v w x y z ž

Písmena v této abecedě neuvedená,

á ä ď é ě í Ĺ ľ ň ó ô ŕ ť ú ů ý

mají sekundární třídicí platnost. To znamená, že je nejprve chápeme jako písmena bez diakritických znamének (podobně se zachází i s písmeny s diakritickými znaménky jiných národních abeced, včetně nerozepsaných přehlásek; to znamená, že *ťuhýk* je v abecedě před *tygrem*). K tomu, že jde o odlišné znaky, přihlížíme pouze v případě jinak zcela totožných hesel.

Další pravidla říkají, v jakém pořadí se řadí hesla, která se liší diakritickými znaménky na různých místech, různými diakritickými znaménky na témže místě, použitím malých a velkých písmen apod. Tato pravidla jsou sice komplikovaná, ale lze je bez problémů zvládnout.

Co ale činí z českého abecedního řazení záležitost v podstatě algoritmicky neřešitelnou, jsou např. pravidla pro zacházení s číslicemi. Např. heslo *10 pohádek* se bude řadit jako *DESET pohádek*, neboť jde o „slovní součást názvu“ knihy, zatímco *Geometrie pro 8. ročník* se zařadí před heslo *Geometrie pro 9. ročník*, neboť „zde číslice výrazně určují pořadí“ (citováno podle [29], str. 8). Jinou lahůdkou je zacházení s čínskými jmény, které se chápou jako jedno slovo, i když obsahují mezery.

Z tohoto povídání plynou dvě mravní naučení:

1. Abecední třídění tak, jak je popsáno v [29], je závislé také na významu řazených hesel. To znamená, že pokud máme pouze soubor řetězců a budeme jej třídit počítačem, zákonitě se dopustíme prohřešků proti této normě. V běžných souvislostech se ale prohřešky v detailech tolerují a při třídění např. hesel ve slovníku lze vždy k heslu připojit klíč, který způsob třídění jednoznačně předepíše (např. k heslu *10 pohádek* připojíme pomocné heslo Deset pohádek, podle kterého ho budeme třídit).

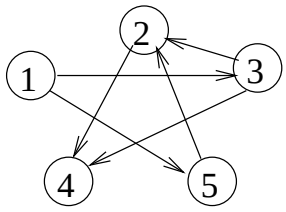
2. Procedura pro porovnání dvou řetězců podle pravidel českého abecedního třídění bude natolik komplikovaná, že při třídění může zabrat více času než přesun záznamu v paměti.

### 5.4.3 Topologické třídění

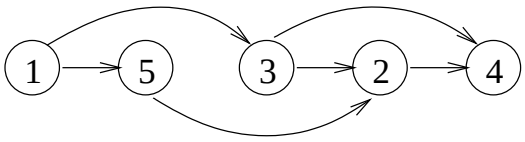
*Topologické třídění* se používá jako označení pro třídění částečně uspořádaných posloupností. Než se pustíme do dalšího výkladu, zavedeme si potřebné pojmy.

**Částečné uspořádání**

Řekneme, že množina $S$ je částečně uspořádaná, je-li pro některé dvojice $(x, y) \in S \times S$ definována relace „$\blacktriangleleft$“, která splňuje následující podmínky:




Obr. 5.11: Orientovaný graf, znázorňující částečné uspořádání (5.14) z příkladu 5.13




Obr. 5.12: Utříděná částečně uspořádaná množina z příkladu 5.13

1. Relace „◀“ je tranzitivní: platí-li pro nějaká $x, y, z \in S$ zároveň relace $x \blacktriangleleft y$ a $y \blacktriangleleft z$, platí také $x \blacktriangleleft z$.
2. Relace „◀“ je asymetrická: jestliže pro nějaká $x, y \in S$ platí relace $x \blacktriangleleft y$, nemůže platit zároveň relace $y \blacktriangleleft x$.
3. Relace „◀“ je ireflexivní: pro žádné $x \in S$ neplatí relace $x \blacktriangleleft x$.

Relace se pochopitelně nazývá *částečné uspořádání*. Částečné uspořádání konečné množiny lze znázornit orientovaným grafem; příklad vidíme na obr. 5.11. Šipky, znázorňující orientaci hran, směřují k uzlu, znázorňujícímu „větší“ prvek (prvek na pravé straně znaku „◀“). Z podmínek, které definují částečné uspořádání, plyne, že takovýto graf nesmí obsahovat cykly.

**Příklad 5.13**

Vezmeme množinu $M = \{1, 2, 3, 4, 5\}$ a na ní definujeme částečné uspořádání tak, že přímo vyjmenujeme dvojice, které ho tvoří:

$$1 \blacktriangleleft 5, 1 \blacktriangleleft 3, 3 \blacktriangleleft 2, 3 \blacktriangleleft 4, 5 \blacktriangleleft 2, 2 \blacktriangleleft 4 \tag{5.14}$$

Toto částečné uspořádání lze znázornit orientovaným grafem na obr. 5.11.

Jiný příklad částečného uspořádání: Vezměme množinu všech podmnožin $R$ neprázdné množiny $T$. Částečné uspořádání množiny $R$ definuje relace „$\subset$“ ($A \subset B$ zde znamená „$A$ je vlastní podmnožinou $B$“).

Částečné uspořádání bychom také mohli zavést pro axiomy a matematické věty, které z nich plynou. Relaci $A \blacktriangleleft B$ by pak odpovídal vztah „z tvrzení $A$ plyne tvrzení $B$“.

**Topologické třídění**

Chceme-li třídit posloupnost $a_i$, pro jejíž prvky je definováno částečné uspořádání, musíme toto částečné uspořádání vnořit do nějakého lineárního uspořádání. To znamená uspořádat prvky posloupnosti ai tak, aby pro každou dvojici $a_i$, $a_j$ ze vztahu $i < j$ plynulo, že buď platí $a_i \blacktriangleleft a_j$ nebo mezi $a_i$ a $a_j$ není relace „◀“ definována. Je jasné, že výsledek topologického třídění nemusí být jednoznačně definován.

Utřídění částečně uspořádané posloupností můžeme znázornit orientovaným grafem, ve kterém budou všechny uzly ležet v jedné přímce a všechny šipky, vyjadřující orientaci hran, budou směřovat doprava (viz např. obr. 5.12 ).

Ukážeme si, jak může vypadat algoritmus pro topologické třídění. Ve vstupním souboru máme seznam relací ve tvaru uspořádaných dvojic, popisujících relace částečného uspořádání. Soubor končí číslem 0. Chceme tyto údaje setřídit a vytisknout. Pro jednoduchost budeme předpokládat, že jde o celá čísla.

Např. vstupní soubor, obsahující čísla $1, 5, 1, 3, 3, 2, 3, 4, 5, 2, 2, 4$ popisuje množinu $\{1, \ldots, 5\}$, na které je definováno částečné uspořádání vztahy (5.14) z příkladu 5.13.

Ze zadání plyne, že prvky tříděné posloupnosti jsou položky, které se ve vstupním souboru vyskytují alespoň jednou. Při třídění využijeme dynamické datové struktury $S$, odvozené od jednosměrného seznamu. Algoritmus, který použijeme, můžeme shrnout do následujících tří bodů:

1. Prvky při vstupu ze souboru ukládáme do seznamu $S$. O každém prvku průběžně zjišťujeme, kolik má v daném částečném uspořádání předchůdců a následovníků.

2. Potom v $S$ vyhledáme prvky, které nemají žádného předchůdce („nejmenší prvky“). Takový prvek musí být ve vstupním souboru - a tady v $S$ - alespoň jeden, neboť jinak by graf tohoto uspořádání obsahoval cyklus.

3. Tyto „nejmenší“ prvky vytiskneme a odstraníme z $S$. Zbude mám částečně uspořádaná množina, takže v ní opět vyhledáme „nejmenší“ prvky, tj. prvky, které nemají předchůdce, a odstraníme je. Tento postup budeme opakovat až do úplného vyprázdnění $S$.

Seznam $S$ bude obsahovat o každém z prvků tyto údaje: hodnotu prvku, množinu jeho následovníků a počet jeho předchůdců v daném částečném uspořádání. Informace o následnících jednotlivých prvků tříděné množiny se budeme dozvídat postupně při čtení vstupního souboru. Proto připojíme ke každému prvku množiny pomocný seznam jeho následovníků, nebo ještě lépe seznam odkazů na následovníky.

Strukturu, která bude prvek seznamu reprezentovat, nazveme *hlavní*, neboť bude hlavní složkou seznamu. Vedle toho budeme ale potřebovat strukturu *pomocný* pro reprezentaci prvků pomocných seznamů následovníků. Jejich deklarace budou

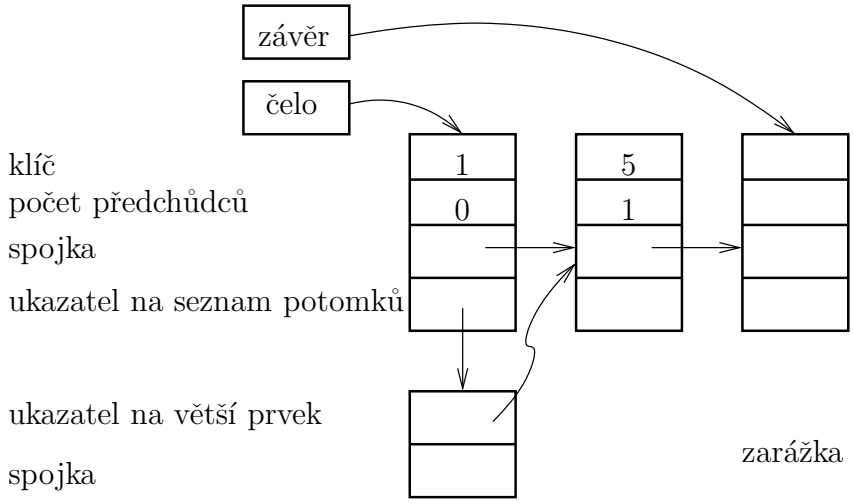
```
type uHlavní = ^Hlavní;
     uPomocný = ^Pomocný;
     Hlavní = record                      {Prvek hlavního seznamu}
        klíč, počet: integer;            {Klíč a počet předchůdců}
        upom: uPomocný;             {Odkaz na seznam následovníků}
        další: uHlavní;                    {Odkaz na další prvek}
     end;
     Pomocný = record                 {Prvek seznamu následovníků}
       id: uHlavní;                     {Ukazatel na následovníka}
       další: uPomocný;          {Ukazatel na další prvek seznamu}
     end;
```

Složka klíč obsahuje prvek tříděné posloupnosti.

Na obr. 5.13 vidíme seznam, který se vytvořil po přečtení první dvojice 1, 5 ze vstupního souboru. První prvek seznamu obsahuje položku 1, žádná položka není menší. Jediný prvek připojeného pomocného seznamu obsahuje ukazatel na větší položku, v tomto případě na 5. Druhý prvek seznamu obsahuje položku 5 a informaci, že má jediného předchůdce. Protože nemá (zatím) následovníky, je připojený pomocný seznam prázdný.
Celý program se bude skládat ze čtyř etap; po přípravných operacích přečteme vstupní soubor a na základě údajů, které obsahuje, vytvoříme seznam prvků a relací mezi nimi. Ve třetí etapě sestavíme seznam prvků, které nemají žádné předchůdce (jsou ve smyslu daného částečného uspořádání „nejmenší“).

V posledním kroku odstraníme ze seznamu nejmenší prvky. Zároveň samozřejmě v následovnících odstraněných prvků zmenšíme odpovídajícím způsobem údaje o počtu předchůdců. Výsledkem bude opět částečně uspořádaná množina, takže bude zase obsahovat alespoň jeden nejmenší prvek. Tento postup budeme opakovat až do vyprázdnění seznamu.

```
Hlavní program tedy bude
begin
  Příprava;
  VytvořSeznam;
  NajdiNejmenší;
  VypišSeznam;
end.
```




Obr. 5.13: po přečtení první dvojice ze vstupního souboru

Tentokrát nepoužijeme objektových typů. To znamená, že musíme definovat ukazatele na počátek a konec seznamu jako globální proměnné (budou se jmenovat *čelo* a závěr a budou typu *uHlavní*). Vedle toho definujeme globální proměnnou z typu integer, která bude sloužit jako počítadlo prvků v seznamu.

Procedura *Příprava* smaže obrazovku a vytvoří prázdný seznam se zarážkou:

```
procedure Příprava;
begin
  clrscr;
  new(celo);         {vytvoření prázdného seznamu hlavních prvků}
  zaver := celo;
  z := 0;                                          {počítadlo prvků}
end;
```

V proceduře *VytvořSeznam* přečteme vždy jednu dvojici $x, y$ ze souboru $f$. Podíváme se, zda je $x$ již v seznamu; pokud tam není, přidáme ho tam. Dále se podíváme, zda je v seznamu $y$, a pokud není, zařadíme ho. Zároveň k $x$ připojíme informaci, že jeho následovníkem je $y$ (tj. do pomocného seznamu, připojeného k $x$, vložíme prvek s adresou $y$) a u prvku $y$ zvýšíme počet předchůdců.

Pro vyhledání prvku v seznamu a jeho případné zařazení použijeme funkci *JeVSeznamu*, která zároveň vrátí adresu tohoto prvku:

```
function JeVSeznamu(w: integer): uHlavní;
{určení hlavního prvku s klíčem w a případně zařazení do seznamu}
var h: uHlavní;
begin  {JeVSeznamu}
    h := čelo;
    závěr^.klíč := w;                          {definice zarážky}
    while h^.klíč <> w do h := h^.další;               {hledani}
    if h = závěr then begin   {prvek není v seznamun - přidej ho}
      new(závěr);
      inc(z);
      h^.počet := 0;
      h^.upom := nil;
      h^.další := závěr;
            {zarážka se stala posledním prvkem, vytvoří se nová}
    end;
    JeVSeznamu := h;
end; {JeVSeznamu}
```

Procedura *VytvořSeznam* čte jednotlivé dvojice, zařazuje je do seznamu a připojuje informace o předchůdcích a následovnících. Přitom předpokládáme, že údaje jsou uloženy v souboru *topo.dta*

```
procedure VytvořSeznam;
var p, q: uHlavní;
    t:    uPomocný;
    x, y: integer;
    f: text;
begin                                                    {Vstup}
  assign(f, 'topo.dta');              {otevření souboru topo.dta}
  reset(f);
  read(f,x);
  while x <> 0 do begin              {Vstupní soubor končí nulou}
    read(f,y);
    writeln(x,' ',y);
    p := JeVSeznamu(x); q := JeVSeznamu(y);
    new(t);                          {Odkaz na nového následovníka}
    t^.id := q;
    t^.další := p^.upom;
    p^.upom := t;
    inc(q^.počet);                   {Informace o počtu předchůdců}
    read(f,x)
  end;
  close(f);
end;
```

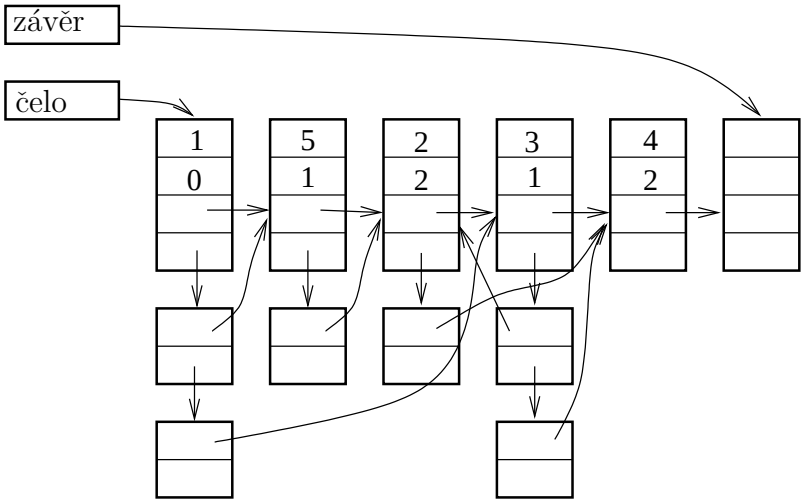
V dalším kroku vyhledáme nejmenší prvky, a spojíme je do nového seznamu. K tomu využijeme jejich spojek (odkazů na následující prvky), neboť původní seznamovou strukturu už nebudeme potřebovat. Ukazatelem na první prvek nového seznamu bude proměnná *čelo*. (Nemusíme se bát, že tak zničíme strukturu seznamu a ztratíme spojení na některé jeho části; každý z nejmenších prvků má nějaké následovníky a odkazy na ně jsou v připojeném pomocném seznamu. Snadno zjistíme, že se touto cestou můžeme dostat ke všem prvkům seznamu.

Spojení nejmenších prvků do nového seznamu obstará procedura *NajdiNejmenší*. Prvky vkládá na začátek seznamu, takže nyní budou v obráceném pořadí než v původní struktuře.

```
procedure NajdiNejmenší;
var p, q: uHlavní;
begin          {vyhledání hlavních prvků s počtem = 0 předchůdců}
  p := čelo;
  čelo := nil;
  while p <> závěr do begin              {Prohledá až po zarážku}
     q := p;
     p := p^.další;
     if q^.počet = 0 then begin
       q^.další := čelo; čelo := q;
     end;
  end;
end;
```

Nyní zbývá vypsat prvky ve správném pořadí (nebo je zpracovat jiným způsobem). Procedura *VypišSeznam* vezme první prvek, vypíše ho a v následovnících, uvedených v pomocném seznamu, sníží počet předchůdců (složka *počet*). Jestliže v některém z následovníků klesne počet předchůdců na 0, přidá ho do seznamu nejmenších prvků.

```
procedure VypišSeznam;
var p, q: uHlavní;
    t:    uPomocný;
begin
  q := celo;{výstupní fáze}
```




Obr. 5.14: Vytvořený seznam

```
  while q <> nil do begin
    writeln(q^.klíč);
    dec(z);                              {Odpočti zpracovaný prvek}
    t := q^.upom;
    q := q^.další;
    while t <> nil do begin               {Probíráme následovníky}
      p := t^.id;
      dec(p^.počet);
      if p^.počet = 0 then begin {Už nemá předchůdce - přidáme ho}
        p^.další := q;          {do seznamu prvků bez předchůdců}
        q := p;
      end;
      t := t^.další;
    end;
  end;
  if z <> 0 then writeln('Tato množina není částečně uspořádaná');
end;
```

Jestliže v seznamu nenajdeme prvek bez předchůdců z přitom seznam není prázdný, nebyla množina částečně uspořádaná; graf relací obsahoval cyklus.

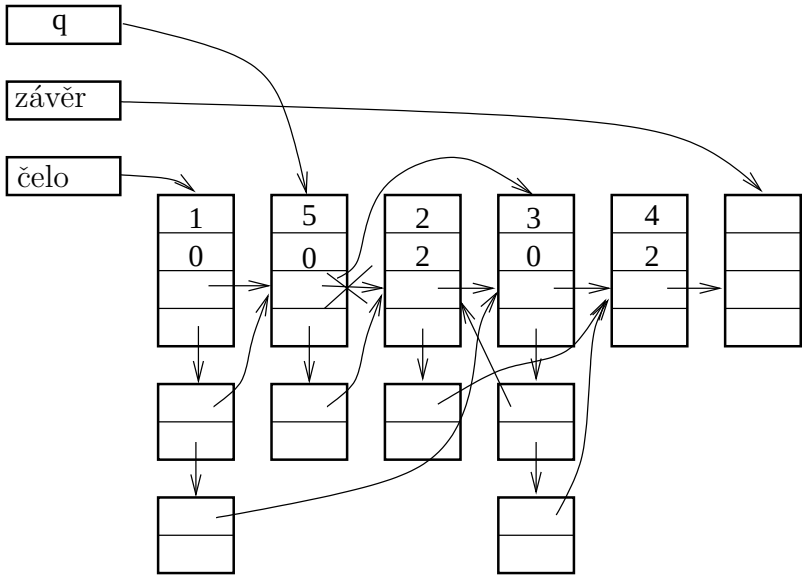
Poznamenejme, že v tomto jednoduchém programu jsme se nestarali o zrušení seznamu po skončení programu. Čtenář se může pokusit upravit jej tak, abychom mohli prvky po zpracování uvolnit pomocí procedury *dispose*, a to i v případě, že vstupní data nedefinovala částečné uspořádání.

### Příklad 5.14

Předpokládejme, že ve vstupním souboru jsou dvojice, pro přehlednost doplníme znak „◀“, vyznačující uspořádání:

$$1 \blacktriangleleft 5, 5 \blacktriangleleft 2, 1 \blacktriangleleft 3, 2 \blacktriangleleft 4, 3 \blacktriangleleft 2, 3 \blacktriangleleft 4.$$

Jde vlastně o uspořádání (14), dvojice jsou ale v jiném pořadí. Procedura *VytvořSeznam* vytvoří seznam, který vidíte na obr. 5.14 Procedura *NajdiNejmenší* tento seznam nezmění, neboť *čelo* ukazuje shodou okolností na jediný nejmenší prvek.
Procedura *VypišSeznam* vypíše hodnotu nejmenšího prvku, 1, a v jeho následovnících podle uspořádání sníží údaj o počtu předchůdců. Přitom zjistí, že prvky s klíči 5 a 3 už nemají žádného předchůdce a zařadí je do seznamu. Strukturu seznamu po této úpravě vidíte na obr. 5.15. Všimněte si, že prvek 2 je nyní dostupný pouze jako následovník prvku 5 nebo 3.




Obr. 5.15: Seznam po zpracování prvku 1

# Kapitola 6

# Použití binárního stromu

S binárním stromem a základními algoritmy pro práci s ním jsme se setkali již v kapitole 2.2.2. Zde si ukážeme některé další vlastnosti a použití této datové struktury.

## 6.1 Vyvážené stromy

Binární strom označíme za dokonale vyvážený, jestliže pro jeho libovolný vrchol v platí, že počet vrcholů v levém a pravém podstromu vrcholu v se liší nejvýše o 1.

Není obtížné sestrojit statický dokonale vyvážený strom, tedy strom, u něhož předem známe počet vrcholů a ten se nebude měnit. (Jde o úlohu podobnou konstrukci optimálního vyhledávacího stromu, se kterou se setkáme v 6.2.2.)

Značné problémy ovšem nastanou, jestliže se bude strom v průběhu své existence měnit, tj. jestliže do něj budeme chtít přidávat vrcholy nebo je rušit. Proto se obvykle používají slabší definice vyváženosti. Jednu z nich navrhli Adelson-Velskij a Landis [32] a stromy, vyvážené podle jejich definice, se (podle nich) označují jako *AVL-stromy* nebo jako *AVL-vyvážené stromy*. Protože žádnou jinou definici vyváženosti nebudeme používat, budeme o nich hovořit prostě jako o *vyvážených stromech*. Tedy:

*Strom je vyvážený, právě když se výšky obou podstromů, připojených k jeho libovolnému vrcholu, liší nejvýše o 1.*

Autoři AVL-stromů dokázali, že výška AVL-stromu nikdy nepřesáhne 1,45-násobek výšky dokonale vyváženého stromu, složeného ze stejných vrcholů. To znamená, že obvyklé operace, jako je přidání vrcholu do stromu, vyhledání nebo zrušení vrcholu lze provést v čase $O(\log_2 n)$, kde $n$ je počet vrcholů.

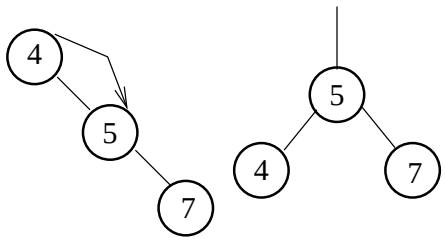
Vyhledávání ve vyváženém stromu se neliší od vyhledávání v „obyčejném“ binárním stromu. Při přidávání nebo rušení vrcholů se ovšem z vyváženého stromu může stát strom nevyvážený. Podíváme se, jak se napravuje rovnováha, porušená při přidávání vrcholů; postup při obnovení rovnováhy po zrušení vrcholu bude podobný.

### 6.1.1 Přidávání vrcholů do vyváženého stromu

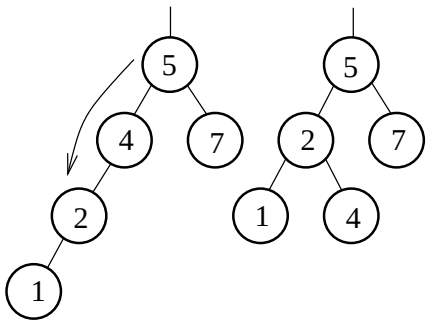
Máme tedy strom, který byl vyvážený, a do kterého jsme přidali jeden vrchol. Pro určitost předpokládejme, že jsme jej přidali levého podstromu. Jestliže si označíme $K$ kořen, $L$ resp. $P$ levý resp. pravý podstrom a $v_L$ resp. $v_P$ jejich výšky, bude před přidáním vrcholu platit jedna z následujících tří možností:

1. $v_L = v_P$. Po přidání bude $v_L$ o jedničku větší než $v_P$, nicméně strom zůstane vyvážený.
2. $v_L < v_P$. Po přidání bude $v_L = v_P$, strom zůstal vyvážený.
3. $v_L > v_P$. Po přidání bude $v_L = v_P + 2$, takže strom již není vyvářený - je třeba zjednat nápravu.

Při vyvažování musíme zachovat relativní pořadí vrcholů ve stromu, neboť to odpovídá pořadí klíčů. To znamená, že jediné přípustné zásahy budou rotace vrcholů, při kterých např. vrchol $L$ nahradí $K$, kořen $K$ přejde do pravého podstromu apod.




Obr. 6.1: Jednoduchá rotace $PP$



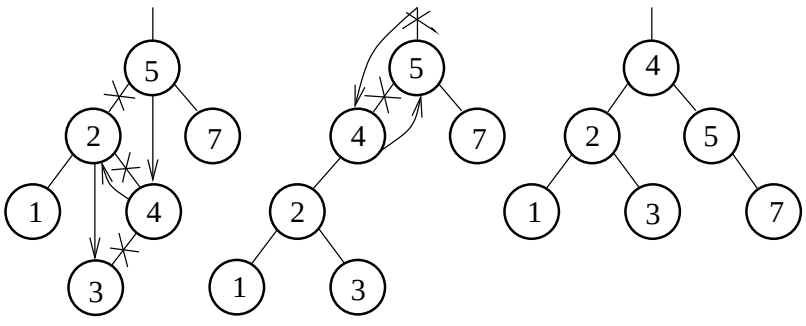

Obr. 6.2: Jednoduchá rotace $LL$

**Příklad 6.1**

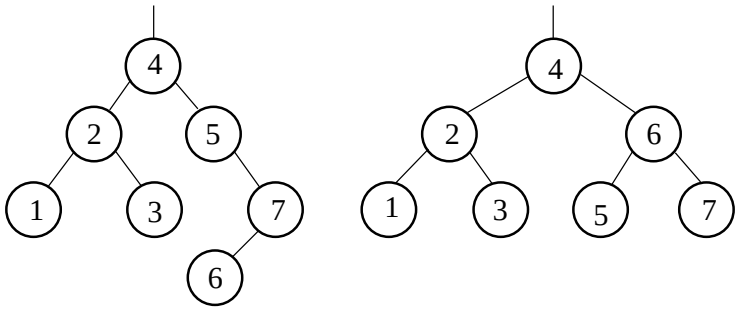
V tomto příkladu si ukážeme všechny situace, které mohou nastat (podle [1]). Uvažujme prázdný AVL-strom, do kterého budeme postupně přidávat vrcholy s hodnotami 4, 5, 7, 2, 1, 3, 6.

Po přidání vrcholu 7 vznikne poprvé nevyvážený strom, který vidíte na obr. 6.4 vlevo. Nápravy dosáhneme jednoduchou rotací vpravo, kterou označíme $PP$ (prodloužil se pravý podstrom pravého podstromu).
Po přidání vrcholu 2 zůstane strom vyvážený. Po přidání vrcholu 1 se rovnováha opět poruší (obr. 6.4), takže musíme použít jednoduchou rotaci doleva ($LL$, neboť se prodloužil se levý podstrom levého podstromu).
Po přidání vrcholu 3 poruší vyváženost kořene, tj. vrcholu 5. K nápravě nyní použijeme dvojitou rotaci, kterou označíme $LP$ (vyrostl levý podstrom pravého podstromu, obr. 6.4). Nejprve upravíme levý podstrom s kořenem 2 tak, aby se jeho kořenem stal vrchol 4. Pak posuneme kořen celého stromu - tím se stane 4.
Poslední přidaná hodnota je 6. Tentokrát se objeví nerovnováha v levém podstromu pravého podstromu, takže použijeme rotaci $PL$. Postup je zrcadlovým obrazem rotace $LP$ (obr. 6.4).
Při práci s vyváženým stromem budeme potřebovat údaje o tom, o kolik se liší výška levého a pravého podstromu. Proto přidáme do každého záznamu položku *vyváženost*, která bude obsahovat rozdíl výšky pravého a levého podstromu (v tomto pořadí). Ve složce *počet* budeme ukládat počet výskytů prvku s daným klíčem.

```
type uVrchol = ^vrchol;
     vrchol = record
       klíč: integer;
       počet:integer;
```




Obr. 6.3: Dvojitá rotace $LP$




Obr. 6.4: Dvojitá rotace $PL$

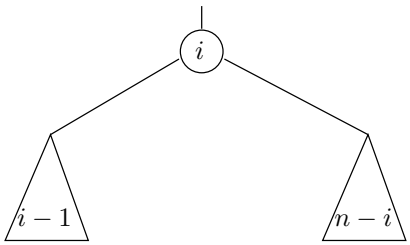
```
    levý, pravý: uVrchol;
    vyváženost: -1..1;
  end;
```

Při vkládání budeme potřebovat informaci o tom, zda se výška podstromu změnila. K tomu použijeme pomocný parametr $h$ typu *boolean*, který budeme předávat hodnotou. Jestliže přidáme prvek do levého podstromu kořene $K$ a výška tohoto podstromu se zvětší, bude potřebné strom znovu vyvážit, jestliže bude $vyvenost = -1$. Jestliže jsme přidávali do pravého podstromu a jeho výška se zvětšila, bude potřebné strom znovu vyvážit, bude-li $vyvenost = 1$.

Podle hodnoty *vyváženost* v kořeni podstromu pak určíme, kterou z rotací použijeme. V následující proceduře *vlož* jsou rotace vyjádřeny jako vnořené procedury $LL$, $LR$ atd.

```
procedure vlož(x: integer; var p: uVrchol; var v: boolean);
var p1, p2: uVrchol;              {na počátku musí být v = false}
procedure LL; {levý podstrom levého podstromu}
begin
  p^.levý := p1^.pravý;
  p1^.pravý := p;
  p^.vyváženost := 0;
  p := p1;
end; {LL}
procedure RR; {pravý podstrom pravého podstromu}
begin
  p^.pravý := p1^.levý;
  p1^.levý := p;
  p^.vyváženost := 0;
  p := p1;
end; {RR}
procedure LR; {levý podstrom pravého podstromu}
begin
  p2 := p1^.pravý;
  p1^.pravý := p2^.levý;
  p2^.levý := p1;
  p^.levý := p2^.pravý;
  p2^.pravý := p;
  if p2^.vyváženost = -1 then p^.vyváženost := 1
                         else p^.vyváženost := 0;
  if p2^.vyváženost = +1 then p1^.vyváženost := -1
                         else p1^.vyváženost := 0;
  p := p2;
end; {LRLLR}
procedure RL; {pravý podstrom levého podstromu}
begin
  p2 := p1^.levý;
  p1^.levý := p2^.pravý;
```

```
  p2^.pravý := p1;
  p^.pravý := p2^.levý;
  p2^.levý := p;
  if p2^.vyváženost = +1 then p^.vyváženost := -1
                         else p^.vyváženost := 0;
  if p2^.vyváženost = -1 then p1^.vyváženost := +1
                         else p1^.vyváženost := 0;
  p := p2;
end; {RL}
begin {vlož}
  if p=nil then begin             {klíč není ve stromu, přidá se}
    new(p);
    v := true;
    with p^ do begin
      klíč := x;
      počet := 1;
      levý := nil;
      pravý := nil;
      vyváženost := 0;
    end;
  end;
  else if x < p^.klíč then begin
    vlož(x, p^.levý,v);
    if v then {zvětšil se levý podstrom}
      case p^.vyváženost of
       1: begin
            dec(p^.vyváženost);
            v := false;
          end;
       0: dec(p^.vyváženost);
      -1: begin {vyvažování}
           p1 := p^.levý;
           if p1^.vyváženost = -1 then LL else LR;
           p^.vyváženost := 0;
           v := false;
          end;
    end; {case}
    end else
    if x > p^.klíč then begin
    vlož(x, p^.pravý,v);
    if v then                         {zvětšil se pravý podstrom}
      case p^.vyváženost of
      -1: begin
            inc(p^.vyváženost);
            v := false;
          end;
       0: inc(p^.vyváženost);
       1: begin                                      {vyvažování}
           p1 := p^.pravý;
           if p1^.vyváženost = +1 then RR else RL;
           p^.vyváženost := 0;
           v := false;
          end;
    end; {case}
    end else begin
     inc(p^.počet);
     v := false;
    end;
```




Obr. 6.5: Binární strom s náhodně zvoleným kořenem $i$

```
end;{vlož}
```

Je zřejmé, že operace s vyváženým stromem jsou podstatně složitější než operace s „obyčejným“ stromem. Proto se tyto stromy vyplatí budovat zejména tehdy, když vyhledávání převažuje nad přidáváním nebo rušením vrcholů. Také v případě, že se používá zhuštěného ukládání záznamů, přestávají být vyvážené stromy výhodné - operace přístupu ke zhuštěným záznamům mohou být velmi neefektivní.

## 6.2 Vyhledávání v binárním stromu

### 6.2.1 Analýza vyhledávání v binárním stromu

V předchozím oddílu jsme se setkali s vyváženými stromy. Protože práce s nimi nebývá efektivní, podíváme se na „obyčejné“ stromy. Nejprve ale definujeme dokonalý strom.

Binární strom s k úrovněmi označíme jako dokonalý, jestliže všechny vrcholy na úrovních $1, \ldots, k-1$ mají dva následovníky. Vrcholu na $k$-té úrovni jsou pochopitelně listy. Dokonalý strom má $n = 2^k - 1$ vrcholů - největší možný počet vrcholů ze všech binárních stromů s $k$ úrovněmi. Je to zvláštní případ dokonale vyváženého stromu.

Nejdelší cesta, kterou musíme při vyhledávání v dokonalém binárním stromě projít, má délku

$$k \approx \log_2 n.$$

Průměrná délka cesty v dokonalém stromě bude podle vztahu (2.1) z kap. 2.2.2.

$$s_n = \frac{1}{n} \sum_{i=1}^{k} i 2^{i-1} = \frac{k2^{k+1} - (k+1)\,2^k + 1}{2^k - 1} = \frac{(k-1)\,2^k + 1}{2^k - 1} \approx k - 1 \approx \log_2 n - 1 \tag{6.1}$$

Součet v (6.1) snadno vypočteme, jestliže si uvědomíme, že $\sum_{i=1}^{k} 2^{i-1}$ je hodnota funkce $\frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{i=1}^{k} x^i\right)$ v bodě $x = 2$. Průměrná délka cesty je tedy přibližně rovna dvojkovému logaritmu počtu uzlů. Na druhé straně dokonalé stromy se vyskytují opravdu zřídka. Strom roste dynamicky podle toho, jak v něm ukládáme údaje - a údaje mohou přicházet nesmírně chaoticky.
V nejhorším případě může mít každý vrchol pouze jediného následovníka, takže místo stromu vznikne lineární seznam. To se stane např. v případě, že budeme ukládat data seřazená podle velikosti. Potom bude k vyhledání údaje potřeba v nejhorším případě $n$, v průměrném případě $n/2$ porovnání (předpokládáme, že všechny položky se hledají stejně často).

Nejhorší případ, kdy vznikne místo stromu seznam, je ovšem podobně málo pravděpodobný jako nejlepší případ, kdy vznikne dokonalý strom. Podíváme se proto, jak vypadá délka cesty, tj. počet operací, v průměrném případě.
Vezmeme libovolnou permutaci množiny $\hat{n} = \{1, \ldots, n\}$ a vytvoříme z ní binární strom; protože permutací množiny $\hat{n}$ je $n!$, musíme uvažovat $n!$ stromů, které budou všechny stejně pravděpodobné.

Pravděpodobnost, že se kořenem stane číslo $i$, je rovna $1/n$. Potom bude mít levý podstrom $i-1$ vrcholů a pravý podstrom $n-i$ vrcholů (obr. 6.4).

Označíme $a_n$ hledanou průměrnou délku cesty ve stromu s $n$ vrcholy. Pravděpodobnosti přístupu k jednotlivým vrcholům pokládáme za stejné, rovné $1/n$. Průměrnou délku cesty pro $i$-tý vrchol můžeme počítat jako součet součinů úrovně vrcholu (tedy délky cesty) a pravděpodobnosti přístupu k němu, tj.

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \tag{6.2}$$

kde $p_i$ je délka cesty pro $i$-tý vrchol.

Vrcholy stromu, který vidíte na obr. 6.5, můžeme rozdělit na tři skupiny:

1. Vrcholy levého podstromu. Jejich průměrná délka cesty je $a_{i-1}+1$ (jednička navíc je za cestu do kořene celého stromu). V tomto stromu je $i-1$ vrcholů, to znamená, že pravděpodobnost přístupu k některému z vrcholů v tomto podstromu je $(i-1)/n$.
2. Kořen stromu. Délka cesty je 1, pravděpodobnost přístupu $1/n$.
3. Vrcholy pravého podstromu. Průměrná délka cesty pro $n-i$ vrcholů v tomto podstromu je $a_{n-i}+1$, pravděpodobnost přístupu k některému z vrcholů v tomto podstromu je $(n-i)/n$.

To znamená, že průměrnou délku cesty (při pevně zvoleném $i$) můžeme vyjádřit jako součet tří členů:

$$a_n^{(i)} = (a_{i-1}+1)\,\frac{i-1}{n} + 1\frac{1}{n} + (a_{n-i}+1)\,\frac{n-i}{n} \tag{6.3}$$

Celkovou průměrnou délku $a_n$ potom můžeme vyjádřit jako průměr z hodnot $a_n^{(i)}$ přes všechna $i$, $i = 1,\ldots,n$, to znamená přes všechny stromy s hodnotami $1,2,\ldots,n$ v kořeni. Dostaneme

$$a_n = \frac{1}{n}\sum_{i=1}^{n} a_n^{(i)} = \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\,\frac{i-1}{n} + \frac{1}{n} + (a_{n-i}+1)\,\frac{n-i}{n}\right] \tag{6.4}$$

V hranatých závorkách můžeme vytknout $1/n$ a sečíst výrazy, které nezávisí na $a_k$. Tak dostaneme

$$a_n = 1 + \frac{1}{n^2}\sum_{i=1}^{n}\left[(i-1)\,a_{i-1} + (n-i)\,a_{n-i}\right] = 1 + \frac{2}{n^2}\sum_{i=1}^{n}(i-1)\,a_{i-1} = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i a_i. \tag{6.5}$$

V (6.5) jsme počítali dva stejné součty v opačném pořadí, proto jsme je nahradili jediným součtem. Tak jsme dospěli k vyjádření tvaru $a_n = f(a_{n-1}, a_{n-2}, \ldots, a_1)$, které se ale pro výpočet $a_n$ příliš nehodí. Pokusíme se z něj získat vztah tvaru $a_n = g(a_{n-1})$.

Z (6.5) plyne

$$a_n = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i a_i = 1 + \frac{2}{n^2}(n-1)\,a_{n-1} + \frac{2}{n^2}\sum_{i=0}^{n-2} i a_i, \tag{6.6}$$

$$a_{n-1} = 1 + \frac{2}{(n-1)^2}\sum_{i=0}^{n-2} i a_i. \tag{6.7}$$

Vyjádříme-li ze vztahu (6.7) součet $\sum_{i=0}^{n-2} i a_i$ a dosadíme-li za něj do (6.6), dostaneme

$$a_n = \frac{1}{n^2}\left(\left(n^2-1\right)a_{n-1} + 2n - 1\right) \tag{6.8}$$

Vztah (6.8) představuje lineární diferenční rovnici. Použijeme obvyklého postupu řešení. Nejprve vyřešíme rovnici bez pravé strany,

$$b_n = \frac{\left(n^2-1\right)}{n^2} b_{n-1} \tag{6.9}$$

Položíme-li $b_1 = \beta$, odvodíme snadno matematickou indukcí, že

$$b_2 = \beta\frac{2^2-1}{2^2}, \qquad b_3 = \beta\frac{2^2-1}{2^2}\frac{3^2-1}{3^2}, \ldots, b_n = \beta\frac{2n-1}{2n}$$

Dále potřebujeme najít jakékoli řešení rovnice s pravou stranou (6.8). Přitom použijeme analogii metody variace konstant, známé z teorie diferenciálních rovnic. Předpokládáme, že řešení má tvar $a_n = K_n b_n = K_n(n+1)/2n$ a po dosazení do (6.8) a úpravě dostaneme pro posloupnost $K_n$ rekurentní vztah (diferenční rovnici)

$$K_n - K_{n-1} = \frac{3}{n+1} - \frac{1}{n}$$

Snadno se přesvědčíme, že řešení této rovnice lze má tvar $3H_{n+1} - H_n$, kde $H_n$ je posloupnost

$$H_n = \sum_{i=1}^{n}\frac{1}{i}$$

Z toho pak plyne, že řešení rovnice (6.8) má tvar

$$a_n = \beta b_n + K_n b_n = \beta \frac{n+1}{n} + \frac{n+1}{n}\left(3H_{n+1} - H_n\right) \tag{6.10}$$

Protože posloupnost $H_n$ se pro velká $n$ chová přibližně jako přirozený logaritmus $n$, bude pro poměr cesty dokonalého stromu $s_n$ a střední hodnoty cesty náhodného stromu $a_n$ platit

$$\lim_{n\to\infty} \frac{a_n}{s_n} = \lim_{n\to\infty} \frac{2\ln n}{\log_2 n} = 2\ln 2 \approx 1,39.$$

Průměrné zhoršení tedy není příliš výrazné, představuje pouhých 39%. Přesto v případě algoritmů, u kterých je čas kritický (např. při řízení procesů v reálném čase) může být nepříjemné.

### 6.2.2 Binární vyhledávací stromy

#### Úvodní úvahy

V předchozím odstavci jsme vycházeli z předpokladu, že přístup ke všem vrcholům je stejně pravděpodobný. To ale nemusí být vždy pravda. Někdy se setkáme se stromy, které se v průběhu programu nemění a které slouží k rychlému vyhledávání informací.

Například při překladu programu narážíme ve zdrojovém textu identifikátory a potřebuje zjišťovat, zda to jsou klíčová slova. Uspořádáme proto všechna klíčová slova do binárního stromu a každý nalezený identifikátor budeme nejprve hledat v tomto stromu. Pokud v něm není, nejde o klíčové slovo.

Takovéto stromy budeme označovat jako vyhledávací. Přesná definice vyhledávacího stromu vypadá takto:

*Binární vyhledávací strom* ($BVS$) je binární strom, pro který platí:

1. Všechny položky, uložené v levém podstromu, jsou menší než položka v kořeni.
2. Všechny položky, uložené v pravém podstromu, jsou větší než položka v kořeni.
3. Levý a pravý podstrom jsou opět binární vyhledávací stromy.

Vzhledem k tomu, že se $BVS$ používají nejčastěji v kompilátorech, budeme o položkách, uložených ve vyhledávacích stromech, dále hovořit jako o „identifikátorech“. Ukážeme si, jak sestrojit optimální $BVS$ za předpokladu, že známe pravděpodobnosti použití identifikátorů (klíčových slov), která v něm budou uložena. Přitom ale musíme znát i pravděpodobnosti neúspěšného vyhledávání, tj. pravděpodobnosti vyhledávání identifikátoru, který ve stromě není.

V $BVS$ budou uloženy identifikátory $a_1, a_2, \ldots, a_n$ pro které platí uspořádání $a_1 < a_2 < \ldots < a_n$. Pro úplnost ještě definujeme $a_0 = -\infty$, $a_{n+1} = +\infty$. Předpokládáme, že pravděpodobnosti, že identifikátor $x$ je roven jedné z hodnot uložených v $BVS$, jsou

$$P_i = P\left(x = a_i\right)$$

Identifikátory, které ve stromě nejsou, se rozdělí na třídy ekvivalence $E_i$,

$$E_i = \left\{x \,|a_i < x < a_{i+1}\right\}, \qquad i = 0, 1, \ldots, n$$

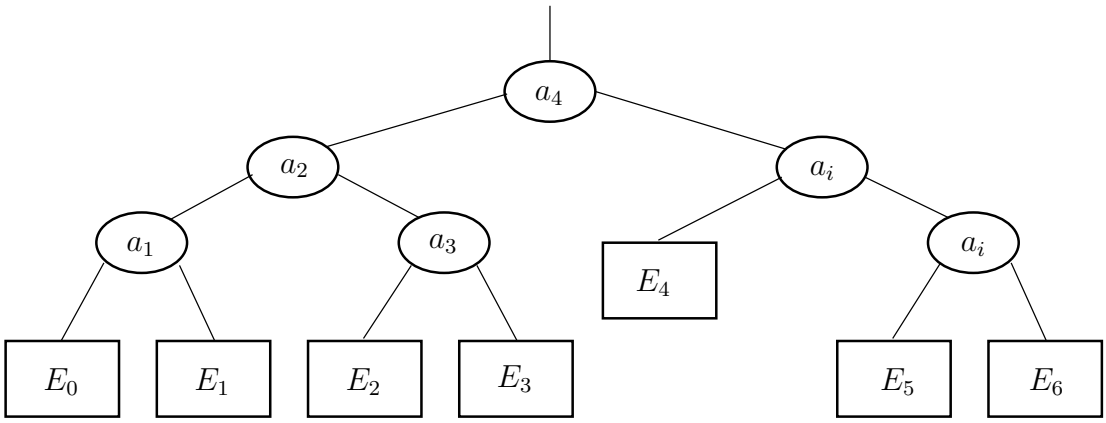
V dalších úvahách budeme muset také vzít v potaz pravděpodobnosti $Q_i$ neúspěšného vyhledávání, tj. pravděpodobnosti, že pro identifikátor $x$ bude platit $a_i < x < a_{i+1}$:

$$Q_i = P\left(x \in E_i\right), \qquad i = 0, 1, \ldots n$$

Identifikátor $x$ v $BVS$ buď najdeme (pak je $x = a_i$ pro nějaké $i$) nebo nenajdeme, a pak pro nějaké $i$ platí $a_i < x < a_{i+1}$. Jiná možnost není, a proto musí pro součet pravděpodobností $P_i$ a $Q_i$ platit

$$\sum_{i=1}^{n} \left(P_i + Q_i\right) + Q_0 = 1.$$

Poznamenejme, že $BVS$ můžeme chápat jako binární strom, ve kterém neúspěšná vyhledávání končí ve zvláštních vrcholech, jak jsme je zavedli v odst. 2.2.2. (obr. 6.6).




Obr. 6.6: Binární vyhledávací strom

### Vyhledávací procedura a cena stromu

Je-li *BVS* složen z vrcholů typu *vrchol*, a data, uložená ve vrcholech *BVS*, jsou typu *data*, můžeme pro vyhledávání použít funkci (napíšeme ji pro změnu v jazyku C)

```
typedef vrchol *uVrchol;
uVrchol zjisti(uVrchol T, data X)
{
uVrchol i;
    i = T;                  //d je složka vrcholu obsahující data
    while(i != NULL)
      if (X < i->d) i = i->Levy; else
      if (X > i->d) i = i->Pravy; else
      return i;
    return i;
}
```

Tato funkce vrátí buď adresu vrcholu, ve kterém je uložena hodnota $X$, nebo NULL, jestliže $X$ ve stromě není. Je-li vyhledávání úspěšné, tj. jestliže se hodnota $X$ najde, skončíme v normálním vrcholu stromu na $l$-té úrovni. To znamená, že proběhlo $l$ iterací cyklu while v proceduře *zjisti*. Je-li vyhledávání neúspěšné, tj. skončíme-li ve zvláštním uzlu stromu na úrovni $l$, proběhlo pouze $l-1$ iterací cyklu while.

To nám umožňuje definovat cenovou funkci pro daný strom S:

$$C(S) = \sum_{i=1}^{n} P_i u(a_i) + \sum_{i=0}^{n} Q_i (u(E_i) - 1), \tag{6.11}$$

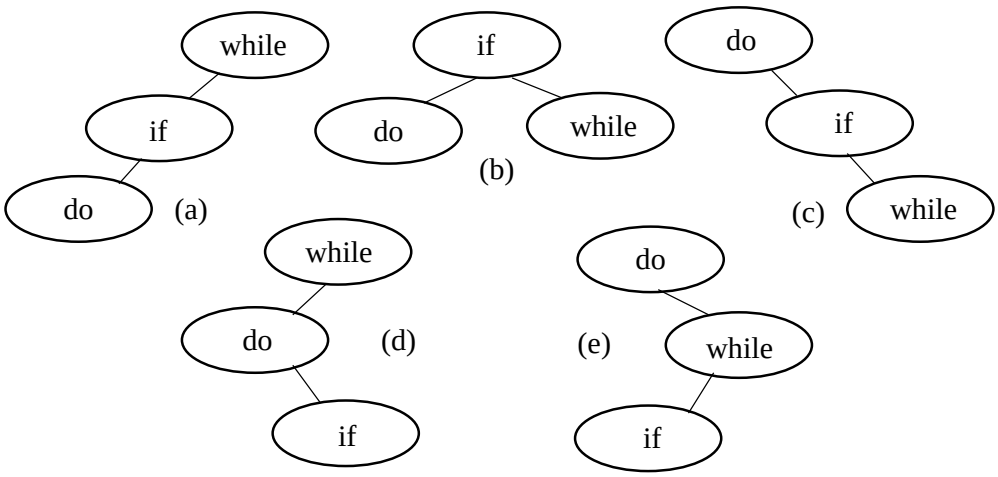
kde $u(a_i)$ je úroveň vrcholu $a_i$; připomeňme si, že kořen je na úrovni 1.

### Příklad 6.2

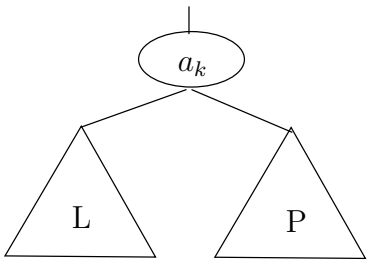
Uvažujme jednoduchý programovací jazyk, který obsahuje pouze tři klíčová slova **if**, **do** a **while**. Tato klíčová slova můžeme uspořádat do pěti vyhledávacích stromů, které vidíte na obrázcích 6.7
Jestliže bude $P_i = Q_i = 1/7$, bude $C(b) = 13/7$, ceny ostatních stromů budou $15/7$. Jak jsme mohli očekávat, nejvýhodnější je v tomto případě pravidelný strom (b). Bude-li však $P_1 = 0,5$, $P_2 = 0,1$, $P_3 = 0,05$, $Q_0 = 0,15$, $Q_1 = 0,1$, $Q_2 = Q_3 = 0,05$, bude $C(a) = 2,65$, $C(b) = 1,9$, $C(c) = 1,5$, $C(d) = 2,05$, $C(e) = 1,6$. V tomto případě je nejvýhodnější strom, který má tvar seznamu!

### Optimální vyhledávací strom

Na konstrukci optimálního vyhledávacího stromu se můžeme dívat jako na problém dynamického programování: prvním rozhodnutím určíme, který vrchol bude kořenem (přesněji který údaj uložíme do kořene). Nechť je to $a_k$.




Obr. 6.7: Možné binární vyhledávací stromy pro daná klíčová slova




Obr. 6.8: Volba kořene

Tím, že jsme zvolili kořen, se strom rozpadl na tři části: kořen a dva podstromy. Cena levého resp. pravého podstromu bude

$$C(L) = \sum_{i=1}^{k-1} P_i u(a_i) + \sum_{i=0}^{k-1} Q_i (u(E_i) - 1), \qquad C(P) = \sum_{i=k+1}^{n} P_i u(a_i) + \sum_{i=k}^{n} Q_i (u(E_i) - 1). \tag{6.12}$$

Tvar tohoto stromu znázorňuje obr. 6.8.

Nyní určíme cenu celého stromu. S pravděpodobností $P_k$ zůstaneme v kořeni $a_k$. Jinak můžeme jít do levého podstromu - pak bude cena součtem ceny levého podstromu a ceny za prřchod z kořene do něj, nebo můžeme jít do pravého podstromu - a cena bude součtem ceny pravého podstromu a ceny za průchod do něj.
Cena za průchod do levého resp. pravého podstromu je rovna součinu délky cesty (1 krok) a pravděpodobnosti, že cílový vrchol leží v tomto podstromu, tedy

$$Q_0 + \sum_{i=1}^{k-1} (P_i + Q_i) = W(0, k-1),$$

resp.

$$Q_k + \sum_{i=k+1}^{n} (P_i + Q_i) = W(k, n).$$

Obecně definujeme funkci $W(i, j)$ vztahem

$$W(i, j) = Q_i + \sum_{l=i+1}^{j} (P_l + Q_l).$$

Cenu celého stromu S tedy můžeme vyjádřit rovnicí

$$C(S) = P_k + W(0, k-1) + W(k, n) + C(L) + C(P). \tag{6.13}$$

*Naším cílem je zkonstruovat BVS, pro který bude $C(S)$ minimální.* Protože volbou kořene $a_k$ je pevně dáno rozdělení vrcholů do levého a pravého podstromu, je jasné, že i $C(L)$ a $C(P)$ musí být minimální. (Platí tedy princip optimality.)

Jestliže se nám podaří zvolit vhodně $a_k$, stojíme před podobným úkolem: určit kořeny levého a pravého podstromu. Opakováním tohoto postupu dospějeme nakonec k podstromům složeným z jediného uzlu.

Abychom mohli tuto minimalizační úlohu vyřešit, definujeme $C(i,j)$ jako cenu optimálního podstromu $S_{ij}$, složeného z uzlů $a_{i+1},\ldots,a_j$ a $E_i,\ldots,E_j$. Odtud pak plyne pro cenu celého stromu $S = S_{0n}$

$$C(0,n) = \min_{l<k\leq n} = \{C(0,k-1) + C(k,n) + P_k + W(0,k-1) + W(k,n)\}. \tag{6.14}$$

Obecně pro podstrom $S_{ij}$ vzhledem k definici $W(i,j)$ platí

$$\begin{aligned} C(i,j) &= \min_{i<k\leq j} \{C(i,k-1) + C(k,j) + P_k + W(i,k-1) + W(k,j)\} = &(6.15)\\ &= \min_{i<k\leq j} \{C(i,k-1) + C(k,j)\} + W(i,j) &(6.16)\end{aligned}$$

Rovnici (6.16) lze řešit pro $C(0,n)$ tak, že nejprve spočteme všechna $C(i,j)$ pro $i-j=1$, pak pro $i-j=2$ atd. To znamená, že nejprve sestrojíme všechny optimální stromy z jednoho vrcholu, pak optimální stromy ze dvou vrcholů atd.

**Příklad 6.3**

Sestrojíme optimální binární vyhledávací strom pro množinu klíčových slov $a_i$ ={**do**, **if**, **real**, **while**} (je tedy $n=4$). Přitom víme, že pravděpodobnosti $P_i$ jsou 3, 3, 1, 1 a pravděpodobnosti $Q_i$ jsou 2, 3, 1, 1, 1 (pro pohodlí jsme všechny pravděpodobnosti vynásobili 16). Označíme $R(i,j)$ vrchol $S_{ij}$.

Z definice $W(i,j)$ plyne, že $W(i,i) = Q_i$ pro libovolné $i = 0,\ldots,n$. Dále pak $C(i,i) = 0$ pro libovolné $i$, neboť strom $S_{ii}$ neobsahuje žádný vrchol. Dále se budeme řídit vztahem (6.16) a vezmeme v úvahu, že

$$W(i,j) = P_j + Q_j + W(i,j-1).$$

Pro stromy složené z jednoho vrcholu rovnici (6.16) vlastně ani nepoužijeme. Dostaneme postupně

$$\begin{aligned} W(0,1) &= P_1 + Q_1 + W(0,0) = 8,\\ C(0,1) &= W(0,1) + \min\{C(0,0) + C(1,1)\} = 8\\ R(0,1) &= 1,\end{aligned}$$

$$\begin{aligned} W(1,2) &= P_2 + Q_2 + W(1,1) = 7,\\ C(1,2) &= W(1,2) + \min\{C(1,1) + C(2,2)\} = 7\\ R(1,2) &= 2,\end{aligned}$$

$$\begin{aligned} W(2,3) &= P_3 + Q_3 + W(2,2) = 3,\\ C(2,3) &= W(2,3) + \min\{C(2,2) + C(3,3)\} = 3\\ R(2,3) &= 3,\end{aligned}$$

$$\begin{aligned} W(3,4) &= P_4 + Q_4 + W(3,3) = 3,\\ C(3,4) &= W(3,4) + \min\{C(3,3) + C(4,4)\} = 3\\ R(3,4) &= 4.\end{aligned}$$

Jestliže nyní známe $W(i,i+1)$ a $C(i,i+1)$ pro $i = 0,\ldots,3$, můžeme použít rovnici (6.16) a vypočítat $W(i,i+2)$, $C(i,i+2)$ a $R(i,i+2)$ pro $i = m,\ldots,2$. Pro stromy složené ze dvou vrcholů dostaneme

$$\begin{aligned} W(0,2) &= P_2 + Q_2 + W(0,1) = 12,\\ C(0,2) &= W(0,2) + \min\{C(0,k-1) + C(k,2)\} = 19 \qquad (\text{minimum přes } k=1,2)\\ R(0,2) &= 1, \qquad (\textit{pro cenově min. vrchol } k).\end{aligned}$$

$$\begin{aligned} W(1,3) &= P_3 + Q_3 + W(1,3) = 9, \\ C(1,3) &= W(1,3) + \min\{C(1,k-1) + C(k,3)\} = 12 \qquad \text{(minimum přes } k = 2,3) \\ R(1,3) &= 2, \end{aligned}$$

$$\begin{aligned} W(2,4) &= P_4 + Q_4 + W(0,1) = 5, \\ C(2,4) &= W(0,2) + \min\{C(2,k-1) + C(k,4)\} = 8 \qquad \text{(minimum přes } k = 3,4) \\ R(2,4) &= 3. \end{aligned}$$

Podobně pro stromy, složené ze tří vrcholů, dostaneme

$$\begin{aligned} W(0,3) &= P_3 + Q_3 + W(0,2) = 14, \\ C(0,3) &= W(0,3) + \min\{C(0,k-1) + C(k,3)\} = 25 \qquad \text{(minimum přes } k = 1,2,3) \\ R(0,3) &= 2, \end{aligned}$$

$$\begin{aligned} W(1,4) &= P_4 + Q_4 + W(1,3) = 11, \\ C(1,4) &= W(1,4) + \min\{C(1,k-1) + C(k,4)\} = 19 \qquad \text{(minimum přes } k = 2,3,4) \\ R(1,4) &= 2. \end{aligned}$$

Nyní již zbývá určit pouze optimální vrchol celého stromu:

$$\begin{aligned} W(0,4) &= P_4 + Q_4 + W(0,3) = 16, \\ C(0,4) &= W(0,4) + \min\{C(0,k-1) + C(k,4)\} = 32 \qquad \text{(minimum přes } k = 1,2,3,4) \\ R(0,4) &= 2. \end{aligned}$$

Odtud můžeme vyčíst, že vrcholem stromu bude položka $a_2$; v levém podstromu je jedině vrchol $a_1$, takže není o čem přemýšlet. V pravém podstromu zbyly vrcholy $a_3$ a $a_4$. Strom $S_{24}$ má optimální kořen $R_{24}$ a to je - jak jsme spočítali - $a_3$. Tím je optimální strom plně určen.

## 6.3 Zpracování aritmetického výrazu

V tomto odstavci se budeme pro jednoduchost zabývat pouze výrazy s binárními operátory.

### 6.3.1 Zápis výrazu pomocí stromu

Binární strom umožňuje popsat strukturu aritmetického výrazu, aniž bychom potřebovali znát priority operátorů nebo používat závorky. Podívejme se na výraz

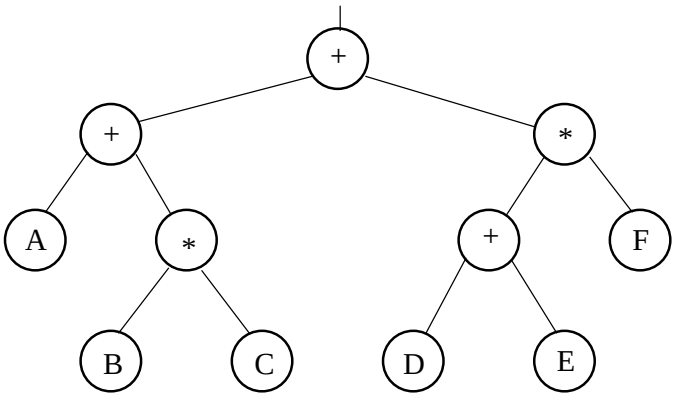
$$A + B * C + (D + E) * F. \tag{6.17}$$

Chceme-li ho vyhodnotit na základě tohoto zápisu, musíme vědět, že násobení má přednost před sčítáním a že výrazy v závorkách se vyhodnocují přednostně. Výraz (6.17) můžeme ovšem také znázornit binárním stromem, ve kterém budou vnitřní uzly představovat operátory, listy budou znamenat operandy (proměnné nebo konstanty). Příklad vidíme na obr. 6.9.
Postup pro vyčíslení hodnoty, kterou tento strom reprezentuje, definujeme výrazem

$$L \text{ op } P,$$

kde $L$ je hodnota levého podstromu, $P$ je hodnota pravého podstromu a op je operátor v kořeni stromu. Obsahuje-li některý podstrom pouze jediný list, je jeho hodnota rovna hodnotě konstanty (proměnné), kterou tento list určuje. Pokud je tento podstrom složitější, použijeme k výpočtu jeho hodnoty rekurzivně týž postup. Při uvedeném postupu je strom na obr. 6.9 ekvivalentní výrazu (6.17).

Popsaný algoritmus můžeme zapsat ve tvaru funkce




Obr. 6.9: Výraz reprezentovaný pomocí binárního stromu

```
function hodnota(t: ^strom): číslo;
begin
    if list(t) then hodnota := t^.data
    else
      hodnota := vyčísli(hodnota(t^.levý), t^.operátor,
                         hodnota(t^pravý));
end;
```

Přitom se odvoláváme na funkci *vyčísli*, která provede s prvním a třetím parametrem operaci, určenou druhým parametrem (operátorem).

### 6.3.2 Obrácený polský zápis

Binární strom lze uložit do pole. K tomu lze použít jak přímé tak zpětné zpracování. Budeme-li předpokládat, že všechny údaje ve stromě (operátory, proměnné) jsou stejného typu, mohli bychom k zápisu použít následující proceduru:

```
procedure ulož(t: uStrom);
begin
    if t<> nil then begin
        if list(t) then zapiš(t^.data) else zapiš(t^.operator);
        ulož(t^.levý);
        ulož(t^.pravý);
    end;
end;
```

Ze stromu z obr. 6.9 vznikne pomocí této procedury

$$++A*BC*+DEF \tag{6.18}$$

Toto vyjádření označujeme jako polský zápis[1]. Jsou-li všechny operátory binární, je tvar (6.18) ekvivalentní funkcionálnímu zápisu

$$+\left(+\left(A,*\left(BC\right)\right),*\left(+\left(D,E\right),F\right)\right) \tag{6.19}$$

ve kterém operátory vystupují jako funkce se dvěma argumenty.

Pro počítačové zpracování se občas používá *obrácený polský zápis* (*RPZ*), který vznikne, budeme-li zapisovat operátor až za operandy (zpětné zpracování). Výraz (6.17), přepsaný do RPZ, má tvar

$$ABC*DE+F*+ \tag{6.20}$$

Ukážeme si jednoduchý algoritmus, který umožňuje převést výraz do RPZ. Tento algoritmus předpokládá, že výraz má tvar řetězce položek, že všechny operátory jsou binární a že každý má operátor přiřazenu kladnou prioritu. Výstupem je opět řetězec položek ve tvaru RPZ. Při převodu využívá zásobník.

[1]Podle polského filozofa Lukasiewicze, který si zřejmě jako první uvědomil, že operátor představuje funkci svých operandů.

**Algoritmus**

1. Vezmi další položku ve vstupním řetězci.
2. Je-li to operand, vlož jej do výstupního řetězce.
3. Je-li to levá závorka „(“, vlož ji do zásobníku s prioritou 0.
4. Je-li to operátor, porovnej jeho prioritu s prioritou operátoru na vrcholu zásobníku:
   (a) Je-li jeho priorita větší, vlož ho do zásobníku;
   (b) Jinak vezmi operátor z vrcholu zásobníku a dej ho do výstupního řetězce; porovnání opakuj s operátorem, který je nyní na vrcholu zásobníku, dokud nebude priorita nového operátoru větší nebo dokud nebude zásobník prázdný. Pak vlož nový operátor do zásobníku.
5. Je-li to pravá závorka „)“, vyber ze zásobníku všechny operátory až po odpovídající levou závorku a dej je do výstupního řetězce; obě závorky zahoď (nepatří do výstupního řetězce).
6. Narazíš-li na konec výrazu, vyber ze zásobníku všechny zbývající operátory a dej je do výstupního řetězce, jinak se vrať na krok 1.

Tento algoritmus má jednu drobnou nevýhodu: prohodí pořadí operandů. To musíme vzít v úvahu u nesymetrických operátorů („-“, mocnina) .

**Příklad 6.4**

Podívejme se, jak by vypadal převod výrazu (6.17) do RPZ pomocí popsaného algoritmu.

| Vstup | Výstup | Zásobník |
|---:|---|:---:|
| A+B*C+(D+E)*F | ∅ | ∅ |
| +B*C+(D+E)*F | A | ∅ |
| B*C+(D+E)*F | A | + |
| *C+(D+E)*F | AB | + |
| C+(D+E)*F | AB | +* |
| +(D+E)*F | ABC | +* |
| (D+E)*F | ABC*+ | + |
| D+E)*F | ABC*+ | +( |
| +E)*F | ABC*+D | (+ |
| E)*F | ABC*+D | +(+ |
| )*F | ABC*+DE | +(+ |
| *F | ABC*+DE+ | + |
| F | ABC*+DE+ | +* |
| ∅ | ABC*+DE+F | +* |
| | ABC*+DE+F*+ | ∅ |

# Kapitola 7

# Seminumerické algoritmy

V této kapitole se podíváme na algoritmy pro základní početní operace, jako je sčítání nebo násobení celých nebo reálných čísel. Zabývat se těmito algoritmy se může zdát zbytečné: v aritmetické jednotce procesoru jsou již implementovány a uživatel je nemůže měnit.

Mohou se ovšem vyskytnout situace, kdy znalost těchto základních algoritmů oceníme. Může se např. stát, že nám nebude stačit přesnost, kterou nám nabízí procesor našeho počítače (počítání s velmi dlouhými čísly je běžné např. při šifrování). Potom nezbývá, než si základní operace naprogramovat sami.

Vedle toho znalost základních algoritmů umožní lépe porozumět možnostem a omezením počítače.

## 7.1 Poziční číselné soustavy

Algoritmy pro aritmetické operace, tedy vlastně pro veškeré zacházení s čísly, jsou bezprostředně závislé na způsobu, jakým jsou číselné hodnoty v počítači zobrazovány. Způsob zobrazení čísel se nazývá *číselná soustava*; naše povídání tedy začne výkladem o číselných soustavách.

V poziční číselné soustavě o základu $z$ představuje zápis $(\ldots a_3a_2a_1a_0, a_{-1}a_{-2}a_{-3}\ldots)_z$ reálné číslo s hodnotou

$$(\ldots a_3a_2a_1a_0, a_{-1}a_{-2}a_{-3}\ldots)_z = \cdots + a_3z^3 + a_2z^2 + a_1z^1 + a_0z^0 + a_{-1}z^{-1} + a_{-2}z^{-2} + a_{-3}z^{-3} + \cdots \quad (7.1)$$

kde $a_i$ jsou číslice, $a_i \leq z$.

Poziční[1] číselná soustava se základem $z$ používá k vyjádření z různých znaků, označovaných jako *číslice*. V číselných soustavách se základem $z \leq 10$ se používají číslice $0, 1, \ldots, 9$. Je-li základ $z \leq 36$, používají se jako další číslice písmena latinské (přesněji anglické) abecedy. Písmeno $a$ bude hrát roli číslice s významem 10, $b$ bude představovat číslici 11 atd.

Osmičková soustava používá pouze číslic $0, \ldots, 7$. Zápis $(120, 3)_8$ představuje číslo

$$(120, 3)_8 = 1.8^2 + 2.8 + 0.1 + 3.8^{-1} = 80 + \frac{3}{8} = (80, 375)_{10}\,.$$

Šestnáctková soustava se skládá z číslic $0, 1, \ldots, 9, a, b, c, d, e, f$. Zápis $(1f)_{16}$ představuje číslo

$$(1f)_{16} = 1.16 + 15 = 31.$$

Číslici $a_k$ označujeme jako *významnější* než číslici $a_l$, je-li $k > l$.

## 7.2 Celá čísla

Podíváme se nyní na problémy, které souvisí s reprezentací celých čísel v soustavě se základem z a s algoritmy pro základní aritmetické operace v počítači, který pracuje s $n$ pozicemi.

[1] Nyní je již zřejmé, proč se tyto soustavy označují jako *poziční*: Váha jednotlivých číslic je určena jejich pozicí vzhledem k desetinné čárce (nebo tečce). Existují i nepoziční číselné soustavy - např. římská. Pro poziční číselnou soustavu se základem $z$ se používá označení $z$-ární soustava.

### 7.2.1 Reprezentace celých čísel se znaménkem

Jak zobrazíme záporná čísla, jestliže máme k dispozici $n$ pozic? Jedno z možných řešení je přidat další pozici, která bude obsahovat znaménko.

Uvažujme počítač, který pracuje v desítkové soustavě a má 10 pozic na číslice a jednu pozici na znaménko. To znamená, že číslo 123 se bude zobrazovat +0000000123, číslo −123 se zobrazí jako −0000000123.

Tato *znaménková reprezentace* (označovaná také jako *přímý kód*) v podstatě odpovídá běžným konvencím zápisu celých čísel. Její možnou nevýhodou je, že čísla +0 a −0 představují tutéž hodnotu 0, což může vést k určitým potížím.

Proto se zpravidla používá doplňková reprezentace (doplňkový kód). Je-li dána konstanta $k$, definujeme v soustavě se základem $z$ obraz čísla $x$, $|x| < z^k$, takto:

- Obrazem celého nezáporného čísla $x < z^k$ je samo číslo $x$, zapsané v $z$-ární soustavě (a případně doplněné zleva nulami).
- Obrazem záporného celého čísla $x > -z^k$ je číslo $z^k + x$.

Doplňková reprezentace nepoužívá znaménka. Proto se zde nesetkáme s problémem kladné a záporné nuly; na druhé straně všechny ostatní obrazy mohou mít dva významy, mohou představovat buď kladné nebo záporné číslo. Proto se obvykle užívá ostřejší definice doplňkového kódu:

- Obrazem celého nezáporného čísla $x < z^k/2$ je samo číslo $x$, zapsané v $z$-ární soustavě (a případně doplněné zleva nulami).
- Obrazem záporného celého čísla $x \geq z^k/2$ je číslo $z^k + x$.

Použijeme-li této definice, můžeme podle hodnoty první číslice obrazu rozhodnout, zda jde o obraz kladného nebo záporného čísla.

**Příklad 7.1**

Klasické mechanické kalkulačky pracovaly v doplňkovém kódu v desítkové soustavě. Představme si, že máme kalkulačku s 10 pozicemi. Jestliže na ní od čísla 0 odečteme 1, dostaneme 9999999999, neboť aritmetické operace se provádějí modulo $10^{10}$. Stejný výsledek ovšem můžeme dostat i sečtením obrazů kladných čísel, např.

$$4444444444 + 4444444444 + 1111111111 = 9999999999.$$

Obvykle se ovšem čísla (obrazy), která mají na nejvyšší pozici $5, \ldots, 9$ interpretují jako záporná.

Dnešní počítače používají obvykle dvojkovou (binární) soustavu. Zpravidla si můžeme předepsat, zda budeme obsah daného čísla chápat jako celé číslo bez znaménka, tedy nezáporné číslo v přímém kódu, nebo jako celé číslo se znaménkem v doplňkovém kódu.

V Turbo Pascalu na PC máme např. typ *byte*, což je jeden byte, chápaný jako celé číslo bez znaménka. Může tedy obsahovat celá čísla v rozmezí 0..255. Vedle toho máme typ *shortint*, což je opět jeden byte, tentokrát ale chápaný jako celé číslo v doplňkovém kódu. To znamená, že může obsahovat celá čísla v rozmezí -128..127.

Podobné dvojice tvoří dvoubytové typy *word* a *integer*. V jazyku C na PC najdeme ještě čtyřbytové typy **long** a **unsigned long**; Turbo Pascal nabízí typ čtyřbytový typ se znaménkem *longint*, z nějakých záhadných důvodů nám však odepírá analogický typ bez znaménka.

Podívejme se nyní na vztah číselné soustavy ze základem $z$ a soustavy se základem $Z = z^n$. Označíme-li $a_i$ číslice v soustavě se základem $z$ a $A_i$ číslice v soustavě se základem $Z$, snadno se přesvědčíme, že platí vztah

$$(\ldots a_3 a_2 a_1 a_0, a_{-1} a_{-2} a_{-3} \ldots)_z = (\ldots A_3 A_2 A_1 A_0, A_{-1} A_{-2} A_{-3} \ldots)_Z$$

kde číslice $A_j$ vznikne ze skupiny $k$ číslic $a_i$

$$A_j = (a_{nj+n-1} \ldots a_{nj+1} a_{nj})_z$$

Protože $a_i$ jsou číslice $z$-ární soustavy, je $0\ a_i < z$, takže pro libovolné $n$-místné číslo

$$(a_0 \dots a_{n-2}a_{n-1})_z$$

platí

$$0 \leq (a_0 \dots a_{n-2}a_{n-1})_z = \sum_{i=0}^{n-1} a_i z^i \leq \sum_{i=0}^{n-1} (z-1)\, z^i = (z-1)\, \frac{z^n - 1}{z-1} = z^n - 1.$$

To znamená, že skupina $(a_0 \dots a_{n-2}a_{n-1})_z$ opravdu může představovat číslici v soustavě se základem $Z = z^n$.

Převedeme číslo, vyjádřené ve dvojkové soustavě $x = (111101011011001)_2$ do šestnáctkové soustavy. Nejprve rozdělíme $x$ na skupiny po čtyřech,

$$\begin{array}{cccc} \underbrace{(1111)} & \underbrace{(0101)} & \underbrace{(1011)} & \underbrace{(0001)} \\ 15 & 5 & 11 & 1 \\ F & 5 & B & 1 \end{array}$$

Číslo $x$ má tedy v šestnáctkové soustavě tvar $F6B1$.

Poznámka: Jako základ číselné soustavy se obvykle volí celé číslo $z > 1$. Volby $z = 1$ a $z = 0$ zjevně nemají smysl. Mohli bychom však za základ vzít celé číslo $z < -1$[2]. V takové soustavě lze vyjádřit jakékoli celé číslo, kladné i záporné, aniž bychom potřebovali znaménko. Např. v soustavě se základem $-10$ se číslo $(-1)_{10}$ vyjádří jako $(19)_{-10}$, neboť $1.(-10) + 9.1 = -1$.

Základem číselné soustavy mohou být i komplexní čísla. Např. soustava se základem $z = 2i$ umožňuje vyjádřit libovolné komplexní číslo s celými složkami jako posloupnost číslic 0, 1, 2 a 3. Podrobnější informace o číselných soustavách najdete např. v Knuthově knize [3].

V tomto i v následujících algoritmech budeme označení číselné soustavy vynechávat.

### 7.2.2 Sčítání celých čísel

Postup při sčítání bude vycházet ze známých algoritmů ze základní školy. Sčítáme dvě $k$-místná čísla, $u$ a $v$, kde $u = u_1u_2 \dots u_{k-1}u_k$ a $v = v_1v_2 \dots v_{k-1}v_k$ (pozor, nyní indexujeme číslice v opačném pořadí než v (7.1)). Výsledek bude

$$u + v = w = w_1w_2 \dots w_{k-1}w_k;$$

($u_i$, $v_i$ a $w_i$ jsou číslice, které tato čísla tvoří). Čísla $u$, $v$ a $w$ mohou začínat nulami. Při sčítání použijeme pomocnou proměnnou, kterou budeme označovat $p$ a nazývat *přenos*.

Vlastní algoritmus sčítání dvou celých čísel bez znaménka bude vypadat takto:

1. Polož $p := 0$ a pro $i = k, k-1, \dots 1$ opakuj následující kroky:
2. Vypočti $b := u_i + v_i + p$.
3. Je-li $b < z$, polož $w_i := b$ a $p := 0$; jinak polož $p := 1$ a $w_i := b - z$.

Protože součet dvou číslic nepřesáhne hodnotu $2z - 2$, nemůže být přenos po sečtení prvních dvou číslic větší než 1. Při sečtení druhé - a každé následující - dvojice nemůže součet $u_i + v_i + p$ přesáhnou $2z - 1$, takže přenos při sčítání bude vždy buď 0 nebo 1. Proto mu v popsaném algoritmu přiřazujeme pouze 0 nebo 1.

Je-li po skončení tohoto algoritmu $p = 1$, došlo k „přetečení“, výsledek má $k + 1$ míst (a výsledek, který jsme získali, je nesprávný - chybí v něm nejvyšší pozice). Uvedený algoritmus tedy vlastně představuje sčítání modulo $z^k$.

Popsaný algoritmus můžeme používat i při sčítání v doplňkovém kódu. Zde se ovšem může stát, že výsledkem sčítání dvou kladných čísel bude hodnota větší než největší zobrazitelné kladné číslo, avšak menší, než největší zobrazitelné kladné číslo v přímém kódu. Výsledek se pak bude jevit jako záporné číslo.

[2] Musíme ovšem pravidlo, které přikazuje, že číslice je menší než základ, nahradit pravidlem, že číslice je menší než absolutní hodnota základu, $0\ a < |z|$.

Uvažujme osmibitovou binární aritmetiku se znaménkem (např. typ *shortint* z Turbo Pascalu). Sečteme-li $127 + 2$, dospějeme popsaným algoritmem k hodnotě 129, která se ovšem interpretuje už jako záporné číslo v doplňkovém kódu. Z definice doplňkového kódu při 8 bitech plyne, že číslo 129 je obrazem záporného čísla $-127$. Dostaneme tedy poněkud překvapující výsledek, že $127 + 2 = -127$.

Ukážeme si, že pokud nenastane přetečení, můžeme uvedený postup použít i pro sčítání záporných čísel. Je-li $x < 0$, $y < 0$, sčítáme jejich obrazy,

$$\left(z^k + x\right) + \left(z^k + y\right) = \left(2z^k + x + y\right) = \left(z^k + x + y\right) \bmod z^k$$

neboť počítáme v aritmetice modulo $z^k$. Poslední výraz ale představuje obraz čísla $x + y$. Podobně ukážeme, že uvedený algoritmus lze použít i pro případ $x > 0$, $y < 0$ nebo $x < 0$, $y > 0$, kde ovšem nemůže dojít k přetečení.

### 7.2.3 Odečítání celých čísel

Algoritmus pro odečítání bude podobný algoritmu pro sčítání; opět využívá pomocnou proměnnou *přenos*. Odečítáme dvě $k$-místná čísla, $u$ a $v$, kde $u = u_1u_2 \ldots u_{k-1}u_k$ a $v = v_1v_2 \ldots v_{k-1}v_k$; výsledek bude

$$u - w = w = w_1w_2 \ldots w_{k-1}w_k.$$

Algoritmus pro odečtení dvou celých čísel bez znaménka má následující tvar.

1. Polož $p := 0$ a pro $i := k, k-1, \ldots, 1$ opakuj následující kroky:
2. Vypočti $b := u_i - v_i - p$.
3. Je-li $b < 0$, položíme $w_i := z - b$ a $p := 1$; jinak položíme $w_i := b$ a $p := 0$.

Opět jde o operaci v aritmetice modulo $z^k$. Je-li $u < v$, dostaneme výsledek $u - v + z^k$. Snadno se přesvědčíme, že tento postup lze použít i pro odečítání záporných čísel v doplňkovém kódu.

Jiný možný postup: K číslu $v$ sestrojíme číslo opačné, $-v$, a sečteme $u + -v$ pomocí algoritmu z předchozího oddílu.

### 7.2.4 Opačné číslo

Jako číslo opačné k $v$ se označuje číslo $-v$. Ukážeme si, jak je v doplňkovém kódu sestrojíme. Budeme jako obvykle předpokládat, že $v = v_1v_2 \ldots v_{k-1}v_k$ a $z$ je základ číselné soustavy.

Nejprve sestrojíme „negované“ číslo $v'$, pro jehož číslice platí

$$v'_i = z - v_i - 1.$$

Snadno se přesvědčíme, že součet $v + v'$, sestrojený podle výše uvedeného algoritmu, se skládá z $k$ stejných číslic s hodnotou $z - 1$. To znamená, že součet $v + v' + 1$ bude (v aritmetice modulo zk) roven 0.

Algoritmus pro sestrojení opačného čísla je tedy jednoduchý:

1. Pro $i = 1, \ldots, k$ nahraď $v_i$ číslicí s hodnotou $z - v_i - 1$ (sestroj „negované“ číslo).
2. K výsledku předchozího kroku přičti 1.

Ve dvojkové soustavě získáme „negované“ číslo pomocí negace v po bitech (odtud název). Za předpokladu, že používáme dvojkovou soustavu s doplňkovým kódem, můžeme předchozí algoritmus vyjádřit pascalským příkazem

```
y := (not v) + 1;
```

Poznamenejme, že některé počítače mají pro výpočet opačného čísla zvláštní instrukci (na PC je to instrukce NEG).

### 7.2.5 Násobení celých čísel

Zde nám postačí umět vynásobit nezáporná celá čísla. V soustavě se základem $z$ hledáme součin dvou čísel $u = u_1u_2 \dots u_n$ a $v = v_1v_2 \dots v_m$. Výsledek bude

$$u \times v = w = w_1w_2 \dots w_{m+n}.$$

Všimněte si, že nyní nepředpokládáme stejný počet cifer v obou činitelích.

Vyjdeme opět od tradičního násobení na papíře. Při tomto všeobecně známém postupu jsme napočítávali parciální součiny a celkový výsledek jsme získali jako jejich součet. Při strojovém výpočtu bude výhodnější ihned přičítat jednotlivé číslice parciálních součinů k celkovému výsledku.

Algoritmus bude opět využívat pomocnou proměnnou $p$ pro přenos; $i$ a $j$ slouží jako parametry cyklů.

1. *Inicializace*: Položíme $w_{m+1} := 0, \dots, \ w_{m+n} := 0, j := m$.
2. *Test nuly*: Je-li $v_j = 0$, nastavíme $w_j := 0$ a jdeme na bod 6. (Tento krok lze vynechat.)
3. *Inicializace i*: $i := n$, $p := 0$.
4. *Násobení a sčítání*: Položíme $t := u_i \times v_i + w_{i+j} + p$. Dále položíme $w_{i+j} := t \bmod z$, $p := [t/z]$. Symbol $[x]$ zde znamená celou část čísla $x$. (Přenos bude vždy v rozmezí $0 \le p < z$, tedy jednociferný.)
5. *Konec cyklu podle i*: Zmenšíme $i$ o jedničku; je-li $i > 0$, vrátíme se na bod 4, jinak položíme $w_j := p$.
6. *Konec cyklu podle j*: Zmenšíme $j$ o jedničku; je-li nyní $j > 0$, vrátíme se na bod 2, jinak konec.

Tento algoritmus předpokládá, že mezivýsledek $t$ splňuje nerovnost $0 \le t < z^2$ a přenos $p$ že leží v rozmezí $0 \le p < z$.

Nerovnost pro přenos dokážeme matematickou indukcí podle jednotlivých kroků algoritmu. V prvním kroku zřejmě platí, neboť na počátku je $p = 0$. Nechť tedy platí pro nějaké $i$. Podle 4. bodu algoritmu dostaneme

$$u_i \times v_i + w_{i+j} + p \le (z-1) \times (z-1) + (z-1) + (z-1) = z^2 - 1 < z^2$$

(to je druhá z dokazovaných nerovností). Odtud plyne pro přenos $p = [t/z] \le z - 1$.

### 7.2.6 Dělení celých čísel

Také algoritmus dělení celých čísel v soustavě o základu $z$ vychází z tradičního postupu dělení. Budeme se opět zabývat pouze dělením nezáporných čísel a budeme předpokládat, že dělencem je číslo $u = u_1u_2 \dots u_{m+n}$, dělitelem číslo $v = v_1v_2 \dots v_n$. Podíl pak mít hodnotu $q = [u/v]$ a bude to $m$-ciferné číslo $q = q_0q_1 \dots q_m$. Zbytek po dělení $r$ bude mít nejvýše $n$ cifer, $r = r_1 \dots r_n$.

Prvním krokem je normalizace, která zajistí, že první cifra dělitele bude větší než $[z/k]$. Toto opatření zajistí lepší vlastnosti odhadu číslic podílu.

1. *Normalizace*: Položíme $d := [z/(v_1 + 1)]$. Dále položíme $u := u \times d$, $v := v \times d$.
2. *Inicializace j*: Polož $j := 0$. (Bude následovat cyklus podle $j$. Kroky 2 - 7 představují v podstatě dělení $u_j \dots u_{j+m}$ číslem $v_1 \dots v_n$ tak, abychom dostali jednočíselný podíl $q_j$.)
3. *Výpočet odhadu q′*: Je-li $u_j = v_j$, položíme $q' := z - 1$, jinak polož $q' := [(u_jz + u_{j+1})/v_1]$. Nyní zkusíme, zda je $v_2q' > (u_jz + u_{j+1} - q'v_1)\,z + u_{j+2}$. Pokud ano, zmenšíme $q'$ o 1 a test zopakujeme.
4. *Násobení a odečtení*: nahradíme $u_ju_{j+1} \dots u_{j+n}$ číslem $u_ju_{j+1} \dots u_{j+n} - (q' \times v_1v_2 \dots v_n)$. Jde o násobení jednomístným číslem a odečtení. Číslo $u_ju_{j+1} \dots u_{j+n}$ musí zůstat kladné. Je-li výsledek tohoto kroku záporný, zvětšíme jeho hodnotu o $z^n$, tj. vezmeme doplněk skutečné hodnoty, a tuto „výpůjčku“ si zapamatujeme.
5. *Test zbytku*: Položíme $q := q'$. Je-li výsledek kroku 4 záporný, jdeme na 6, jinak jdeme na 7.
6. *Zpětné přičtení*: Zmenšíme $q$ o 1. K $u_ju_{j+1} \dots u_{j+n}$ přičti $0v_1v_2 \dots v_n$. (Objeví se přenos; ten se ignoruje, neboť pouze ruší „výpůjčku“ ze 4. kroku.)
7. *Konec cyklu podle j*: $j := j + 1$. Je-li nyní $j \le m$, vrátíme se na krok 3.
8. *Zrušení normalizace*: Nyní je $q_0q_1 \dots q_m$ podíl. Zbytek získáme dělením $u_{m+1} \dots u_{m+n}$ číslem $d$ (faktorem, který jsme použili k normalizaci). Jde o dělení jednociferným číslem.

**Příklad 7.2 Celá čísla na PC**

Procesory Intel 80x86 umožňují používat buď celá čísla bez znamének nebo celá čísla se znaménky v doplňkovém kódu. Máme na vybranou tyto možnosti:

| velikost | znaménko | rozsah hodnot | typ v Pascalu | typ v C |
|---|---|---|---|---|
| 1 B | se znaménkem | -128..127 | *shortint* | **signed char** |
| 1 B | bez znaménka | 0..255 | *byte* | **unsigned char** |
| 2 B | se znaménkem | -32768..32767 | *integer* | **int, short** |
| 2 B | bez znaménka | 0..65535 | *word* | **unsigned, unsigned short** |
| 4 B | se znaménkem | -2147483648..2147483647 | *longint* | **long** |
| 4 B | bez znaménka | 0..4294967295 | - | **unsigned long** |
| 8 B | se znaménkem | $-2^{63}..2^{63}-1$ | *comp* | - |

[3]

Tabulka 7.1 Celá čísla na PC a odpovídající typy v Turbo Pascalu a v Borland C++

Poznamenejme, že typ comp vyžaduje použití matematického koprocesoru.

## 7.3 Reálná čísla

Pod označením *reálná čísla* se v počítačové terminologii ukrývá konečná podmnožina množiny racionálních čísel. Vedle toho se občas hovoří o *číslech s pohyblivou řádovou čárkou* (*tečkou*), což je kalk anglického termínu *floationg point numbers*.

### 7.3.1 Zobrazení reálných čísel

Zobrazení reálných čísel se opírá o semilogaritmický[4] zápis, běžně používaný k zápisu extrémně velkých nebo malých hodnot. Např. Planckova konstanta, známá z kvantové mechaniky, má hodnotu $h = 6,6256.10^{-34} Js$.

V tomto zápisu označujeme 6,6256 jako *mantisu* a $-34$ jako *exponent*.

V soustavě se základem z bychom k vyjádření reálného čísla mohli použít dvojici $(e, f)$, která by představovala číslo

$$(e, f) = fz^e.$$

Z praktických důvodů se ale používá kódování

$$(e, f) = fz^{e-q},$$

kde $f$ je $p$-ciferné číslo, vyjadřující zlomkovou část (mantisu), $e$ je exponent a $q$ je *posun* (*exces*); $e$ a $q$ jsou celá čísla. Požadujeme, aby pro mantisu platilo $|x| < 1$. Z toho plyne, že

$$-z^p < z^p f < z^p.$$

Zvolíme-li posun $e$ rovný 50 a počet cifer $p = 5$, můžeme Planckovu konstantu vyjádřit zápisem

$$h = (27, +0, 66256).$$

Pro mantisu se používá prakticky výlučně přímý kód, tj. kódování se znaménkem. Obvykle se také požaduje, aby zobrazované číslo bylo *normalizované*, tj. aby pro mantisu platilo

$$z^{-1} \leq |f| \leq 1 \qquad \text{nebo } f = 0. \tag{7.2}$$

Setkáme se ale i s jinými definicemi normalizovaného čísla. Za normalizované můžeme např. pokládat číslo, jehož mantisa splňuje podmínky

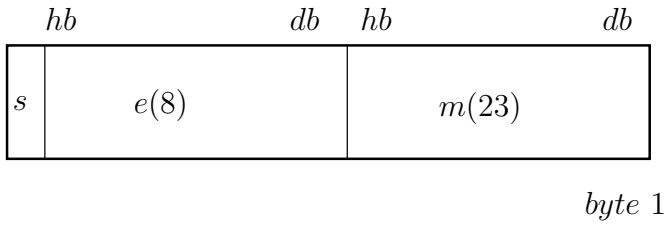
$$1 \leq |f| \leq z \qquad \text{nebo } f = 0.$$

V případě, že je mantisa nulová, $f = 0$, musí mít exponent nejmenší možnou hodnotu.

Z těchto požadavků vychází uložení reálných čísel v paměti počítače. Je-li $r$ proměnná, do které ukládáme reálná čísla, bude část z ní obsahovat exponent, část mantisu a jeden bit bude obsahovat znaménko mantisy.

[3] Uvedené hodnoty platí pro překladače pro "šestnáctibitové" programy (tj. např. programy pro DOS nebo pro Windows 3.1). Ve "dvaatřicetibitových" programech (např. v programech pro Windows NT) se v Céčku používají typy **int** a **unsigned int** o velikosti 4 B a velikost 2 B mají pouze typy **short** a **unsigned short**.

[4] Z návodů ke kalkulačkám známe také poněkud nesmyslné označení *vědecký zápis* (*scientific notation*). Dosud se mi nepodařilo zjistit, co je na tomto zápisu vědeckého - pokud vím, učí se v 8. třídě základních škol.




Obr. 7.1: Uložení čtyřbytového reálného čísla v paměti PC; $s$ je 1 bit, určující znaménko mantisy, $m$ je mantisa, $e$ je exponent; čísla v závorkách udávají počet bitů; *db* resp. *hb* je dolní, nejméně významný resp. horní, nejvýznamnější bit složky.

## Příklad 7.3 Reálná čísla na PC

Zobrazení reálných čísel na PC, tedy na počítačích vybavených procesory Intel 80x86, vychází z vlastností matematického koprocesoru Intel 80x87[5]. Máme tedy k dispozici 3 typy reálných čísel, které se v Turbo Pascalu označují *single*, *double* a *extended*; v Céčku se pro tyto typy používá označení **float**, **double** a **long double**.

Uložení hodnot těchto typů v paměti PC vychází v podstatě z popsaného způsobu zobrazení; navíc umožňují pracovat se speciálními hodnotami ±INF a ±NaN. Hodnoty ±INF představují „strojové nekonečno“, to znamená, že se ve výrazech chovají podobně jako v limitách v matematické analýze.

INF lze - při určitém nastavení stavového slova matematického koprocesoru - získat např. jako výsledek operace 1.0/0.0. (Tak je tomu např. při standardním nastavení v překladači Borland C++ 3.1; překladač Turbo Pascalu používá jiné nastavení, které způsobí, že dělení nulou vyvolá chybu.)

NaN je zkratka ze slov „Not a Number“, tedy údaj, který nelze chápat jako číslo. Vznikne zpravidla jako výsledek chyby v definičním oboru parametrů matematických funkcí, např. při pokusu o odmocninu ze záporného čísla. (Připomeňme si, že výpočty hodnot běžných matematických funkcí, jako je odmocnina, logaritmus, sinus apod., představují jednu instrukci matematického koprocesoru 80x87.)

Podívejme se nyní, jak jsou tyto datové typy zobrazeny v paměti PC.

Typ *single* je v paměti uložen následujícím způsobem (viz obr. 7.1): pro mantisu ve vyhrazeno 23 bitů, pro exponent 8 bitů a pro znaménko mantisy 1 bit. Nejvýznamnější bit jednotlivých složek je vždy vlevo, nejméně významný vlevo. Všimněte si, že hranice jednotlivých částí se nekryjí s hranicemi bytů.
Exponent leží v rozmezí $0 \leq e \leq 255$, posun $q$ je 127. Pro „běžná“ čísla se používají hodnoty exponentu v rozmezí $0 < e < 255$; hodnota 0 je vyhrazena pro nenormalizovaná čísla, hodnota 255 pro INF a NaN. Za normalizovaný pokládáme takový tvar, kdy je $1 \leq m < 2$ nebo $m = 0$. Pokud je $m \geq 1$, začíná mantisa jedničkou a tu nemusíme zobrazovat. Obraz mantisy se proto skládá pouze z číslic za řádovou tečkou. Skutečnou hodnotu mantisy budeme v tomto případě vyjadřovat zápisem $1.m$. Pouze pro nejmenší možnou hodnotu exponentu se předpokládá, že mantisa začíná nulou, že jde o tzv. *denormální číslo*; její hodnota je pak interpretována jako $0.m$.

Hodnotu $v$, uloženou v proměnné typu *single*, vypočteme z následujících vztahů:

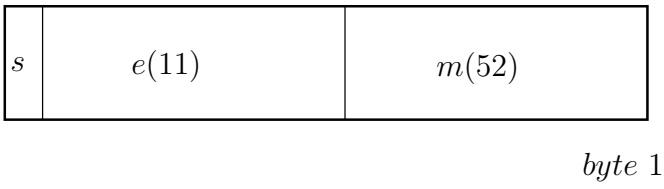
$$\text{Je} - \text{li } 0 < e < 255, \text{je} \qquad v = (-1)^s \times 2^{e-q} \times (1.m)\,; \tag{7.3}$$

$$\text{Je} - \text{li } e = 0, m \neq 0, \text{je} \qquad v = (-1)^s \times 2^{1-q} \times (0.m)\,;$$

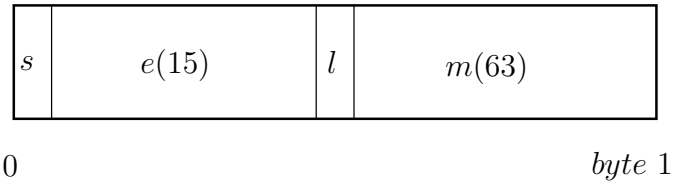
$$\text{Je} - \text{li } e = 0, m = 0, \text{je} \qquad v = (-1)^s \times 0;$$

$$\text{Je} - \text{li } e = 255, m = 0, \text{je} \qquad v = (-1)^s \times \text{INF};$$

[5]Typy reálných čísel v podstatě odpovídají standardu světové organizace elektrických a elektrotechnických inženýrů IEEE [30]. Základní informace o tom, jak pracuje matematický koprocesor, najdeme např. v [34].




Obr. 7.2: Uložení osmibytového reálného čísla v paměti PC; čísla v závorkách udávají počet bitů.




Obr. 7.3: Uložení desetibytového reálného čísla v paměti PC; $l$ je bit před řádovou tečkou mantisy, čísla v závorkách udávají počet bitů.

$$\text{Je} - \text{li } e = 255, m \neq 0, \text{je} \qquad v = \text{NaN}.$$

Typ *double* je uložen v 8 bytech, jak to ukazuje obr. 7.2. Exponent leží v rozmezí $0 \leq e \leq 2048$, posun $q$ je 1023. Pravidla pro výpočet hodnoty jsou podobná jako u čtyřbytových reálných čísel, pouze musíme ve vztazích (7.3) nahradit horní mez exponentu 255 hodnotou 2047.
Typ *extended* je uložen v 10 bytech, jak ukazuje obr. 7.3. Exponent $e$ leží v rozmezí $0 \leq e \leq 32767$, posun $q$ je 16383. U tohoto typu se vždy zobrazuje i jednička nebo nula před řádovou tečkou v mantise (jednobitové pole $l$ na obr. 7.4). Hodnotu $v$, uloženou v proměnné typu *extended*, vypočteme takto:

$$\text{Je} - \text{li } 0 < e < 32767, \text{je} \qquad v = (-1)^s \times 2^{e-q} \times (l.m)\,.$$

$$\text{Je} - \text{li } e = 32767, m = 0, \text{je} \qquad v = (-1)^e \times \text{INF};$$

$$\text{Je} - \text{li } e = 32767, m \neq 0, \text{je} \qquad v = \text{NaN}.$$

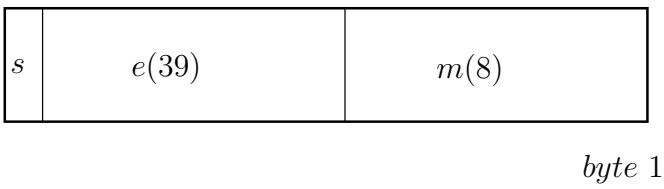
V Turbo Pascalu najdeme navíc typ *real*, se kterým ovšem matematický koprocesor neumí pracovat. Je uložen v 6 bytech (viz obr. 7.4), posun $q$ je 129; tento typ neumožňuje zobrazit hodnoty ±INF a ±NaN a denormální čísla.
Hodnotu $v$, uloženou v šestibytovém rálném čísle, vypočteme takto:

$$\text{Je} - \text{li } 0 < e < 255, \text{je} \qquad v = (-1)^s \times 2^{e-q} \times (1.m)\,,$$

$$\text{Je} - \text{li } e = 0, \text{ je} \qquad v = 0.$$

V následujících oddílech se budeme zabývat algoritmy pro základní aritmetické operace s reálnými čísly. Jak uvidíme, nejsou přesné - nalezená dvojice $w$ se bude zpravidla lišit od obrazu matematického součtu $u + v$, rozdílu $u - v$, podílu $u/v$, nebo součinu $uv$. Rozdíl mezi nalezenou hodnotou $w$ a skutečným výsledkem matematické operace označujeme jako *zaokrouhlovací chybu*.

Vzhledem k zaokrouhlovacím chybám budeme pro operace pomocí uvedených algoritmů používat místo „+“ značku „+°“, místo „/“ značku „/°“ apod.




Obr. 7.4: Uložení čísla typu real v paměti PC; čísla v závorkách opět udávají počet bitů.

### 7.3.2 Sčítání a odečítání reálných čísel

Jak uvidíme, je algoritmus pro sčítání reálných čísel nejkomplikovanější z algoritmů pro práci s reálnými čísly.

Předpokládáme, že je opět dán základ $z$, posun $q$, počet cifer $p$ a dvě *normalizovaná* reálná čísla $u = (e_u, f_u)$ a $v = (e_v, f_v)$. Hledáme součet $w = u + v$, tj. hledáme dvojici $(e_w, f_w)$, která bude obrazem součtu $u + v$ (nebo přesněji $u +^\circ v$).

1. *Rozbalení*: V reprezentaci čísel $u$ a $v$ od sebe oddělíme exponent a mantisu. To znamená, že budeme u obou čísel $u$ a $v$ pracovat zvlášť s mantisou $f$ a zvlášť s exponentem $e$.
2. *Předpokládáme* $e_u \geq e_v$: Je-li $e_u < e_v$, zaměníme $u$ a $v$.
3. *Určení exponentu výsledku* $e_w$: Položíme $e_w = e_u$.
4. *Test* $e_u - e_v$: Je-li $e_u - e_v \geq p + 2$ (velký rozdíl exponentů), položíme $f_w := f_u$ a půjdeme na krok 7. (Protože jsme předpokládali, že jsou obě čísla normalizovaná, mohli bychom skončit.)
5. *Posun doprava*: Posuneme $f_v$ o $e_u - e_v$ míst doprava (tzn. vydělíme $f_v$ číslem $z^{eu-ev}$). Jde o posun o maximálně $p + 1$ míst doprava, to znamená, že potřebujeme registr, který bude mít alespoň $2p + 1$ míst. Dá se ale ukázat, že za jistých okolností lze tento požadavek zredukovat na $p + 2$ míst.
6. *Sečtení*: Položíme $f_w := f_u + f_v$. Zde použijeme algoritmus pro sčítání celých čísel, se kterým jsme se seznámili v 7.2.2.
7. *Normalizace*: V tomto bodě algoritmu již $(e_w, f_w)$ představuje $u +^\circ v$, avšak $w$ nemusí splňovat podmínky normalizace, tj. $f_w$ může mít více než $p$ číslic, může být větší než 1 nebo menší než $1/z$ (případně větší než $z$ nebo menší než 1 - záleží na tom, jak definujeme normalizované číslo). Normalizaci popíšeme jako samostatný algoritmus v následujícím oddílu.

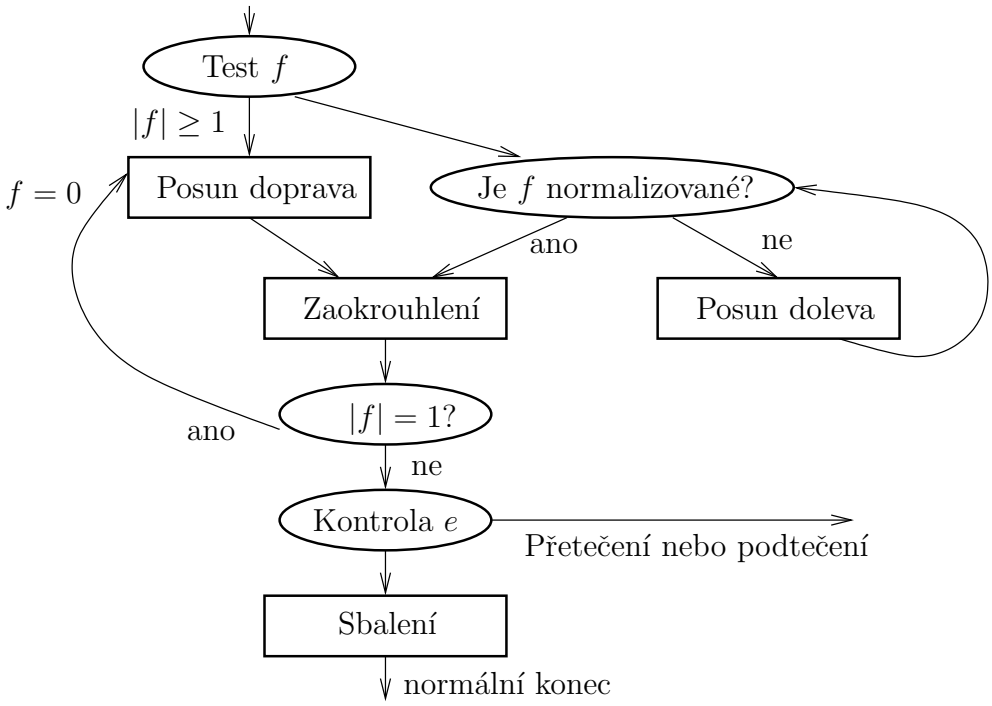
Jestliže v tomto algoritmu zaměníme $v$ za $-v$, dostaneme algoritmus pro odečítání reálných čísel, tedy pro výpočet $u -^\circ v$.

### 7.3.3 Normalizace reálného čísla

Při normalizaci převedeme „hrubý“ exponent $e$ a „hrubou“ mantisu $f$ do normalizovaného tvaru a mantisu v případě potřeby zaokrouhlíme na $p$ číslic. Uvedeme algoritmus pro normalizační podmínku (7.2); předpokládáme, že $|f| < z$ a $z$ že je sudé číslo.

1. *Test* $f$: Je-li $|f| \geq 1$ (nastalo „přetečení mantisy“), jdeme na 4. Je-li $f = 0$, nastavíme $e$ na nejmenší možnou hodnotu a jdeme na 7.
2. *Je* $f$ *normalizované*? Je-li $|f| \geq 1/z$, jdeme na 7.
3. *Posun doleva*: Na tento krok přijdeme pouze v případě, že $f < 1/z$. Posuneme $f$ o jednu pozici doleva (tj. vynásobíme $f$ základem $z$) a zmenšíme $e$ o 1. Vrátíme se na 2.
4. *Posun doprava*: Na tento krok přijdeme pouze v případě, že $|f| \geq 1$. Posuneme $f$ o jednu pozici doprava (tj. vydělíme $f$ základem $z$) a zvětšíme $e$ o 1.
5. *Zaokrouhlení*: Zaokrouhlíme $f$ na $p$ míst. Zpravidla používáme vztahů
$$f := z^{-p}\,[z^p f + 0,5] \text{ pro } f > 0, \qquad f := z^{-p} z^p f - 0,5 \text{ pro } f < 0.$$
Zde $[x]$ označuje celou část čísla $x$, tj. nejbližší menší celé číslo, $x$ označuje „horní celou část“, nejbližší větší celé číslo. Lze použít i jiných vztahů; ty, které jsme uvedli, však vedou k nejpříznivějšímu rozložení zaokrouhlovacích chyb. V tomto místě může dojít k „zaokrouhlovacímu přetečení“: z mantisy $|f| < 1$ vznikne zaokrouhlením mantisa $|f| = 1$. V takovém případě se vrátíme na 4.
6. *Kontrola* $e$: Je-li $e$ příliš velké (přesahuje-li povolený rozsah), nastalo „přetečení exponentu“. Je-li $e$ příliš malé, nastalo „podtečení exponentu“. Tyto situace se obvykle hodnotí jako chyby, neboť výsledek nelze vyjádřit jako normalizované reálné číslo.) Nastavíme příznak přetečení nebo podtečení.
7. *Sbalení*: Složíme $e$ a $f$ dohromady do předepsaného tvaru reprezentace reálného čísla.

Algoritmus normalizace lze znázornit vývojovým diagramem, který vidíte na obr. 7.5. Kvůli pohodlí v něm rozhodování místo předepsané značky vyznačíme oválem.




Obr. 7.5: Vývojový diagram algoritmu normalizace reálného čísla

### 7.3.4 Násobení a dělení reálných čísel

Algoritmy pro násobení a dělení reálných čísel se liší pouze v jednom kroku, takže je zapíšeme společně. Odvolávají se na algoritmy pro násobení resp. dělení celých čísel.

Opět předpokládáme, že je dán základ $z$, posun $q$, počet číslic $p$ a normalizované reálná čísla $u = (e_u, f_u)$ a $v = (e_v, f_v)$. Hledáme jejich součin $w = u \times^\circ v$ resp. podíl $w = u/^\circ v$. Podobně jako u předchozího algoritmu předpokládáme, že základ $z$ je sudý.

1. *Rozbalení*: V reprezentaci čísel $u$ a $v$ od sebe oddělíme exponent a mantisu. To znamená, že budeme u obou čísel $u$ a $v$ pracovat zvlášť s mantisou $f$ a zvlášť s exponentem $e$.

2. *Vlastní násobení nebo dělení*: V případě násobení položíme
$$e_w = e_u + e_v - q, \qquad f_w = f_u + f_v \tag{7.4}$$
v případě dělení pak
$$e_w = e_u - e_v + q + 1, \qquad f_w = \frac{\left(\frac{1}{z} f_u\right)}{f_v} \tag{7.5}$$
Ve vztazích (7.4 a 7.5) používáme operace násobení resp. dělení pro celá čísla. Protože jsou obě čísla podle předpokladu normalizovaná, bude buď $f_w = 0$, nebo $1/z^2 \leq |f_w| > 1$, nebo nastalo dělení nulou. V případě potřeby můžeme také zkrátit v tomto místě $f_w$ na $p + 2$ číslic (odseknout přebývající číslice).

3. *Normalizace*: Podobně jako v případě sčítání a odečítání zbývá normalizovat výsledek (a sbalit exponent s mantisou do jedné proměnné).

### 7.3.5 Převod celého čísla na reálné a naopak

Je dáno celé číslo $i$ v doplňkovém kódu. Hledáme reálné číslo $u = (e, f)$ tak, aby se hodnota $i$ rovnala hodnotě $u$. Předpokládáme, že dané celé číslo má $j$ míst, hledané reálné číslo má mít $p$ míst mantisy.

K tomu můžeme použít následující algoritmus:

1. Položíme $e = j$.

2. Zjistíme znaménko výsledku a zapamatujeme si je.
3. Položíme $f := |i|\, z^{p-j}$.
4. Normalizujeme $u = (e, f)$ a upravíme znaménko mantisy výsledku.

**Příklad 7.4**

Budeme pracovat v dekadické aritmetice. Počet míst reálného čísla $j = 5$, počet míst mantisy je $p = 10$. Chceme převést převést číslo $i = -123$ na reálné číslo $u$ se stejnou hodnotou.

Položíme tedy $e := 5$ a zapamatujeme si, že $i < 0$. Dále budeme používat $|i| = 123$. V dalším kroku položíme $f := 123 \times 10^{10-5} = 123 \times 10^5$. Protože mantisa obsahuje pouze číslice za desetinnou tečkou, dospěli jsme tak k vyjádření $u = 0,0012300000 \times 10^5$. Normalizací dospějeme k vyjádření $u = 0,12300000 \times 10^3$ nebo $u = 1,2300000 \times 10^2$ (podle toho, jak definujeme normalizovaný tvar). Nakonec upravíme znaménko mantisy – dostaneme $u = -0,12300000$.

Nyní se podívejme na opačnou úlohu. Je dáno reálné číslo $u = (e, f)$ a hledáme celé číslo $i$, jehož hodnota vznikne odseknutím zlomkové části $u$. Budeme předpokládat, že mantisa $f$ daného čísla $u$ má $p$ míst a hledané celé číslo že má $j$ míst.

Při tomto převodu může dojít k přetečení, bude-li celá část reálného čísla větší než největší zobrazitelná hodnota celého čísla.

1. *Rozbalení*: Rozložíme $u$ na exponent $e$ a mantisu $f$.
2. *Kontrola exponentu*: Je-li $e \leq 0$, položíme $i = 0$ a skončíme, neboť $|u| < 1$. Je-li $e > E$, kde $E$ je největší možná hodnota exponentu celého čísla, nastalo přetečení, chyba, konec.
3. Zjistíme znaménko $u$ a zapamatujeme si je.
4. Je-li $e = E$, jdeme na 6.
5. Zvětšíme $e$ o 1, posuneme $f$ o jednu pozici doprava a vrátíme se na krok 4.
6. Položíme $i := f/z^{p-j}$ (tj. přesuneme prvních $j$ míst $f$ do $i$).
7. Upravíme znaménko výsledku: je-li $u < 0$, položíme $i := -i$.

### Příklad 7.5

Předpokládáme opět dekadickou aritmetiku, deset míst mantisy a pětimístná celá čísla. Je dáno číslo $u = 32,1 = 0,321 \times 10^2$. Mantisa je tedy zobrazena jako desetimístné číslo 3210000000, exponent $e = 2$. Maximální možný exponent celého čísla je 5 (zapíšeme-li největší celé číslo jako reálné, dostaneme $0,99999 \times 10^5$.

Ve čtvrtém a pátém kroku algoritmu toto číslo upravíme do tvaru $0,000321 \times 10^5$., tj. mantisa bude nyní reprezentována číslem $0003210000 \times 10^2$. Vydělíme-li mantisu (jako celé číslo) číslem 105, dostaneme 0000000032, což je hledaná hodnota.

## 7.4 Přesnost aritmetiky reálných čísel

V tomto oddílu si naznačíme základní problémy, na které narážíme v souvislosti se zaokrouhlovacími chybami při výpočtech s reálnými čísly. Podrobnější informace lze najít např. v Knuthově knize [3].

### 7.4.1 Základní úvahy

Při úvahách o přesnosti výpočtů se obvykle používá *relativní chyba*. Jestliže reprezentujeme přesnou hodnotu $x$ v počítači přibližně hodnotou $\xi = x\,(1 + \varepsilon)$, označujeme veličinu $\varepsilon = (\xi - x)\,/x$ jako relativní chybu.

V následujících příkladech uvidíme, že zaokrouhlovací chyby mohou do aritmetických operací (zejména jde o sčítání a odečítání) vnést velkou relativní chybu. Jedním z nepříjemných důsledků pak je, že neplatí asociativní zákon,

$$(u +^\circ v) +^\circ w \neq u +^\circ (v +^\circ w)\,.$$

**Příklad 7.6 Asociativní zákon**

Ukážeme si příklad, ve kterém bude záležet na pořadí sčítání resp. odečítání. Předpokládáme dekadickou aritmetiku s osmimístnou mantisou. Budeme zapisovat pouze číslice, které jsou součástí mantisy, takže nám některá čísla budou končit desetinnou čárkou.

$$(11111113, +^\circ - 11111111, ) +^\circ 7,5111111 = 2,0000000 +^\circ 7,5111111 = 9,5111111;$$

$$11111113, +^\circ (-11111111, +^\circ 7,5111111) = 11111113, +^\circ - 11111103 = 10,000000;$$

Všimněte si, že relativní chyba výsledku je téměř 5%. Podobně lze ukázat, že asociativní zákon neplatí pro násobení reálných čísel podle algoritmu z odst. 7.3.4.

Skutečnost, že asociativní zákon neplatí, může mít obrovské důsledky. Uvědomme si, že řada běžných matematických zápisů, jako např. $a + b + c$ nebo $\sum_{i=1}^{n} a_i$ , je založena právě na předpokladu, že platí asociativní zákon.

Na druhé straně z tvaru algoritmu pro sčítání plyne, že pro operace „$+^\circ$“ platí komutativní zákon a pro „$+^\circ$“ a „$-^\circ$“ platí ještě některé další vztahy:

$$u +^\circ v = v +^\circ u; \tag{7.6}$$

$$u -^\circ v = u +^\circ -v,$$

$$-(u +^\circ v) = -u +^\circ -v,$$

$$u +^\circ 0 = u,$$

$$u -^\circ v = -(v -^\circ (-u))$$

Dále odtud plyne, že pokud platí $u +^\circ v = 0$, musí být $u = -v$. Pro libovolné $w$, pro které nenastane přetečení, platí:

$$\text{je} - \text{li } u < v, \text{ je } u +^\circ w < v +^\circ w \tag{7.7}$$

Vztah (7.7) není zcela zřejmý, proto jej dokážeme. Za tím účelem definujeme funkci $r(x, p)$, která bude zaokrouhlovat číslo $x$ na $p$ míst:

$$(x, p) = \begin{cases} z^{e-p}\,[z^{p-e}x + 0,5] & \text{pro } z^{e-1} \le x < z^e, \\ 0 & \text{pro } x = 0, \\ z^{e-p}\,[z^{p-e}x + 0,5] & \text{pro } z^{e-1} \le -x < z^e. \end{cases} \tag{7.8}$$

Odtud plyne $r(-x, p) = -(x, p)$ a $r(bx, p) = br(x, p)$ pro každé $b$. Operace, zavedené algoritmy z oddílů 7.3.2. a 7.3.4., splňují vztahy

$$u +^\circ v = r(u + v, p), \quad u -^\circ v = r(u - v, p),$$

$$u \times^\circ v = r(uv, p), \quad u/^\circ v = r(u/v, p),$$

za předpokladu, že nedojde k přetečení exponentu, tj. za předpokladu, že výsledky operací $u + v$, $u - v$, $u \times v$, $u/v$ leží ve správném rozmezí hodnot. Odtud a ze skutečnosti, že funkce $r$ je neklesající vzhledem k $x$, již plyne (7.7).

Na základě předchozích vztahů snadno zjistíme, že pokud nenastane přetečení, platí řada běžných vztahů, jako např.

$$v \times^\circ u = u \times^\circ v, \quad (-u) \times^\circ v = -(u \times^\circ v), \quad 1 \times^\circ u = u$$

$$u \times^\circ v = 0 \text{ právě když } u = 0 \text{ nebo } v = 0,$$
$$(-u)/^\circ v = u/^\circ (-v) = -(u/^\circ v), \qquad 0/^\circ v = 0, \qquad u/^\circ 1 = u, \qquad u/^\circ u = 1.$$

Bude-li platit $0 < u \le v$ a $w > 0$, zůstanou v platnosti obvyklé nerovnosti

$$u \times^\circ w \le v \times^\circ w, \qquad u/^\circ w \le v/^\circ w, \qquad w/^\circ u \ge w/^\circ v.$$

Na druhé straně následující příklad ukazuje, že neplatí distributivní zákon.

### Příklad 7.7 Distributivní zákon

Uvažujme opět dekadickou aritmetiku s osmi platnými číslicemi. Ukážeme si příklad, kdy pro operace, zavedené algoritmy z 7.3.2. a 7.3.4., neplatí distributivní zákon. Jako příklad vezmeme hodnoty tří proměnných $u = 20000,000$, $v = -6,0000000$, $w = 6,0000003$. Pak

$$(u \times^\circ v) +^\circ (u \times^\circ w) = -120000,00 +^\circ 120000,01 = 0,0100000,$$

zatímco

$$u \times^\circ (v +^\circ w) = 20000,00 \times^\circ 0,00000030 = 0,00600000.$$

Podobně vezmeme-li $u = v = 0,00005000$, bude platit nerovnost $2\,(u \times^\circ u + v \times^\circ v) < (u +^\circ v) \times^\circ (u +^\circ v)$, tj. - přepsáno běžným matematickým způsobem -

$$2\left(u^2 + v^2\right) < (u+v)^2.$$

To znamená, že při výpočtu standardní směrodatné odchylky podle vzorce

$$\sigma = \frac{1}{n}\sqrt{n\sum_{k=1}^{n} x_k^2 - \left(\sum_{k=1}^{n} x_k\right)^2}$$

se nám může stát, že budeme počítat odmocninu ze záporného čísla!

#### Příklad 7.8 Rovnost reálných čísel

Při řešení soustav lineárních algebraických rovnic, ale i při dalších numerických výpočtech, se setkáme s iteračními procedurami, založenými na rekurentním vztahu tvaru

$$x_{n+1} = f(x_n) \tag{7.9}$$

Z teorie víme, že posloupnost $x_n$ konverguje k limitě $x$. Nicméně bylo by chybou stanovit jako podmínku ukončení iteračního cyklu rovnost $x_n = x_{n+1}$, neboť vzhledem k zaokrouhlovacím chybám při výpočtu $f(x_n)$ může být posloupnost $x_n$ periodická.

Proto je rozumnější ukončit výpočet, bude-li např. $|x_n - x_{n+1}| < \delta$ pro $\delta$ vhodné . Takováto podmínka ovšem předpokládá, že známe alespoň řád limity $x$. Pokud nemáme žádné předběžné informace o výsledku, je vhodnější použít podmínku

$$\frac{|x_{n+1} - x_n|}{|x_n|} \le \varepsilon \tag{7.10}$$

Veličina ve vztahu (7.10) udává relativní míru nepřesnosti, kterou jsme ochotni přijmout jako výsledek iteračního procesu (7.9). Vztah (7.10) bychom také mohli interpretovat jako tvrzení, že $x_n$ „je přibližně rovno“ $x_{n+1}$.

### 7.4.2 Míra nepřesnosti

V předchozím odstavci jsme zjistili, že některé běžné vztahy pro základní aritmetické operace díky zaokrouhlovacím chybám neplatí. Podívejme se tedy podrobněji na vliv zaokrouhlovacích chyb algoritmů z oddílů 7.3.2. a 7.3.4.

Z definice (7.8) funkce $r$ plyne

$$r(x,p) = x\,(1 + \delta_p(x)), \qquad \text{kde } |\delta_p(x)| \le 0,5z^{1-p}.$$

To znamená, že můžeme vždy psát

$$a +^\circ b = (a+b)\,(1+\delta_p(a+b)), \qquad a \times^\circ b = (a \times b)\,(1+\delta_p(a \times b)),$$

$$a -^\circ b = (a-b)\,(1+\delta_p(a-b)), \qquad a/^\circ b = (a/b)\,(1+\delta_p(a/b)).$$

Tyto vztahy nám mohou posloužit jako podklad pro odhad chyby. Podívejme se na asociativní zákon pro násobení reálných čísel. Dostaneme postupně

$$(u \times^\circ v) \times^\circ w = ((uv)\,(1+\delta_i)) \times^\circ w = uvw\,(1+\delta_1)\,(1+\delta_2) \tag{7.11}$$

$$u \times^\circ (v \times^\circ w) = u \times^\circ ((vw)\,(1+\delta-3)) = uvw\,(1+\delta_3)\,(1+\delta_4).$$

Tyto vztahy platí (za předpokladu, že nedojde k přetečení exponentu) pro nějaká $\delta_i$, $i = 1,\ldots,4$. Připomeňme si, že všechna tato i vyhovují podmínce $|\delta_i| \le 0,5z^{1-p}$. To znamená, že

$$\frac{(u \times^\circ v) \times^\circ w}{u \times^\circ (v \times^\circ w)} = \frac{(1+\delta_1)\,(1+\delta_2)}{(1+\delta_3)\,(1+\delta_4)} = 1+\delta$$

kde $|\delta| =\leq 2z^{1-p}/\left(1-0,5z^{1-p}\right)^2$. Ukázali jsme tedy, že pokud nenastane přetečení exponentu, je $(u \times^\circ w)\times^\circ w$ nějakém smyslu přibližně rovno $u \times^\circ (v \times^\circ w)$.

Pokusme se dále upřesnit, co znamená *přibližně rovno*. Zavedeme nové operace pro porovnávání reálných čísel zobrazených v počítači, které budou brát v úvahu velikost porovnávaných čísel a možnou relativní chybu. Je-li $z$ základ číselné soustavy a $q$ posun a platí-li $u = (e_u, f_u)$ a $v = (e_v, f_v)$, definujeme

$u \prec v\,(\varepsilon)$, právě když $v - u > \varepsilon.\max\left(z^{e_u-q}, z^{e_v-q}\right)$ ... „$u$ je určitě menší než $v$“;

$u \sim v\,(\varepsilon)$, právě když $|v - u| \leq \varepsilon.\max\left(z^{e_u-q}, z^{e_v-q}\right)$ ... „$u$ je přibližně rovno $v$“;

$u \succ v\,(\varepsilon)$, právě když $u - v > \varepsilon.\max\left(z^{e_u-q}, z^{e_v-q}\right)$ ... „$u$ je určitě větší než $v$“;

$u \approx v\,(\varepsilon)$, právě když $|v - u| \leq \varepsilon.\max\left(z^{e_u-q}, z^{e_v-q}\right)$ ... „$u$ je v podstatě rovno $v$“.

V těchto definicích udává $\varepsilon$ uvažovanou relativní míru aproximace. Zavedené operátory splňují řadu jednoduchých vztahů, které ukazují pravidla pro počítání s přibližnými hodnotami, např.

$$u \prec v\,(\varepsilon) \Rightarrow v \succ u\,(\varepsilon)\,, \quad u \approx v\,(\varepsilon) \Rightarrow u \sim v\,(\varepsilon)\,, \quad u \approx u\,(\varepsilon)$$

$$u \succ v\,(\varepsilon_1) \wedge \varepsilon_1 < \varepsilon_2 \Rightarrow v\,(\varepsilon_2)\,. \tag{7.12}$$

Obdobné vztahy jako (11) platí i pro operace $\sim$ a $\approx$. Přímo z definice těchto operací dále plyne, že

$$je - li\, u \succ v\,(\varepsilon_1) \wedge v \succ w\,(\varepsilon_2) \Rightarrow u \succ w\,(\varepsilon_1 + \varepsilon_2)\,,$$

$$je - li\, u \approx v\,(\varepsilon_1) \wedge v \approx w\,(\varepsilon_2) \Rightarrow u \sim w\,(\varepsilon_1 + \varepsilon_2)\,.$$

Bez obtíží lze také dokázat, že

$$|u - v| \leq \varepsilon\,|u| \wedge |u - v| \leq \varepsilon\,|v| \Rightarrow u \approx v\,(\varepsilon)\,,$$

$$|u - v| \leq \varepsilon\,|u| \vee |u - v| \leq \varepsilon\,|v| \Rightarrow u \sim v\,(\varepsilon)\,,$$

a naopak, pro normalizovaná čísla $u$ a $v$ a $\varepsilon < 1$ platí

$$u \approx v\,(\varepsilon) \Rightarrow |u - v| \leq z\varepsilon\,|u| \wedge |u - v| \leq z\varepsilon\,|v|\,,$$

$$u \sim v\,(\varepsilon) \Rightarrow |u - v| \leq z\varepsilon\,|u| \vee |u - v| \leq z\varepsilon\,|v|\,.$$

**Příklad 7.9 Opět asociativní zákon**

Podíváme se, jak je to s asociativním zákonem pro násobení (viz též (7.11) a vztahy následující). Dostaneme

$$|(u \times^\circ v) \times^\circ w - u \times^\circ (v \times^\circ w)| = |u \times^\circ (v \times^\circ w)| \left|\frac{(1+\delta_1)(1+\delta_2)}{(1+\delta_3)(1+\delta_4)}\right| \leq \frac{2z^{1-p}}{\left(1-\frac{1}{2}z^{1-p}\right)}\,|u \times^\circ (v \times^\circ w)|\,.$$

Týž výsledek dostaneme i pro případ $|u \times^\circ (v \times^\circ w) - (u \times^\circ v) \times^\circ w|$. Z toho plyne, že pro

$$\varepsilon \geq 2z^{1-p}/\left(1-z^{1-p}\right)^2$$

platí

$$u \times^\circ (v \times^\circ w) \approx (u \times^\circ v) \times^\circ w.$$

Počítáme-li v desítkové soustavě, $z = 10$, s přesností na 8 platných cifer, $p = 8$, můžeme na základě tohoto výsledku položit $\varepsilon = 0,00000021$.

Na závěr uvedeme větu, která ukazuje, že rozdíl mezi $u +^\circ v$ a $u + v$ lze odhadnout pomocí veličin, které můžeme vypočítat s pomocí operací v jednoduché přesnosti. Tato věta ovšem platí pouze za předpokladu, že v normalizačním algoritmu použijeme zaokrouhlování, nikoli odřezávání.

**Věta**: Buďte $u$ a $v$ normalizovaná čísla. Položíme

$$u' = (u+^\circ) -^\circ v, \qquad v'' = (u +^\circ v) -^\circ u'.$$

Pak platí:

$$u + v = (u +^\circ v) + ((u -^\circ u') + (v -^\circ v''))\,.$$

Důkaz lze najít např. v Knuthově knize [3], str. 203.

**Příklad 7.10: použití Taylorovy řady**

Často se setkáme s úlohou vypočítat se zadanou přesností hodnoty nějaké transcendentní funkce. Jedním z prvních nápadů, jak postupovat, bývá využití Taylorovy nebo jiné mocninné řady. Ukážeme si, že se v tom mohou skrývat nepříjemné problémy.

Jak víme, jsou funkce *sinus* a *kosinus* definovány pomocí řad

$$\sin x = \sum_{k=0}^{+\infty} \frac{(-1)^k\, x^{2k+1}}{(2k+1)!}, \qquad \cos x = \sum_{k=0}^{+\infty} \frac{(-1)^k\, x^{2k}}{(2k)!} \tag{7.13}$$

Podívejme se, jak dopadne pokus počítat pomocí řady (7.13) funkci kosinus; závěry pro funkci sinus jsou podobné. Snadno dokážeme např. pomocí D'Alembertova kritéria, že řada pro tuto funkci konverguje pro libovolné komplexní $x$. Navíc jsou to řady se střídavými znaménky.

Podíl dvou po sobě následujících sčítanců je

$$\frac{a_{n+1}}{a_n} = -\frac{x^2}{(2n+1)\,(2n+2)}; \tag{7.14}$$

To znamená, že počínaje jistým $n_0\,(x)$ se bude pro libovolné $x$ absolutní hodnota $a_n$ monotónně zmenšovat. Jestliže tedy řadu (7.13) nahradíme konečným součtem

$$\cos x \approx \sum_{k=0}^{m} \frac{(-1)^k\, x^{2k}}{(2k)!},$$

dopustíme se chyby, která nepřesáhne absolutní hodnotu prvního zanedbaného členu. Tolik teorie, známá z matematické analýzy ze 2. semestru.

Nejprve se pokusíme spočítat hodnotu $\cos 1$ s chybou, která nepřesahuje $10^{-6}$. Podle (7.14) klesá $a_n$ monotónně k nule počínaje nultým členem. To znamená, že stačí najít $n$, pro které bude

$$|a_n| = \frac{1}{(2n)!} \le 10^{-6}.$$

Snadno se přesvědčíme, že tato podmínka je splněna již pro $n = 5$; to znamená, že stačí sečíst prvních 5 sčítanců v řadě (7.13) (pro $n = 0, \ldots, 4$). Použijeme-li kterýkoli z typů reálných čísel, popsaných v příkladu 7.3, budou zaokrouhlovací chyby menší než požadujeme, proto se jimi nemusíme zabývat.

Nyní se pokusme spočítat $\cos 100$. Podle (7.14) budou absolutní hodnoty sčítanců v řadě (7.13) monotónně narůstat pro $n = 0, \ldots, 50$ a teprve pak začnou klesat k nule. Největší člen, $a_{50}$, je

$$a_{50} = \frac{100^{200}}{100!}$$

Snadno zjistíme, že dekadický logaritmus $a_{50}$ je $\log a_{50} = 200 - 157,97 = 42,03$. To znamená, že největší ze sčítanců v řadě (7.13) má 43 číslic před desetinnou čárkou, ale my jej potřebujeme znát s přesností nejméně $10^{-6}$. Celkem bychom tedy potřebovali mít k dispozici reprezentaci reálných čísel s alespoň 49 místy mantisy. Ovšem počítače PC nám poskytují maximálně 19 cifer. Proto výsledek, který bychom získali přímým výpočtem z řady (7.13), by neobsahoval ani jednu „spolehlivou" číslici.

V tomto případě je ovšem řešení snadné. Vzhledem k periodicitě funkcí cos resp. sin stačí odečíst vhodný násobek 2 a pak počítat hodnotu funkce pro argument z intervalu $\langle -\pi, \pi)$. Z dalších vztahů pro goniometrické funkce

$$\sin\left(x \pm \frac{\pi}{2}\right) = \pm \cos x, \qquad \cos\left(x \pm \frac{\pi}{2}\right) = \mp \sin x$$

plyne, že stačí počítat hodnoty z intervalu $-/2$, $/2$. Ze vztahů

$$\sin\left(\frac{\pi}{2} - x\right) = \cos x, \qquad \cos\left(\frac{\pi}{2} - x\right) = \sin x$$

nakonec můžeme uzavřít, že v případě funkcí sinus a kosinus můžeme úlohu výpočtu hodnoty vždy převést na součet řady (7.13) s $|x| < \pi/4 < 1$.

Nicméně i zde se může projevit vliv zaokrouhlovacích chyb. Vrat'me se k výpočtu $\cos 100$. Z toho, co jsme si řekli, plyne, že stačí spočítat

$$\cos\,(100 - 32\pi) = \cos\,(-0,5309849148762)$$

Zde jsme měli štěstí, že goniometrické funkce jsou periodické. Pro jiné funkce se může stát, že nebudeme moci Taylorův rozvoj využít vůbec.

# Kapitola 8

# Některé další algoritmy

V této kapitole se seznámíme s několika dalšími algoritmy, které se nehodily do žádné z předchozích kapitol.

## 8.1 Rozklad grafu na komponenty

Občas se setkáme s úlohou rozložit graf na komponenty.

**Příklad 8.1**

Vezměme orientovaný graf z obrázku 8.1. Jeho incidenční matice je

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}. \tag{8.1}$$

Je zřejmé, že se tento graf skládá ze dvou nezávislých komponent, tvořených uzly 1, 3 5 a 2, 4.

Při konstrukci algoritmu, který bude hledat rozklad grafu na komponenty, vyjdeme z očividného faktu, že pokud jak z uzlu $i$ tak z uzlu $j$ vede hrana do uzlu $k$, leží všechny tři ve stejné komponentě. Skutečnost, že v grafu existuje zároveň cesta $i \to k$ a $j \to k$, znamená, že incidenční matice bude obsahovat nenulové prvky $A_{ik}$ a $A_{jk}$.

To znamená, že do téže komponenty budou patřit ty řádky matice $A$, které oba obsahují nenulový prvek v témže sloupci.

Algoritmus, který určí komponenty grafu, popsaného incidenční maticí $A$, bude používat pomocných množin $S_i$, složených z řádkových indexů matice $A$. Můžeme jej formulovat takto:

1. Pro $i = 1, \ldots, n$ ($n$ je počet uzlů grafu, tedy počet řádků incidenční matice $A$) přiřadíme $i$-tému řádku matice $A$ množinu $S_i = \{i\}$.

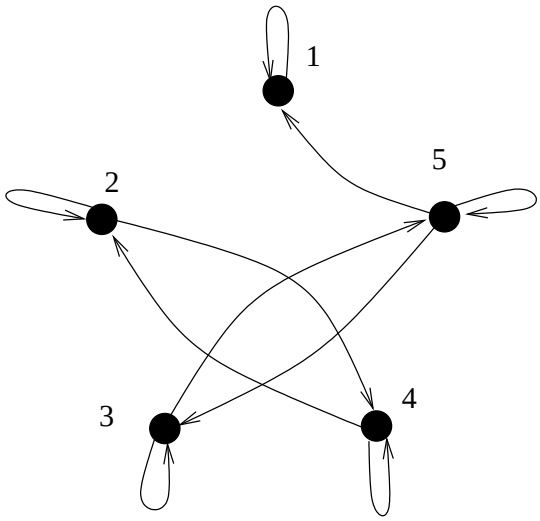
2. Najdeme sloupec $l$ s největším počtem nenulových prvků - nechť je to $q$. Je-li $q = 1$, jdeme na krok 5.

3. Uděláme logický součet[1] všech řádků, které mají v $l$-tém sloupci 1. Tyto řádky z matice $A$ odstraníme a nahradíme je vytvořeným logickým součtem.

4. Nově vytvořenému řádku přiřadíme množinu $S = S_i$, tj. sjednocení množin $S = \bigcup S_i$ pro odstraněné řádky, a vrátíme se na krok 2.

5. Každý řádek nyní představuje jednu komponentu grafu, popsaného maticí $A$. Množina $S_i$, která takovému řádku odpovídá, obsahuje čísla uzlů, které tvoří jednu komponentu.

[1] Logický součet definujeme pro čísla 0 a 1 takto: $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1 + 1 = 1$.




Obr. 8.1: Orientovaný graf matice $A$

**Příklad 8.1 (pokračování)**

Podívejme se, jak bude popsaný algoritmus fungovat v případě grafu z obrázku 8.1, popsaného maticí (8.1). Množiny $S_i$ jsou

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{array}{l} S_1 = \{1\} \\ S_2 = \{2\} \\ S_3 = \{3\} \\ S_4 = \{4\} \\ S_5 = \{5\} \end{array} .$$

Nejvíce nenulových prvků je v 5. sloupci (v 1., 3. a 5. řádku). Nahradíme proto 1. řádek logickým součtem 1., 3. a 5. řádku a množinu $S_1$ sjednocením $S_1 \cup S_3 \cup S_5$. Dostaneme

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{array}{l} S_1 = \{1,3,5\} \\ S_2 = \{2\} \\ S_3 = \{4\} \end{array} .$$

Nyní je nejvíce nenulových prvků ve 2. sloupci. Nahradíme proto 2. řádek logickým součtem 2. a 3. řádku a dostaneme

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{array}{l} S_1 = \{1,3,5\} \\ S_2 = \{2,4\} \end{array} .$$

Nyní již všechny sloupce obsahují nejvýše jeden nenulový prvek, takže algoritmus končí. Zjistili jsme, že jednu komponentu tvoří uzly 1, 3 a 5 a druhou zbývající dva.

## 8.2 Tranzitivní uzávěr orientovaného grafu

V tomto odstavci si ukážeme algoritmus, který umožňuje řešit následující úlohu: Máme orientovaný graf $G$ s $n$ uzly. Zajímá nás, jak zjistit, zda pro libovolné $i$ a $j$ v tomto grafu existuje cesta (libovolné délky) z uzlu $i$ do uzlu $j$.

S touto úlohou se setkáme, jestliže budeme např. zjišťovat, zda se v programu vyskytuje nepřímá rekurze, a budeme mít k disposici seznam podprogramů, které volají jednotlivé funkce a procedury. Další aplikace této úlohy se mohou týkat rozkladu matice soustavy rovnic apod.

Graf $G$ popíšeme incidenční maticí $A(G) = A$. Připomeňme si, že v $A$ platí $a_{ij} = 1$, právě když v $G$ existuje orientovaná hrana $\langle i, j \rangle$. To znamená, že $A$ popisuje všechny cesty délky 1.

Podívejme se nyní na cesty délky 2. Aby v $G$ existovala orientovaná cesta délky 2 z $i$ do $j$, musí existovat uzel s tak, že $G$ obsahuje hrany $\langle i, s \rangle$ a $\langle s, j \rangle$. Jestliže tedy projdeme všechny možné uzly $s$, zjistíme, zda v $G$ hledaná hrana existuje. Cesta z $i$ do $j$ přes pevně zvolené $s$ existuje, právě když jsou v incidenční matici grafu $G$ rovny jedné prvky $a_{is}$ a $a_{sj}$, tj. je-li $a_{is} \times a_{sj} = 1$.

Jakákoli cesta z $i$ do $j$ délky 2 existuje, je-li nenulový součet

$$b_{ij} = \sum_{s=1}^{n} a_{is} a_{sj}. \tag{8.2}$$

Je zřejmé, že $b_{ij}$ je prvek matice $B = A^2$. Budeme-li v (8.2) uvažovat místo obyčejného součtu logický součet, bude $B$ incidenční matice grafu $G_2$, který má stejný počet uzlů jako $G$, a ve kterém jsou uzly $\langle i, j\rangle$ spojeny hranou, jestliže v $G$ existuje cesta z $i$ do $j$ délky 2.

Podobně ukážeme, že $A^3$ je incidenční matice grafu $G_3$, který má stejný počet uzlů jako $G$, a ve kterém jsou uzly $\langle i, j\rangle$ spojeny hranou, jestliže v $G$ existuje cesta z $i$ do $j$ délky $3, \ldots, A^{n-1}$ je incidenční matice grafu $G_{n-1}$, který má stejný počet uzlů jako $G$, a ve kterém jsou uzly $\langle i, j\rangle$ spojeny hranou, jestliže v $G$ existuje cesta z $i$ do $j$ délky $n-1$. Vzhledem k tomu, že graf $G$ obsahuje $n$ uzlů, může mít nejkratší orientovaná cesta mezi dvěma různými uzly délku nejvýše $n-1$.

Chceme-li tedy umět pro každé $i$ a $j$ zjistit, zda v $G$ existuje cesta jakékoli délky z $i$ do $j$, stačí spočítat matici

$$Q = \sum_{i=1}^{n} A^i. \tag{8.3}$$

V tomto výpočtu používáme místo obyčejného sčítání opět logický součet.

$Q$ je incidenční matice tzv. *tranzitivního uzávěru* $G^-$ grafu $G$, tedy grafu, který obsahuje hranu $\langle i, j\rangle$, právě když $G$ obsahuje cestu libovolné délky z $i$ do $j$.

Budeme-li součet (8.3) počítat na základě definice násobení matic, budeme potřebovat $O\left(n^4\right)$ operací ($n$ násobení matic, každé vyžaduje $n^3$ operací).

Ukážeme si ale, že matici $Q$ můžeme spočítat s použitím $O\left(n^3\right)$ operací. Tento algoritmus vytvoří z matice $A$ přímo matici $Q$, tj. z grafu $G$ vytvoří přímo jeho tranzitivní uzávěr $G^-$. Jde o postup, ve kterém vytvoříme posloupnost grafů $G_0, G_1, \ldots, G_n$:

1. Položíme $G_0 = G$.
2. Pro $i = 1, 2, \ldots, n$ sestrojíme $G_i$ z $G_{i-1}$ takto: vezmeme $i$-tý uzel grafu $G_{i-1}$ a za každou dvojici $j$ a $k$, $j \in \{1, \ldots, n\}$, $k \in \{1, \ldots, n\}$, pro kterou v $G_{i-1}$ existují hrany $\langle j, i\rangle$ a $\langle i, k\rangle$, přidáme do $G_i$ hranu $\langle j, k\rangle$.
3. $G_n = G^-$.

Dokážeme, že tímto postupem opravdu získáme tranzitivní uzávěr grafu $G$. Označíme $H$ množinu hran grafu $G$, $H_i$ množinu hran grafu $G_i$ a $H^-$ množinu hran $G^-$ Je jasné, že stačí dokázat, že $H_n = H^-$. Z bodu 2 je zřejmé, že platí $H_i \subset H_{i+1}$. Matematickou indukcí dokážeme, že pro všechna $i = 1, 2, \ldots, n$ platí $H_i \subset H^-$.

Inkluze $H_0 \subset H^-$ je zřejmá: každá hrana původního grafu musí být i hranou tranzitivního uzávěru. Předpokládejme tedy, že pro nějaké i platí $H_i \subset H^-$ (tedy že z existence hrany $\langle i, j\rangle$ v $H_i$ plyne existence cesty z uzlu $i$ do uzlu $j$ v $H_0$).

Je-li $\langle k, j\rangle \in H_{i+1}$, znamená to, že buď hrana $\langle k, j\rangle \in H_i$, nebo že jsme ji do $H_{i+1}$ přidali na základě toho, že v $H_i$ existují pro nějaké $l$ hrany $\langle k, l\rangle$ a $\langle l, j\rangle$. V obou případech ale z indukčního předpokladu plyne, že v $H_0$ existuje cesta z $k$ do $j$. To znamená, že pro všechna i je $H_i \subset H^-$, a tedy speciálně $H_n \subset H^-$.

Nyní ještě potřebujeme dokázat opačnou inkluzi, tj. $H^- \subset H_n$. Je-li $\langle k, j\rangle \in H^-$, znamená to, že v $H$ existuje cesta $\langle k, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_s, j\rangle$, $s \leq n$. Vezmeme nejmenší z čísel $i_1, \ldots, i_s$; nechť je to číslo $i_r$. Pak ve druhém kroku algoritmu pro uzel $i_r$ přidáme do $H_i$ hranu $\langle i_{r-1}, i_{r+1}\rangle$. To znamená, že $H_r$ obsahuje cestu $\langle k, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_{r-1}, i_{r+1}\rangle, \ldots, \langle i_s, j\rangle$. Nyní opět vezmeme nejmenší ze zbývajících indexů atd. Tak ukážeme, že $\langle k, j\rangle \in H_n$. Tím je správnost algoritmu dokázána.

Na základě popsaného algoritmu můžeme napsat program, který transformuje incidenční matici $A$ grafu $G$ na incidenční matici $G^-$. Jeho základem bude konstrukce

```
for j := 1 to n do                              {pro každý uzel j}
  for i := 1 to n do                          {prozkoumáme všechny uzly i}
    if A[i,j] <> 0 then                       {existuje-li cesta z i do j}
      for k := 1 to n do if A[j,k] <> 0 then A[i,k] := 1;
              {tak budeme zkoumat, zda existuje i cesta z j do k}
```

Tento úsek programu obsahuje 3 do sebe vnořené cykly pro hodnoty parametru od 1 do $n$; to znamená, že doba jeho provádění bude $O\left(n^3\right)$.

## 8.3 Násobení matic: Strassenův algoritmus

Obvyklý algoritmus pro násobení matic je založen na definici této operace. Jsou-li $A$ a $B$ matice typu $n \times n$ a je-li

$$C = AB, \tag{8.4}$$

je $i,j$-tý prvek matice $C$ dán vztahem

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{8.5}$$

Na výpočet $c_{ij}$ potřebujeme $n$ násobení a $n-1$ sčítání; na výpočet matice $C$ tedy potřebujeme $O\left(n^3\right)$ operací. Kromě prostoru na uložení matic $A$, $B$ a $C$ nepotřebujeme prakticky žádný další paměťový prostor.

### 8.3.1 Rozdělení matice na bloky

Předpokládejme nyní, že počet řádků a sloupců matic vyhovuje podmínce

$$n = 2^k. \tag{8.6}$$

Pokusíme se použít techniky *rozděl a panuj*. Matice $A$, $B$ a $C$ rozdělíme na čtvercové bloky o velikosti $n/2 = 2^{k-1}$, takže násobení (8.4) bude mít tvar

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Snadno se přesvědčíme, že pro bloky $C_{ij}$ platí

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}. \tag{8.7}$$

To znamená, že potřebujeme 8 násobení bloků a 4 sčítání. Protože velikost bloků je $2^k$, můžeme bloky při násobení opět rozdělit na čtvercové bloky atd.; nakonec dospějeme k násobení bloků o velikosti 1 - tedy k násobení čísel.

Označme $S(n)$ počet operací, potřebný k násobení dvou matic $n \times n$. Protože na sečtení dvou matic typu $n \times n$ potřebujeme $O\left(n^2\right)$ operací, dostaneme, že posloupnost $S(n)$ vyhovuje rekurentnímu vztahu

$$S(n) \leq 8S\left(\frac{n}{2}\right) + qn^2 \text{ pro } n \geq 2, \tag{8.8}$$

$S(n) = 1$ pro $n = 1$ ($q$ je konstanta).

S podobným rekurentním vztahem se setkáme ještě jednou, a proto jej vyšetříme obecněji. Budeme zkoumat posloupnost $S(n)$, pro kterou platí

$$S(1) = d,$$

$$S(n) \leq aS(n/c) + bn^2, \tag{8.9}$$

kde $a$, $b$, $c$ a $d$ jsou kladná celá čísla a $n = c^k$. Ze vztahů (8.9) plyne postupným dosazováním

$$S(c) \leq ad + bc^2 = bc^2\left(1 + \frac{ad}{bc^2}\right);$$

$$S\left(c^2\right) \leq abc^2\left(1 + \frac{ad}{bc^2}\right) + bc^4 = bc^2\left(1 + \frac{a}{c^2} + \frac{a^2 d}{bc^4}\right);$$

Matematickou indukcí snadno dokážeme, že

$$S\left(c^k\right) \leq bc^{2k}\left(1+\frac{a}{c^2}+\cdots+\left(\frac{a}{c^2}\right)^{k-1}+\left(\frac{a}{c^2}\right)^k\frac{d}{b}\right). \tag{8.10}$$

Z (8.10) pak plyne, že pro $a > c$ a $n = c^k$ platí

$$S(n) = S\left(c^k\right) \leq bc^{2k}\left(\frac{\left(\frac{a}{c^2}\right)^k-1}{\frac{a}{c^2}-1}+\left(\frac{a}{c^2}\right)^k\frac{d}{b}\right) \leq bc^{2k}\left(\frac{\left(\frac{a}{c^2}\right)^k}{\frac{a}{c^2}-1}+\left(\frac{a}{c^2}\right)^k\frac{d}{b}\right) = \tag{8.11}$$

$$= \frac{a^k bc^2}{a-c^2}+a^k b\frac{d}{b} = a^k b\left(\frac{c^2}{a-c^2}+\frac{d}{b}\right) = c^{k\log_c a}b\left(\frac{c^2}{a-c^2}+\frac{d}{b}\right) = Kn^{\log_c a}. \tag{8.12}$$

V našem případě je $a = 8$, $c = 2$, takže na základě (8.12) dostáváme $S(n) \leq O\left(n^{\log 8}\right) = O\left(n^3\right)$. To znamená, že popsané rozdělení na bloky nejspíš nepřineslo žádný užitek (vzhledem k tomu, že jsme pracovali s nerovností, nemůžeme si dovolit ostřejší tvrzení).

### 8.3.2 Strassenův algoritmus

Strassenův algoritmus násobení matic je založen na postřehu, že bloky $C_{ij}$ můžeme počítat také na základě vztahů

$$\begin{array}{ll} C_{11} = P+S-T+V, & C_{12} = R+T \\ C_{21} = Q+S, & C_{22} = P-Q+R+U, \end{array} \tag{8.13}$$

kde

$$\begin{array}{ll} P = (A_{11}+A_{22})(B_{11}+B_{22}), & Q = (A_{21}+A_{22})B_{11}, \\ R = A_{11}(B_{12}-B_{22}), & S = A_{22}(B_{21}-B_{11}), \\ T = (A_{11}+A_{12})B_{22}, & U = (A_{21}-A_{11})(B_{11}+B_{12}), \\ V = (A_{12}-A_{22})(B_{12}+B_{22}). & \end{array} \tag{8.14}$$

Vztahy (8.13 a 8.14) obsahují celkem 7 násobení a 18 sčítání. To znamená, že složitost algoritmu násobení matic, založeného na tomto rozkladu, bude pro $n = 2^k$ dána nerovností

$$S(n) \leq 7S\left(\frac{n}{2}\right)+18\left(\frac{n}{2}\right)^2.$$

Jde opět o nerovnost tvaru (8.9), ve které máme $a = 7$, $b = 18$, $c = 2$. Dosazením dostaneme vztah $S(n) = O\left(n^{\log 7}\right) = O\left(n^{2,81}\right)$, neboť $\log_2 7 \approx 2,81$.

Praktický význam tento algoritmus ovšem příliš nemá. Numerické pokusy ukazují, že vzhledem k implementační složitosti výpočtu založeného na rozkladu (8.13 a 8.14) bude dosáhneme úspory času až při cca $n \geq 50$. Lze ukázat, že minimální spotřeba paměti bude také $O\left(n^{2,81}\right)$, takže pro $n$, pro který by mohla být významná úspora času, bude již hrát roli ztráta operační paměti.

## 8.4 Výpočet hodnoty polynomu

Výpočet funkčních hodnot polynomů patří mezi běžné operace. Podívejme se např. na výpočet hodnoty polynomu

$$p(x) = x^5+7x^4-12x^2+9x-25, \tag{8.15}$$

v nějakém bodě $x$. Ponechme stranou možnost vypočítat $x^5$, pak vypočítat a přičíst $7x^4$ atd. To je nejspíš nejhorší myslitelný postup, neboť při něm zbytečně opakujeme výpočty mocnin $x$. Poněkud rozumnější je postupovat od nižších mocnin $x$ a jednou vypočtené hodnoty si pamatovat. K výpočtu $x^4$ využijeme již vypočtené hodnoty $x^2$ atd.

Ještě výhodnější je použít *Hornerova schématu*. Postup výpočtu bude zřejmý, uzávorkujeme-li polynom (8.15) takto:

$$p(x) = x(x(x(x(x+7))-12)+9)-25,$$

Podobným způsobem můžeme uzávorkovat libovolný polynom daného stupně $n$. Odtud plyne postup výpočtu. Hodnotu koeficientu u $x^n$ (nejvyšší mocniny) vynásobíme $x$, přičteme koeficient u $x^{n-1}$, opět vynásobíme $x$ atd. Jsou-li koeficienty polynomu uloženy v poli $p$ o $n+1$ prvcích,

```
var p: array [0..n] of real;
```

můžeme zapsat výpočet podle Hornerova schématu takto:

```
s := p[n];
for i := n-1 downto 0 do s := s*x+p[i];
```

Při tomto výpočtu potřebujeme celkem $n$ násobení a $n-1$ sčítání.

Hornerovo schéma přestane být výhodné, je-li $n$ velké a polynom $p$ obsahuje velmi málo nenulových koeficientů. Typickým příkladem může být

$$p(x) = x^{55}.$$

Ukážeme, že není potřeba 54 násobení, ale podstatně méně. Rozložíme-li číslo 55 na součet mocnin dvou, dostaneme

$$x^{55} = x^{32}x^{16}x^4x^2x^1,$$

odkud je zřejmé, že stačí spočítat pouze $x^{32}$ (a průběžně si zapamatovat $x^2$, $x^4$ atd.). Přitom k výpočtu $x^{32}$ stačí 5 násobení, takže k výpočtu $x^{55}$ potřebujeme celkem 9 násobení.

## 8.5 Diskrétní Fourierova transformace

Mezi nejčastěji používané aritmetické algoritmy patří bezesporu *rychlá Fourierova transformace*. Tento algoritmus, který byl publikován v polovině 60. let [33], znamenal zrychlení klasického výpočetního postupu o řád.

### 8.5.1 Úvodní úvahy

Fourierova transformace je pro integrovatelnou funkci $f(t)$ definována vztahem

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)\, e^{2\pi i\omega t}\mathrm{d}t \tag{8.16}$$

a inverzní transformace má tvar

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)\, e^{-2\pi i\omega t}\mathrm{d}\omega. \tag{8.17}$$

Vzor, tedy funkce $f$, je komplexní funkce jedné reálné proměnné $t$. Tato proměnná se obvykle interpretuje jako čas. Obraz, tedy funkce $F(w)$, je opět komplexní funkcí jedné reálné proměnné, která se obvykle interpretuje jako *frekvence*. Fourierova transformace tedy představuje převod časové závislosti na závislost frekvenční. Frekvenční analýza také představuje jedno z nejčastějších použití této transformace v elektrotechnice.

Vedle toho může Fourierova transformace posloužit v matematické fyzice jako účinný nástroj při řešení některých úloh pro obyčejné i parciální diferenciální rovnice.

V praxi se ovšem místo funkcí, definovaných na celé množině reálných čísel, setkáváme s často s konečnými množinami diskrétních vzorků, tedy s konečnými posloupnostmi (obecně komplexních) čísel. Pro ně zavádíme *diskrétní Fourierovu transformaci* takto:

Diskrétním Fourierovým obrazem posloupnosti $a = \{a_0, a_1, \ldots, a_{n-1}\}$ je posloupnost $A = \{A_0, A_1, \ldots, A_{n-1}\}$, jejíž prvky jsou dány vztahy

$$A_j = \sum_{t=0}^{n-1} a_t e^{\left(\frac{2\pi ijt}{n}\right)},\ j = 0, \ldots, n-1; \tag{8.18}$$

inverzní transformace je dána vztahy[2]

[2]Důkaz vzorce pro zpětnou transformaci: Dosadíme za $A_j$ do výrazu na pravé straně (15b). Dostaneme

$$q = \frac{1}{n}\sum_{s=0}^{n-1} A_s \exp\left(\frac{2\pi isk}{n}\right) = \frac{1}{n}\sum_{s=0}^{n-1}\sum_{t=0}^{n-1} a_t \exp\left(\frac{2\pi ist}{n}\right)\exp\left(\frac{2\pi isk}{n}\right) = \frac{1}{n}\sum_{s=0}^{n-1}\sum_{t=0}^{n-1} a_t \exp\left(\frac{2\pi is\,(t-k)}{n}\right) =$$

$$= \frac{1}{n}\sum_{t=0}^{n-1} a_t \sum_{s=0}^{n-1}\exp\left(\frac{2\pi is\,(t-k)}{n}\right).$$

Snadno zjistíme, že pro $t = k$ bude vnitřní součet roven $n$. Pro $t$ $k$ dostaneme

$$\sum_{s=0}^{n-1}\exp\left(\frac{2\pi is\,(t-k)}{n}\right) = \frac{\exp\left(\frac{2\pi in(t-k)}{n}\right)-1}{\exp\left(\frac{2\pi i(t-k)}{n}\right)-1} = 0$$

Odtud již plyne $q = a_k$.

$$a_k = \frac{1}{n}\sum_{s=0}^{n-1} A_s e^{(-\frac{2\pi i s k}{n})},\ k = 0, \ldots, n-1. \tag{8.19}$$

Označíme-li

$$w = e^{(\frac{2\pi i}{n})},$$

představují vztahy (8.18) výpočet hodnot polynomu

$$a(x) = \sum_{j=0}^{n-1} a_j x^j$$

v bodech $x = w^j$ a vztahy (8.19) výpočet hodnot polynomu

$$A(x) = \frac{1}{n}\sum_{j=0}^{n-1} A_j x^j$$

v bodě $x = w^{-k}$. K výpočtu přímé i inverzní transformace bychom mohli použít např. Hornerova schématu, se kterým jsme se seznámili v předchozím oddílu. Vzhledem k tomu, že výpočet opakujeme pro $n$ různých hodnot a vyčíslení polynomu $a(x)$ resp. $A(x)$) tímto způsobem vyžaduje $O(n)$ kroků, vyžaduje výsledný algoritmus celkem $O\left(n^2\right)$ kroků.

Ukazuje se však, že lze využít zvláštních vlastností bodů $w^j$, ve kterých polynomy $a(x)$ resp. $A(x)$ počítáme. Číslo $w$ představuje hlavní hodnotu $n$-té odmocniny z 1, $w^n = 1$. Přidáme-li navíc požadavek, aby pro počet $n$ hodnot v posloupnosti $a = \{a_0, a_1, \ldots, a_{n-1}\}$ platilo $n = 2^k$, můžeme odvodit algoritmus, který bude vyžadovat pouze $O(n \log_2 n)$ operací.

### 8.5.2 Rychlá Fourierova transformace

Jak jsme již naznačili, budeme v tomto oddílu předpokládat, že je dána konečná posloupnost komplexních čísel $a = \{a_0, a_1, \ldots, a_{n-1}\}$, kde $n = 2^k$. Symbolem $w$ zde označujeme hlavní hodnotu $n$-té odmocniny z 1, tedy číslo

$$w = e^{(\frac{2\pi i}{n})} = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right) \tag{8.20}$$

Z toho, že $w^n = 1$, plyne, že $w^k = w^{(k \bmod n)}$. To znamená, pro že jak pro přímou tak i pro inverzní transformaci vystačíme se znalostí $w^0, w^1, w^2, \ldots, w^{n-1}$. (Odtud také plyne, že $\left(w^k\right)^n = 1$, takže i číslo $w^k$ představuje - pro libovolné $k \in$ Z - odmocninu z 1.)

Začneme u přímé transformace. V předchozím odstavci jsme si ukázali, že $w^k A_j = a\left(w^j\right)$, kde $a(x)$ je polynom s koeficienty danými posloupností $a$. Při výpočtu hodnoty tohoto polynomu se pokusíme použít metodu rozděl a panuj. Koeficienty polynomu $a$ rozdělíme na dvě skupiny - na sudé a liché, které označíme $a_s$ a $a_l$. Tím rozložíme polynom $a$ na součet dvou polynomů,

$$a(x) = a_s\left(x^2\right) + x a_l\left(x^2\right). \tag{8.21}$$

Úlohu vyčíslit polynom $a$ stupně $n-1$ v bodě $x$ jsme tedy převedli na úlohu vyčíslit dva polynomy $a_s$ a $a_l$ stupně $n/2-1$ v bodě $x^2$.

Jak víme, potřebujeme při výpočtu hodnoty $A_k$ vyčíslit (8.21) v bodě $x = w^k$, kde $w$ je hlavní hodnota $n$-té odmocniny z 1. Z (8.20) ale plyne, že $w^2$ je hlavní hodnota $(n/2)$-té odmocniny z 1 - a to je přesně to, co k úspěšnému použití metody *rozděl a panuj* potřebujeme. Vztah (8.21) totiž říká, že místo abychom počítali hodnotu polynomu a v $n$-té odmocnině z jedné, můžeme počítat hodnoty polynomů $a_s$ a $a_l$ v $(n/2)$-té odmocnině z 1.

Protože platí $n = 2^k$, můžeme použít týž postup i na polynomy $a_s$ a $a_l$. Rekurzivně tak dospějeme až k polynomům 1. stupně, které budeme vyčíslovat v bodech 1 a -1. Při výpočtu můžeme navíc využít skutečnosti, že $w^{(n/2)} = -1$, takže $w^i = -w^{(n/2)+i}$.

## Příklad 8.2

Podívejme se na polynom 8. stupně,

$$\begin{aligned} p(x) &= p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 + p_5x^5 + p_6x^6 + p_7x^7 = \\ &= p_0 + p_2x^2 + p_4x^4 + p_6x^6 + x\left(p_1 + p_3x^2 + p_5x^4 + p_7x^6\right) = \\ &= p_s\left(x^2\right) + xp_l\left(x^2\right). \end{aligned}$$

Je-li nyní

$$w = w_8 = e^{\left(\frac{2\pi i}{8}\right)} = \cos\left(\frac{\pi}{4}\right) + i\sin\left(\frac{\pi}{4}\right),$$

dostaneme

$$\begin{aligned} p\left(w_8^0\right) &= p_s\left(w_4^0\right) + w_8^0 p_l\left(w_4^0\right), & p\left(w_8^1\right) &= p_s\left(w_4^1\right) + w_8^1 p_l\left(w_4^1\right), \\ p\left(w_8^2\right) &= p_s\left(w_4^2\right) + w_8^2 p_l\left(w_4^2\right), & p\left(w_8^3\right) &= p_s\left(w_4^3\right) + w_8^3 p_l\left(w_4^3\right), \\ p\left(w_8^4\right) &= p_s\left(w_4^0\right) + w_8^0 p_l\left(w_4^0\right), & p\left(w_8^5\right) &= p_s\left(w_4^1\right) + w_8^1 p_l\left(w_4^1\right), \\ p\left(w_8^6\right) &= p_s\left(w_4^2\right) + w_8^2 p_l\left(w_4^2\right), & p\left(w_8^7\right) &= p_s\left(w_4^3\right) + w_8^3 p_l\left(w_4^3\right). \end{aligned}$$

Při výpočtu uložíme transformovanou posloupnost do pole $p$ a délce $n$. V tomto poli také dostaneme výsledek.

Algoritmus přímé transformace, tedy vyčíslení polynomu s koeficienty $a_0, a_1, \ldots, a_{n-1}$, můžeme popsat následujícími kroky:

1. Je-li stupeň polynomu $n = 1$, jsou hledané hodnoty rovny $A_0 = a_0 + a_1$ a $A_1 = a_0 - a_1$ a skončíme.
2. Je-li stupeň polynomu $n > 1$, přerovnáme pole $a$ v pořadí $a_0, a_2, \ldots, a_{n-2}, a_1, a_3, \ldots, a_{n-1}$.
3. Použijeme tento algoritmus rekurzivně zvlášť pro první polovinu pole $a$, se sudými koeficienty a zvlášť pro druhou polovinu pole s lichými koeficienty.
4. Pole, získané ve 3. kroku, obsahuje obrazy polynomů $a_s$ a $a_l$. Přerovnáme je do původního pořadí a složíme z něj podle (8.21) obraz původního pole.

Program, který popsaný algoritmus implementuje, napíšeme v C++, neboť tento jazyk jako jeden z mála nabízí programátorům jak rekurzi tak i typ **complex** pro práci s komplexními čísly.

Následující procedura *vypočti* bude při přerovnávání používat pomocné pole $t$. Deklarujeme je jako globální, stejně jako některé další pomocné proměnné. Další pomocné pole $w$ bude obsahovat hodnoty $w_0, w_1, w_2, \ldots, w_{n-1}$ (mocniny $n$-té odmocniny z 1, viz (8.20)).

Prvním vstupním parametrem procedury *vypočti* je pole $a$, které obsahuje koeficienty transformovaného polynomu (tedy transformovanou posloupnost). Druhý parametr $N$ obsahuje stupeň transformovaného polynomu. Předpokládáme, že pro hodnotu parametru $N$ platí $N = 2^r - 1$ pro nějaké $r$. (To znamená, že pole $a$ obsahuje $N + 1$ prvků.) třetím parametrem je index $k$, od kterého se máme polem $a$ zabývat.

```
#include <complex.h>
#define M 8                          // Velikost pole - např. 8...
#define PI 3.14159265358979

complex w[M], t[M];                              // Pomocná pole
complex p0, p1;                                  // Pomocné proměnné
complex q[M];                                    // Transformované pole
void vypočti(complex a[ ], int N, int k){
 int i, j;
 if(N==1) {                        // Je-li stupeň polynomu N == 1,
  p0=a[k]; p1=a[k+1];                            // vypočteme hodnotu
  a[k]=p0+p1;
  a[k+1]=p0-p1;
 } else {                          // Jinak pole nejprve přerovnáme
  for(i=0; i <= N/2; i++) {                      // (s pomocí pole t)
```

```
   j = k+2*i;
   t[i] = a[j];
   t[i+1+N/2] = a[j+1];
  }                                // a pak zavoláme tuto proceduru
  for(i = 0; i <= N; i++) a[k+i] = t[i];           // rekurzivně
  vypočti(a, N/2, k);               // zvlášt' pro první  polovinu
  vypočti(a, N/2, k+1+N/2);       // a zvlášt' pro druhou polovinu
  j = M/(N+1);
  for(i=0; i <= N/2; i++) {          // Pak pole přerovnáme zpět
   p0 = w[i*j]*a[k+(N/2)+1+i];                // a složíme z něj
   t[i] = a[k+i] + p0;                    // obraz původního pole
   t[i+(N/2)+1] = a[k+i] - p0;
  }
  for(i=0; i <= N; i++) a[k+i] = t[i];
 }
}
```

Před prvním voláním procedury vypočti musíme vypočítat hodnoty $w^j$, $j = 0, 1, \ldots, n-1$. Procedura pro přímou transformaci proto může mít následující tvar:

```
void FFTR(complex a[ ], int N) {
 static jeW = 0;            // Indikuje, zda již bylo w vypočteno
 if(!jeW){
   jeW++;
   w[0]=1;
   w[1] = complex(cos(2*PI/N), sin(2*PI/N));
   for(int i = 2; i < N; i++) w[i] = w[i-1]*w[1];
 }
 vypočti(a,N-1,0);
}
```

Inverzní transformace spočívá podle (8.19) ve vyčíslení polynomu s koeficienty $A_k$ v bodech $w^{-j}$, kde $j = 0, 1, \ldots, n-1$ a vydělení výsledku číslem $n$. Snadno se ale přesvědčíme, že pro $n$-tou odmocninu z 1 platí $w^{-0} = w^0 = 1$, $w^{-1} = w^{n-1}, \ldots, w^{-(n-1)} = w^{-1}$. To znamená, že stačí použít opět funkci vypočti a výsledek přerovnat. Funkce pro inverzní transformaci proto bude mít tento tvar:

```
void IFFTR(complex a[ ], int N) {
 vypočti(p, N-1, 0);
 for(int i = 1; i <= N/2; i++) {                    // Přerovnání
  complex t = a[i]; a[i] = a[N-i]; a[N-i]= t;
 }
 for(i=0; i < N; i++) a[i] /=N;
}
```

Pole $q$, deklarované před funkcí *vypočti*, transformujeme příkazem

```
FFTR(q,M);
```

a zpětnou transformaci obstará příkaz

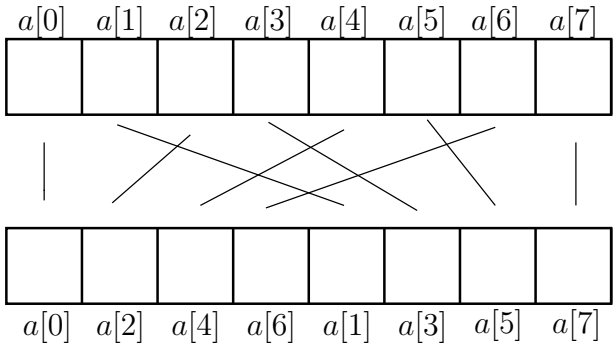
```
IFFTR(q,M);
```

### 8.5.3 Složitost rychlé Fourierovy transformace

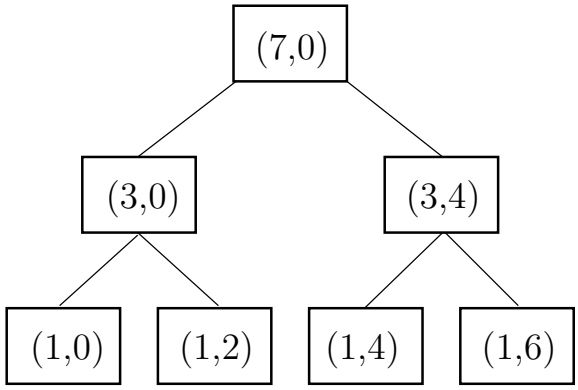
Nejprve se vrátíme k příkladu transformace pole o 8 prvcích, tedy polynomu 7. stupně.

**Příklad 8.2 (pokračování)**

Při výpočtu $A_0$ převedeme výpočet hodnoty polynomu 7. stupně na výpočet hodnot dvou polynomů 3. stupně. Použijeme k tomu proceduru *vypočt*i - zavoláme ji s parametry a, $N = 7$ a $k = 0$ (bereme pole $a$ od nultého




Obr. 8.2: Přesun prvků při rychlé Fourierově transformaci pro $n = 8$




Obr. 8.3: Druhý a třetí parametr rekurzivních volání procedury *vycisli*

prvku). Tato procedura nejprve přerovná pole $a$ tak, aby v prvcích $a_0, \ldots, a_3$ ležely koeficienty u sudých mocnin $a$ v $a_4, \ldots, a_7$ koeficienty u lichých mocnin původního polynomu (viz obr. 8.1).

Pak zavoláme proceduru *vypočti* rekurzivně pro vyčíslení polynomů $a_s$ a $a_l$. Při vyčíslování $a_s$ bude mít volání procedury vypočti parametry $a$, $N = 3$ (stupeň polynomu) a $k = 0$ (koeficienty jsou uloženy v poli $a$ od nultého prvku). Při vyčíslování $a_l$ bude mít volání procedury vypočti parametry $a$, 3 (stupeň polynomu, stejný jako u $a_s$) a 4 (koeficienty $a_l$ jsou uloženy v poli $a$ od prvku $a\,[4]$).
Vyčíslování $a_s$ a $a_l$ bude znamenat opět dvě rekurzivní volání procedury *vycisli*. Posloupnost rekurzivních volání můžeme vyjádřit stromem, který vidíte na obr. 8.3

Je zřejmé, že hloubka rekurze bude rovna $\log_2 n$, v případě $n = 8$ tedy 3.

Složitost rekurzivního algoritmu rychlé Fourierovy transformace odvodíme takto: Je-li $T\,(n)$ počet operací, potřebný při daném $n$, musí platit

$$T\,(n) = 2T\,(n/2) + cn,$$

kde $c$ je konstanta taková, že $cn$ je čas potřebný na obě přerovnání pole o $n$ prvcích. Protože $T\,(1) = d$ je také konstanta, dostaneme

$$T\,(2^m) = 2T\left(2^{m-1}\right) + c2^m = 4T\left(2^{m-2}\right) + 2c2^m = \cdots cm2^m + T\,(1)\,2^m = cn\log_2 n + dn,$$

odkud plyne složitost $O\,(n\log_2 n)$.

# Kapitola 9

# Softwarový projekt

V této kapitole si krátce povíme o hlavních zásadách konstrukce software.

Zkušenosti ukazují, že práci na velkých softwarových projektech je výhodné maximálně formalizovat[1]. Cílem takové formalizace je samozřejmě vyhnout se nejen chybám, ale i nepřesnostem ve formulaci projektu a nedorozuměním v kontaktu se zákazníkem.

Studie, prováděné např. ministerstvem obrany USA, ukazují, že poměr dodatečných nákladů, vyvolaných změnou v počáteční fázi a nákladů, vyvolaných změnou v koncových fázích projektu, může být i vyšší než 1:200. Tabulku, která ukazuje relativní náklady na opravu chyby v závislosti na tom, kdy chyba vznikne a kdy my ji odhalíme, jsme převzali ze [7].

| | Okamžik vzniku | | |
|---|---|---|---|
| Okamžik odhalení | Požadavky | Návrh | Psaní programu |
| Analýza | 1 | - | - |
| Návrh | 2 | 1 | - |
| Pasivní testy | 5 | 2 | 1 |
| Strukturární testy | 15 | 5 | 2 |
| Funkcionální testy | 25 | 10 | 5 |

Tabulka 9.1 Výše relativních nákladů způsobených chybou v závislosti na tom, kdy chyba vznikne a kdy ji odhalíme.

V následujícím oddílu si povíme o jednotlivých fázích vývoje softwarového produktu.

## 9.1 Životní cyklus softwarového produktu

Životní cyklus softwarového produktu můžeme z hlediska vývojáře rozdělit do následujících fází:

1. Definice problému.
2. Analýza požadavků
3. Návrh architektury
4. Podrobný návrh
5. Programování a ladění
6. Testování modulů
7. Testování systému
8. Údržba

Na některé z nich se nyní podíváme podrobněji.

## 9.2 Definice problému

Na počátku je nezbytné definovat problém, který má budoucí softwarový produkt řešit. Popis problému má používat jazyk (terminologii) uživatele a má vycházet z jeho pohledu. Rozsah specifikace by neměl překročit jednu až dvě stránky.

[1] Ve smyslu úřednickém, nikoli matematickém. Jde o to dokumentovat jak požadavky zákazníka tak i postup projektu.

V popisu problému není vhodné naznačovat možná řešení, neboť jejich hledání zpravidla není problém zákazníka, nýbrž řešitele. Naznačujeme-li řešení již ve formulaci problému, můžeme řešitele zavést na scestí; rozhodnutí o způsobu řešení by mělo být až výsledkem analýzy problému.

Například tvrzení „Máme problémy s evidencí materiálu ve skladu“ nejen zní jako problém, můžeme je i považovat za rozumnou definici problému.

Na druhé straně věta „Potřebujeme optimalizovat automatické zadávání dat při evidenci materiálu ve skladu“ už nezní jako problém - zde je již vnucováno řešení. Takováto formulace může mít smysl až ve specifikaci předběžných požadavků.

Přitom je dobré uvědomovat si, že nejlepší řešení nemusí být vždy nový program nebo úprava programu, který už existuje. Jestliže např. používáme program, který umí udělat půlroční bilanci hospodaření, a potřebujeme roční bilanci, může být nejlacinějším a nejrychlejším řešením ručně sečíst jednou do roka deset čísel.

## 9.3 Předběžné požadavky

Formulace předběžných požadavků představuje první krok k řešení problému. Zde vlastně sestavíme podrobný popis toho, co se bude od softwarového produktu očekávat.

Ze zkušeností softwarových firem plyne, že je vhodné sestavit se zákazníkem písemný seznam předběžných požadavků. Tím mimo jiné dosáhneme toho, že o funkci programu bude rozhodovat zákazník, nikoli programátor.

Formální seznam požadavků umožní snáze se vyhnout chybám a nedorozuměním. Jestliže při psaní programu přijdeme na chybu v kódu, stačí zpravidla přepsat pár řádek. Přijdeme-li při psaní programu na chybu v požadavcích, musíme se k nim vrátit a musíme znovu projít návrh architektury, podrobný návrh a možná začít psát program úplně od začátku.

Adekvátní specifikace požadavků je proto klíčovým momentem úspěchu projektu.

Na druhé straně je předem jasné, že požadavky ze strany zákazníka se budou průběžně měnit. Často totiž teprve samotný proces vývoje produktu zákazníkovi umožní, aby si plně uvědomil, co vlastně potřebuje - a podle toho bude své požadavky měnit. (Formálně sepsané požadavky nám pak umožní naúčtovat mu to...)

Studie IBM ukazují, že u typického projektu se v průběhu vývoje změní požadavky přibližně z 25% [7]. Proto je důležité, aby zákazník znal cenu změn a abychom si s ním dohodli formální postup při změnách z jeho strany.

### 9.3.1 Kontrola seznamu požadavků

Při sestavování seznamu předběžných požadavků by se mohlo stát, že opomeneme některou důležitou oblast. Následující seznam kontrolních otázek je sestaven na základě praktických zkušeností (podle [7]) a má nám pomoci vyhnout se nedopatřením. Samozřejmě ne všechny tyto otázky musí mít v souvislosti s konkrétním problémem smysl.

Následující otázky lze použít také jako vodítko při sestavování seznamu požadavků. Rozdělíme si je do několika okruhů.

**Obsah požadavků**

- Specifikovali jsme všechny vstupní datové proudy systému? Známe jejich zdroje? Známe přesnost vstupních dat? Známe rozsahy hodnot? Známe frekvence vstupů?
- Specifikovali jsme všechny výstupní datové proudy? Známe jejich příjemce, rozsah hodnot dat a jejich přesnost, frekvenci? Známe jejich formát?
- Specifikovali jsme formáty všech sestav?
- Známe specifikace všech vnějších hardwarových i softwarových rozhraní připravovaného produktu?

- Známe potřebná komunikační rozhraní, jako jsou komunikační protokoly, přenosové rychlosti, způsoby kontroly chyb apod.?
- Jsou z hlediska uživatele specifikovány potřebné doby odezvy pro všechny nezbytné operace?
- Známe případná další časová omezení, jako je doba zpracování, doba přenosu dat apod.?
- Specifikovali jsme všechny úlohy, které má systém podle představ uživatele provádět?
- Specifikovali jsme data, která se budou používat v jednotlivých úlohách a která budou výsledkem jednotlivých úloh?
- Specifikovali jsme požadavky na úroveň utajení? Specifikovali jsme úroveň zabezpečení dat před ztrátou nebo zničením (může jít o zabezpečení před sabotáží, před počítačovými viry, před nechtěným smazáním, před pádem meteoritu na počítač...)?
- Specifikovali jsme požadavky na úroveň spolehlivosti - včetně možných důsledků selhání programu? Specifikovali jsme, které informace jsou životně důležité a které je nezbytně třeba chránit? Specifikovali jsme strategii detekce chyb a zotavování se z nich?
- Specifikovali jsme maximální požadavky na paměť RAM?
- Specifikovali jsme maximální požadavky na vnější paměťová média (disky, magnetické pásky apod.)?
- Vyjasnili jsme požadavky na údržbu a udržovatelnost systému? Známe požadavky na možnost přenosu programového produktu do jiného operačního prostředí, pod jiný operační systém? Známe požadavky na komunikaci s jinými softwarovými produkty (např. OLE / DDE pod MS Windows)? Jak je to s případnými změnami přesnosti vstupů/výstupů? Co v případě požadavků na změnu výkonnosti?
- Jaké jsou vztahy mezi vlastnostmi, které si mohou navzájem konkurovat nebo se i vylučovat? Dostane např. přednost rychlost před přesností, velikost programu před robustností?
- Známe přesnou specifikaci úspěšného zpracování nebo chyby?

**Úplnost požadavků**

- Pokud neznáme potřebné informace před začátkem zpracování, známe alespoň rozsah jejich neúplnosti?
- Jsou požadavky úplné v tom smyslu, že pokud bude produkt vyhovovat opravdu všem uvedeným požadavkům, bude pro zákazníka přijatelný?
- Působí některé z požadavků potíže? Jeví se některé z nich jako neimplementovatelné a zařadili jste je jen proto, abyste uspokojili zákazníka (případně šéfa)?

**Kvalita požadavků**

- Jsou požadavky sepsány v jazyku uživatele? Myslí si to uživatel? Rozumí jim?
- Jsou požadavky bezkonfliktní? Nebo jsme se alespoň při jejich specifikaci vyhnuli konfliktům v maximální možné míře?
- Vyhnuli jsme se tomu, aby požadavky určovaly návrh architektury?
- Jsou všechny požadavky na přibližně stejné úrovni podrobnosti? Není třeba specifikovat některé z požadavků podrobněji? Nebo jsou naopak příliš podrobné?
- Jsou požadavky formulovány natolik jasně, aby je šlo implementovat? Porozumí jim i člověk, který o nich se zákazníkem nejednal?
- Jsou jednotlivé požadavky v rozumném vztahu k problému a k jeho řešení? Je jasný původ jednotlivých požadavků v kontextu výchozího problému?
- Lze testovat splnění jednotlivých požadavků? Může o splnění jednotlivých požadavků rozhodnout nezávisle zkonstruovaný a provedený test?
- Specifikovali jsme u jednotlivých požadavků všechny možné změny (případně včetně pravděpodobnosti takové změny)?

## 9.4 Návrh architektury

Po vyjasnění požadavků přijde na řadu *návrh architektury*; někdy také hovoříme o *strategii implementace*. Jde o návrh „na nejvyšší úrovni“, ve které se budeme zabývat především těmito aspekty budoucího produktu:

### Organizace programu

Nejprve stanovíme *organizaci programu*. To znamená, že vypracujeme přehled, který popíše programový produkt pomocí obecných pojmů. V něm určíme hrubý smysl jednotlivých modulů (hlavních částí programu) a jejich rozhraní.

Přitom je vhodné již v této fázi vývoje dbát na to, aby na sobě jednotlivé moduly byly co nejméně závislé, tj. aby jejich rozhraní byla pokud možno nejužší.

### Zacházení se změnami

Stanovíme strategii *zacházení se změnami* v projektu - může se jednat o čísla verzí, o návrhy záložních datových polí pro další použití, o soubory pro přidávání dalších tabulek apod.

Je vhodné také je určit způsob, jakým se bude v projektu zacházet s rozhodováním. Místo použití „tvrdě“ implementovaných rozhodnutí můžeme použít rozhodování podle tabulek; data, podle kterých se rozhodujeme, mohou být uložena v externích souborech atd.

### Vyvíjet či koupit

Již v této fázi se také musíme rozhodnout, zda budeme program sami vyvíjet či zda jej koupíme, případně které části koupíme a které budeme sami vyvíjet. Podobně se musíme rozhodnout, které části implementujeme sami a které převezmeme z operačního prostředí (například zda použijeme grafickou knihovnu Borland C++ nebo zda si napíšeme svoji, zda použijeme standardních služeb operačního systému nebo zda je obejdeme, protože nejsou účinné nebo dostatečně bezpečné, zda využijeme standardních ovladačů zařízení či zda vyvineme vlastní atd.).

Pokud se rozhodneme použít koupený software, musíme si ujasnit, jak vyhovuje ostatním požadavkům.

### Hlavní datové struktury

V dalším kroku specifikujeme *hlavní datové struktury*. Přitom musíme určit především jejich obsah a způsob zobrazení. Doporučuje se dokumentovat alternativy, které jsme při výběru zvažovali, a důvody, proč jsme se nakonec rozhodli určitým způsobem.

Při návrhu hlavních datových struktur bychom měli vycházet z požadavku skrývání dat - to znamená pokud možno se vyhýbat globálním datům. K datům by měl mít přístup vždy pouze jeden modul; pokud je potřebují i jiné moduly, musí použít přístupových procedur.

Při návrhu architektury modulů se uplatňuje „zákon zbytečných dat“: *Údaje, které vstoupí do programu (modulu, funkce ...) a neovlivní jeho výstup nebo chování, jsou zbytečné.*

V objektově orientovaném návrhu musíme stanovit hlavní třídy objektů, jejich vzájemné vztahy (dědické hierarchie), hlavní instance, jejich stavy, funkci a také doby trvání, tj. rozmezí jejich existence v programu.

### Hlavní algoritmy

Současně s návrhem hlavních datových struktur přijdou na řadu návrhy hlavních algoritmů (odkazy na ně). Také zde je třeba dokumentovat možné alternativy a důvody svých rozhodnutí.

### Všeobecné problémy

Další aspekt, kterým se ve fázi návrhu architektury musíme zabývat, jsou všeobecné problémy, které nezávisí na konkrétní podobě řešeného problému. Jde především o uživatelské rozhraní programu, řízení vstupních a výstupních operací, správu paměti, ale také třeba o zacházení se znakovými řetězci.

Uživatelské rozhraní může být specifikováno již v požadavcích. Pokud ne, musíme určit struktury příkazů, menu, vstupních formulářů apod. Přitom modulární složení programu je třeba navrhnout tak, aby přechod k jinému rozhraní neovlivnil jeho funkčnost.

Jako příklad uvedeme např. překladače Turbo Pascalu; ty jsou zpravidla dodávány dva, jeden integrovaný v programovém prostředí a druhý, který lze spouštět samostatně z příkazové řádky. Samozřejmě jde o týž program s jiným uživatelským rozhraním - jiný postup by byl ekonomicky neúnosný.

Architektura by měla specifikovat úroveň, na které budou detekovány a ošetřovány chyby při vstupních / výstupních operacích, bezpečnostní opatření (pořizování záložních kopií souborů) apod.

V této fázi vývoje bychom také měli odhadnout potřebnou velikost paměti v běžných a v mimořádných případech. Například pokud bude výsledkem databáze, měli bychom odhadnout paměť potřebnou pro jeden záznam a případně předpokládaný počet záznamů v databázi. Je třeba ukázat, zda jsou požadavky naší aplikace v mezích možností předpokládaného operačního prostředí.

Ve složitějších případech může aplikace mít vlastní systém pro správu paměti.

Může se zdát, že věnovat pozornost zacházení se znakovými řetězci již při návrhu architektury je zbytečné. Komerčně využívané programy ovšem obsahují velká množství řetězců (menu, nápovědy, texty v dialogových oknech a jiné); uživatel si může přát některé z nich změnit, přizpůsobit svým potřebám nebo svému vkusu. Při prodeji produktu do zahraničí je třeba všechny textové řetězce přeložit.

Proto je třeba odhadnout množství textových řetězců, které budou součástí programu, a rozhodnout se, jak s nimi naložíme. Lze je uložit do zvláštních souborů (to se obvykle dělá s nápovědou) nebo je třeba definovat jako konstanty ve zvláštním modulu. V programech pro MS Windows je můžeme popsat v tabulce řetězců (stringtable) v popisu prostředků programu (*resources*, soubor .RC). Tím získáme možnost znakové řetězce snadno měnit, aniž by to mělo nějaký závažnější dopad na zdrojový kód programu.

Lze je také samozřejmě deklarovat přímo na místě použití a nevzrušovat se případnými problémy se změnami - záleží na požadavcích a na naší předvídavosti. V návrhu architektury bychom se měli rozhodnout pro některou z možností a zaznamenat důvody, proč jsme se tak rozhodli.

**Zpracování chyb**

Zacházení s chybami patří k nejožehavějším problémům v moderním programování. Existují odhady, které tvrdí, že téměř 90% kódu v běžných programech se zabývá ošetřováním výjimečných situací. Při rozhodování o architektuře programu musíme také určit, jak bude náš produkt nakládat s chybovými situacemi. Jde především o odpověď na následující otázky:

- *Použijeme spíše korektivního nebo spíše detektivního přístupu*? Korektivní přístup znamená, že pokud nastane chyba, pokusíme se ji napravit. Při detektivním přístupu můžeme rovnou skončit - můžeme se ale také tvářit, jako by se nic nestalo, a doufat, že se opravdu nic neděje. V obou případech by ale program měl uživateli sdělit, jaké problémy nastaly.

- *Budeme detekovat chyby aktivně nebo pasivně*? Jinými slovy, budeme se snažit předvídat chyby uživatele a např. testovat správnost vstupních dat (třeba kontrolovat, zda uživatel nepožaduje výpis ze dne 31. února), nebo se budeme o chyby starat až v případě, že se nic jiného nedá dělat - např. když hrozí dělení nulou nebo když k němu dokonce už došlo?

- Aktivní detekce chyb klade značné nároky na uživatelské rozhraní. At' se rozhodneme pro kteroukoli alternativu, měl by program v obou případech včas informovat uživatele, co se děje.

- *Jak se v programu chyby šíří*? Jak bude program reagovat na vznik chyby? Podle okolností můžeme zvolit některou z následujících alternativ: můžeme ihned odmítnout data, která chybu způsobila; můžeme ihned přejít ke zpracování chyby; můžeme pokračovat v původní operaci až do dokončení všech procesů a teprve pak informovat uživatele, že někde nastala chyba.

- *Jaké konvence budeme používat pro ohlašování chyb*? Je rozumné, aby všechny moduly používaly při ohlašování chyb totéž rozhraní, jinak bude program působit zmateně.

- *Na jaké úrovni budou chyby v programu ošetřovány*? Můžeme se jimi zabývat v místě vzniku, můžeme pro ně volat zvláštní podprogramy nebo se jimi zabývat „na nejvyšší úrovni“. Možnosti nápravy chyby (zotavení programu po chybě) se samozřejmě liší podle úrovně, na které chybu detekujeme. Pokud chybu zachytíme až na úrovni celého programu, nemůžeme obvykle dělat nic jiného, než oznámit, co se stalo, a skončit.
- *Jaká je zodpovědnost jednotlivých modulů za správnost jeho vstupních dat*? Bude se každý z modulů starat o správnost svých dat, nebo budeme mít speciální moduly, které se postarají o správnost dat pro celý systém? Mohou moduly na nějaké úrovni předpokládat, že dostávají bezchybná data?

**Robustnost**

Pod robustností se obvykle chápe schopnost systému pokračovat v činnosti poté, co došlo k chybě. Při návrhu architektury je rozumné určit úroveň robustnosti z několika hledisek.
Především musíme určit míru pečlivosti, kterou při zpracování požadujeme. Obvykle se od jednotlivých modulů požaduje vyšší robustnost než od celého systému. Zkušenost totiž ukazuje, že softwarový produkt je zpravidla podstatně slabší než jeho nejslabší část.

Dále je třeba určit rozsah používání testovacích příkazů, jako je makro assert v jazyku C. Tyto příkazy testují, zda je splněna nějaká podmínka, plynoucí z předpokladů (můžeme například ověřovat předpoklad, že vstupní soubor nebude větší než 64 KB; pokud ano, ohlásí program chybu).

Vedle toho je třeba stanovit míru tolerance k chybám a způsob reakce na ně. Podívejme se na některé jednoduché možnosti.

Systém se např. může při výskytu chyby pokusit vrátit k místu, kde ještě bylo vše v pořádku, a pokračovat odtud. Může se pokusit použít pro zpracování daného problému jinou funkci (např. jestliže se při řešení soustavy lineárních algebraických rovnic zjistí, že soustava má špatně podmíněnou matici a že tedy nelze použít Gaussovu eliminaci, může se systém pokusit použít některou z iteračních metod).

Systém také může pro řešení rovnou použít několika metod, získané výsledky porovnat a podle zadané tolerance pak určit, který výsledek použije nebo zda použije např. jejich aritmetický průměr.

Hodnotu, kterou systém určí jako špatnou, se může pokusit nahradit hodnotou, která nebude mít katastrofické důsledky. Kromě toho může systém přejít do stavu, ve kterém není schopen provádět všechny své funkce, může skončit a případně se znovu spustit.

Systém by také měl brát v úvahu možnost, že sám obsahuje chyby, a měl by (v omezené míře) prověřovat své vlastní výsledky, aby důsledky těchto chyb omezil.

**Výkonnost**

Pod výkonností můžeme chápat jak rychlost tak i paměťové nároky. Pokud je výkonnost systému důležitá, je třeba specifikovat její kritéria.

Ve stádiu návrhu architektury bychom měli specifikovat odhad výkonnosti a zdůvodnit, proč se domníváme, že takovéto výkonnosti lze dosáhnout. Měli bychom specifikovat také oblasti, ve kterých hrozí nebezpečí, že požadovaného výkonu nedosáhneme.

Pokud je k dosažení výkonnosti třeba použít speciálních algoritmů nebo datových struktur, je třeba to zdůraznit již v tomto stadiu. Zde je vhodné také specifikovat předpokládané časové a paměťové nároky jednotlivých modulů.

**Celková kvalita návrhu architektury**

Pro kvalitu návrhu architektury lze vymyslet řadu kritérií. Zejména je samozřejmé, že návrh musí jasně vymezit a zdůvodnit cíle projektu a použité metody. Musí být jasný a přehledný, musí přesně specifikovat nebezpečné oblasti a určit zacházení s nimi.

Ten, kdo bude navrženou architekturu implementovat, musí návrhu beze zbytku rozumět.

### 9.4.1 Kontrola návrhu architektury

Podobně jako v případě prověřování úplnosti předběžných požadavků, i v případě návrhu architektury má smysl sepsat si kontrolní otázky a podle nich hledat, na co jsme mohli zapomenout. Následující seznam opět vychází z [7].

- Je jasná celková organizace programu, včetně přehledů a zdůvodnění architektury?
- Jsou moduly dobře definovány, je jasná jejich funkce, jsou jasná jejich rozhraní s jinými moduly?
- Jsou pokryty všechny funkce, uvedené v seznamu požadavků? Jsou pokryty rozumně, tj. není jim věnováno příliš málo nebo příliš mnoho modulů?
- Je architektura navržena tak, aby se mohla vyrovnat se kteroukoli z pravděpodobných změn?
- Obsahuje návrh všechna rozhodnutí, zda určitý produkt (nebo jeho součást) koupit nebo vyvíjet?
- Popisuje návrh, jak se přizpůsobí knihovní funkce a jiné opakovaně používané programové součásti, aby vyhovovaly našemu produktu?
- Jsou v návrhu popsány a zdůvodněny všechny hlavní datové struktury?
- Lze s hlavními datovými strukturami zacházet pouze prostřednictvím přístupových funkcí?
- Je specifikován obsah a organizace databází?
- Popsali a zdůvodnili jsme všechny hlavní algoritmy?
- Popsali jsme strategii při zacházení se vstupními a výstupními operacemi (včetně vstupů a výstupů uživatelských)?
- Definovali jsme klíčové aspekty uživatelského rozhraní?
- Je modulární struktura uživatelského rozhraní taková, že neovlivňuje zbytek programu?
- Popsali a zdůvodnili jsme strategii správy paměti a odhady paměťových nároků?
- Odhadli jsme časové a paměťové nároky jednotlivých modulů?
- Obsahuje návrh strategii pro zacházení s řetězci a odhad potřebného místa pro ně?
- Obsahuje návrh vhodnou strategii pro ošetřování chyb? Jsou chybová hlášení konzistentní s uživatelským rozhraním?
- Specifikovali jsme úroveň robustnosti?
- Formulovali jsme jasně hlavní cíle systému?
- Je návrh vyvážený? Není některá část zpracována příliš pečlivě a jiná nedbale?
- Je návrh architektury jako celek konzistentní? To znamená, je jasná souvislost jednotlivých částí?
- Je návrh na nejvyšší úrovni nezávislý na počítači a na programovacím jazyku?
- Jsou jasné důvody všech rozhodnutí?
- Klíčová otázka na závěr: Jste s návrhem spokojen jako programátor, který jej bude implementovat?

### 9.4.2 Programovací jazyk

Po vyjasnění architektury je třeba rozhodnout se pro vhodný programovací jazyk. Jeho volba může podstatným způsobem ovlivnit nejen výkonnost programu, ale také programátora.
Rozhodnutí může ovšem záviset na dostupných prostředcích, na požadavcích zákazníka, ale i na schopnostech a osobním vkusu programátora. Podívejme se na některé nejběžnější možnosti.

Pokud nám jde o maximální úspornost kódu a o výkonnost programu, obvykle volíme *asembler*.

Pro numerické výpočty se stále často používá *Fortran*; výhodou tohoto jazyka jsou obrovské knihovny podprogramů pro téměř všechny běžnější matematické, fyzikální a obecně technické problémy, se kterými se lze setkat.

V oblasti hromadného zpracování dat dlouhou dobu převládal jazyk *Cobol*. Nabízel řadu nástrojů, zaměřených na rozsáhlé datové soubory (např. příkazy pro třídění souborů, generátor sestav apod.). Již ve verzi *Cobol 70* se objevily prostředky pro paralelní programování. Na druhé straně neumožňoval ukrývání dat - veškerá data se deklarovala jako globální. Zdá se, že jej v současné době vytlačují databázové jazyky.

Pro komunikaci s databázemi se dnes často používá *SQL* (*structured querry language* - strukturovaný dotazovací jazyk). Vedle toho se lze často setkat s aplikacemi, napsanými v jazycích, které jsou součástí databází Gupta, Oracle, PowerBuilder, dBase nebo Paradox.

Mezi dnes nejrozšířenější jazyky patří *Pascal* a jazyk *C*. Možnosti, které tyto jazyky nabízejí, jsou v podstatě stejné (alespoň pokud jde o implementace těchto jazyků na osobních počítačích). Jde o jazyky univerzální, které lze rozumně použít pro řešení většiny problémů. Jediný závažnější rozdíl mezi nimi představuje céčkovská adresová aritmetika, která nemá v běžných implementacích *Pascalu* obdobu.

Pokud ale vycházíme z vyloženě objektově orientovaného návrhu, bude asi nejvhodnější jazyk *C++*. Tento jazyk je v podstatě nadmnožinou Céčka, navíc obsahuje především objektové typy s plně rozvinutými možnostmi (omezování přístupu ke složkám, vícenásobnou dědičnost, polymorfismus). Pro objektové typy lze v *C++* také definovat vlastní verze většiny operátorů. Kromě toho nabízí *C++* šablony (generické typy a funkce) a možnost vyvolávat a ošetřovat výjimečné situace (to se používá zejména při zpracování chyb). Bohužel zdaleka ne všechny překladače dosud tyto pokročilé možnosti jazyka *C++* implementují.

Poslední dobou se objevují i „čistě objektové“ programovací jazyky, jako *Actor*, *Eiffel* nebo *Smalltalkk*. V současné době jde většinou o interpretační systémy, které se příliš nehodí k implementaci výkonných aplikací; v budoucnosti se však mohou stát účinným vývojovým nástrojem.

Pokud nepředstavuje velikost programu a rychlost velký problém, lze použít některých vývojových prostředků založených na generátorech kódu. Např. při vývoji aplikací pro MS Windows v prostředí Borland C++ 4.0 můžeme využít *AppExpert*, aplikaci, která na základě jakéhosi dotazníku vytvoří zdrojový kód v C++, založený na objektově orientované knihovně Object Windows Library. Do tohoto „prototypu“ programu pak doplníme výkonné součásti.

V poslední době se při tvorbě aplikací pro Windows dosti rozšířilo využívání vizuálních vývojových prostředí, jako je *Visual Basic* nebo *Visual C++*. V těchto prostředích se obvykle pomocí myši sestaví z předdefinovaných prvků uživatelské rozhraní aplikace a případně i některé funkční součásti. (Součástmi mohou být i velmi rozsáhlé funkční celky - třeba celý tabulkový procesor, textový procesor, utilita pro kreslení diagramů apod. [16].)
Na základě „graficky“ sestaveného uživatelského rozhraní pak prostředí vytvoří program, ke kterému doplníme další potřebné části.

## 9.5 Další kroky

Následujícím krokem při vývoji softwarového produktu je podrobný návrh. Zde určujeme vlastně vnitřní strukturu modulů. Definujeme význam jednotlivých funkcí v modulech a navrhujeme algoritmy, které v nich použijeme. Přitom specifikujeme i datové struktury, které jsme neuvažovali při návrhu architektury.

Teprve po sestavení podrobného návrhu lze přistoupit k vlastnímu programování, tedy k přepisu algoritmů do zvoleného programovacího jazyka.

Pak přijde na řadu ladění. První z možných kroků, doporučovaný při týmové práci, je nechat přečíst svůj kód jinému programátorovi. Často se tak podaří odhalit včas různá nepříjemná opomenutí.

Dalším krokem, který následuje po formálním odladění, je testování. Nejprve se zpravidla testují jednotlivé funkce, potom jednotlivé moduly. Teprve nakonec se testuje systém jako celek. Přitom se volí různé přístupy. Např. seznam předběžných požadavků bude základem testů ověřujících, zda programový produkt vyhovuje požadavkům zákazníka.

Obvykle se také testují extrémní případy vstupních dat, případy, kdy má aplikace jen minimum paměti apod.

Při testování lze s aplikací zacházet jako s černou skříňkou nebo lze analyzovat její zdrojový kód a podle toho hledat chyby.

U aplikací, které budou komerčně šířeny, se zpravidla také provádí tzv. beta-testování, při kterém se produkt poskytne zdarma nebo za minimální poplatek vybraným uživatelům a ti shromáždí informace o jeho vadách, problémech a nedostatcích.

Poslední fází, o které se zmíníme, je údržba. Ta zahrnuje např. průběžné odstraňování nedostatků (chyby by se v konečném produktu vyskytnout neměly, ale zkušenosti s produkty velkých firem ukazují, že se tam se železnou pravidelností objevují - viz např. problémy kolem překladačů Borland C++).

Vedle toho půjde o vývoj nových verzí, které budou vyhovovat novým nárokům uživatelů, které bude možno provozovat v jiných operačních prostředích (může jít např. o přechod z Windows pod Windows NT) nebo které budou obsahovat účinnější algoritmy.

Je samozřejmé, že speciálně při údržbě programu oceníme průzračnost architektury, přehlednost návrhu na všech úrovních, dobře zpracovanou dokumentaci, zkrátka dobrý programovací styl.

# Kapitola 10

# Návrh architektury založený na analýze požadavků

Metody analýzy požadavků se zpravidla - přímo nebo nepřímo - opírají buď o rozbor toku dat v systému nebo o rozbor struktury dat. Tok dat se obvykle charakterizuje v souvislosti s funkcemi, které přetvářejí vstupní data na data výstupní.

Jako příklady si ukážeme metodu založenou na diagramech toku dat [31] a Jacksonovu metodu (*Jackson System Development* [14], [15]; u nás se pro ni občas používá označení „Jacksonovo strukturované programování“).

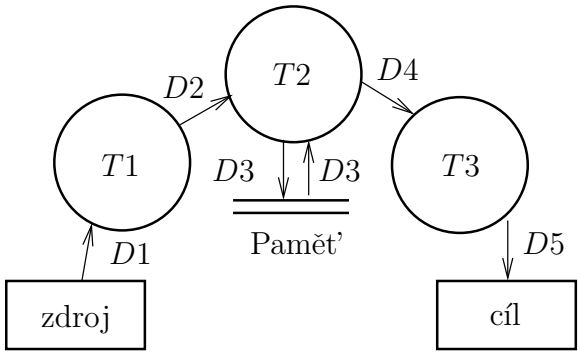
## 10.1 Diagramy toku dat

Při rozboru toku dat v systému se zabýváme transformacemi informací při průchodu systémem, řízeným počítačem. Do systému přicházejí informace v různých podobách a z různých zdrojů. Jejich transformace může zahrnovat stejně dobře náročné numerické výpočty jako pouhé porovnání. Výstupem pak může být tisk sestavy nebo třeba jen rozsvícení kontrolky.

Informace, které procházejí systémem, jsou podrobovány řadě transformací. Tyto transformace vyjádříme diagramem, ve kterém se obvykle používají následující značky:

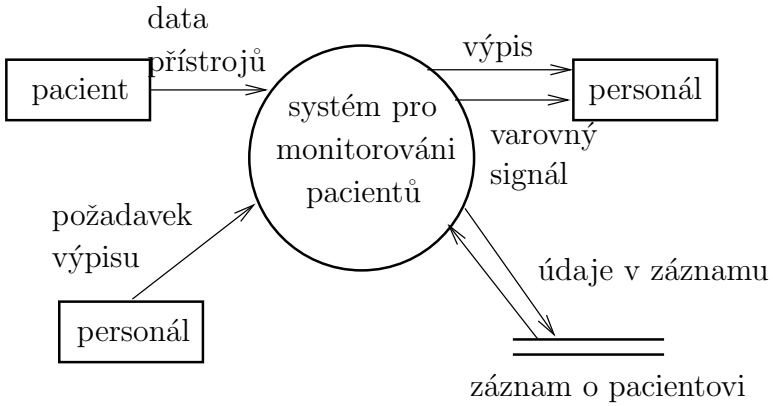
- Obdélník vyznačuje vnější objekt, který představuje buď zdroj, odkud přicházejí informace do systému, nebo příjemce informací od systému. Od jednoho zdroje může vycházet i více datových toků.
- Kruh („bublina“) označuje proces, tedy transformaci dat.
- Spojnice označují přenosy informací mezi procesy navzájem nebo mezi procesy a vnějšími objekty. Šipky ukazují směr přenosu dat. Každá spojnice by měla být pojmenována.
- Dvojice vodorovných rovnoběžek označuje zařízení na ukládání dat.

Příklad diagramu vidíte na obr. 10.1; v něm data ze zdroje ($D1$) procházejí transformačním procesem $T1$, při kterém se přemění na data $D2$. Ta převezme proces $T2$, uloží si je jako data $D3$ do zařízení pro úschovu dat. Odtud si je vyzvedne a jako data $D4$ je předá procesu $T3$, který je po patřičné transformaci předá jako data $D5$ konečnému příjemci.




Obr. 10.1: Příklad diagramu toku dat.

Pomocí takovýchto diagramů lze popisovat systém nebo program na libovolné úrovni abstrakce. Na základní úrovni se zpravidla celý softwarový systém vyjadřuje jediným kruhem - to znamená, že na této úrovni popisujeme pouze informační toky mezi systémem a jeho okolím. V dalších krocích pak upřesňujeme strukturu systému, tj. rozkládáme jednotlivé procesy na podprocesy a určujeme datové toky mezi nimi.




Obr. 10.2: Základní diagram toku dat v systému pro sledování pacientů

Tyto diagramy nevyjadřují přímo pořadí událostí (např. pořadí vstupů od jednotlivých zdrojů apod.). Nevyznačují také žádným způsobem řídicí struktury programu (cykly, podmínky apod.). Takovéto problémy se řeší později, až při vlastním softwarovém návrhu.

**Příklad 10.1: monitorovací systém**

Jako příklad použití diagramů toku dat k rozboru požadavků si ukážeme návrh programového vybavení pro sledování pacientů na jednotce intenzivní péče [12]. Na obrázku 10.2 vidíte, jak může vypadat základní diagram.
Při zjemňování tohoto diagramu musíme rozložit *systém pro monitorování pacientů* na podsystémy a ujasnit si tok dat mezi nimi. Zmíněný systém se např. bude skládat z *centrálního monitorovacího systému*, který bude přijímat informace od *systémů pro sledování jednotlivých pacientů* a bude využívat *soubor s informacemi o mezních hodnotách* sledovaných údajů. Získané údaje bude předávat *systému pro aktualizaci záznamů* o pacientech. *Systém pro generování výpisů* (sestav) může být na centrálním monitorovacím systému nezávislý, potřebuje mít přístup pouze k záznamům o pacientech.

Personál bude přijímat varovné signály od centrálního monitorovacího systému. S požadavky na výpis se bude obracet na generátor výpisů. Diagram, který vznikne na základě uvedených zjemnění, vidíte na obrázku 10.3.

Čtenář se může sám pokusit navrhnout další zjemnění, která se budou týkat nejspíše centrálního monitorovacího systému.
Analýza ovšem nebude úplná, pokud se budeme zabývat pouze tokem dat. Musíme si samozřejmě všímat i obsahu těchto toků, tedy samotných dat, která zde proudí. Z popisu těchto dat vycházíme při návrhu funkcí, které provádějí jejich transformaci.

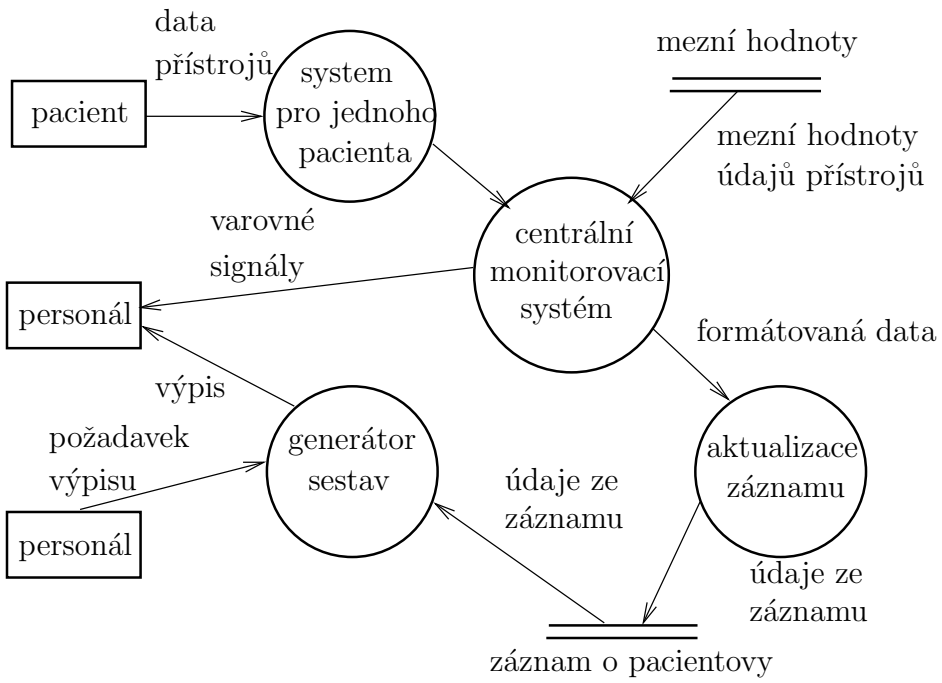
Jednu z možných metod představují *slovníky dat*. Jde o popis struktury dat, založený na vhodně definované formální gramatice, která často připomíná popis datových typů programovacího jazyka. Slovník musí obsahovat definice všech dat, na která v tokovém diagramu narážíme. Elementární data definujeme tak, že popíšeme jejich význam, složená data vyjádříme rozkladem na komponenty.

Při popisu dat stanovíme základní datové struktury (posloupnost údajů, opakující se data, selekci). Vyznačíme - je-li to možné - počet opakování a údaje, které se mohou, ale nemusí vyskytovat (volitelná data).

**Příklad 10.2: slovník dat**

Jako jednoduchý příklad se podíváme se na strukturu telefonního čísla. Můžeme je popsat takto:

telefonní číslo = [místní číslo | meziměstské číslo]
místní číslo = $\{\text{číslice}\}_3^8$
meziměstské číslo = předčíslí + kód města + místní číslo
předčíslí = [vnitrostátní předčíslí | mezinárodní předčíslí]
vnitrostátní předčíslí = 0
mezinárodní předčíslí = 00 + kód státu




Obr. 10.3: Zjemnění diagramu toku dat v systému pro sledování pacientů

Při tomto popisu jsme používali „+“ pro vyznačení sekvence (mezinárodní předčíslí se skládá ze dvou nul, za kterými následuje kód státu), „[ | ]“ pro vyznačení selekce (telefonní číslo je buď místní nebo meziměstské) a „$\{\}_m^n$“ označovalo opakování (místní číslo se skládá ze tří až osmi číslic).

Při návrhu rozsáhlých softwarových produktů může být slovník dat velice rozsáhlý a práce s ním může přesahovat možnosti manuálního zpracování. Proto některé systémy pro počítačovou podporu tvorby software (CASE) umožňují automatickou tvorbu a zpracování datových slovníků.

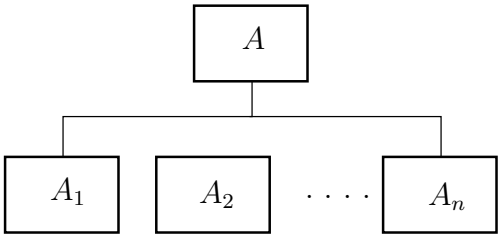
Jakmile skončíme popis dat, musíme popsat funkce, které budou obstarávat jejich transformace.

## 10.2 Jacksonova metoda

Tuto metodu vyvinul na základě analýzy informací a dat a jejich vztahů k návrhu programu M. A. Jackson koncem sedmdesátých let [14], [15]. Soustřeďuje se, podobně jako některé podobné systémy, na oblast informací o „reálném světě“. Programátor (vývojář) vytvoří nejprve model reality, kterou se bude systém zabývat.

Vývoj programu Jacksonovou metodou probíhá v následujících krocích:

- *Výběr objektů a akcí.* Určíme objekty, entity vnějšího světa, se kterými bude program pracovat, a akce, události, které se těchto objektů týkají. Vyjdeme od formulace problému, tedy od stručného popisu problému v běžném jazyku. Zhruba lze tvrdit, že objekty odpovídají podstatným jménům v tomto popisu, akce pak slovesům. (Samozřejmě takovéto tvrzení je třeba brát se značnou rezervou.)
- *Určení struktury objektů.* Pod pojmem *struktura* objektu v souvislosti s Jacksonovou metodou rozumíme dopad jednotlivých akcí na objekt. Akce, působící na objekt, mohou tvořit posloupnost, může nastat právě jedna akce z několika možných nebo se může určitá akce periodicky opakovat. Události a akce, které se týkají jednotlivých objektů, uspořádáme podle časového hlediska a popíšeme je pomocí Jacksonových diagramů.
- *Vytvoření počátečního modelu.* V tomto kroku začneme tvořit specifikaci systému jako modelu reálného světa. Komunikaci mezi procesy vyznačujeme pomocí *diagramů pro specifikaci systému*. V nich kruhem vyznačíme přenos pomocí vyrovnávací paměti FIFO (fronty) o neomezené kapacitě a kosočtvercem komunikaci, při které jeden proces může přímo použít stavový vektor druhého procesu. V těchto diagramech se objekty reálného světa obvykle označují příponou 0 a příponou 1 se označují procesy, které je modelují.




Obr. 10.5: Znázornění posloupnosti

V tomto kroku tedy sestavíme objekty a události do modelu procesu a určíme vztahy a spojení mezi modelem a reálným světem. Při popisu struktury modelu můžeme používat také popis pomocí *jazyka pro popis logických struktur* (v originále nazývaný *structure text*). Pokud jej použijeme, budeme jej označovat jako *textový popis*. V něm zkratkami *seq*, *itr* a *sel* označujeme postupně posloupnost (sekvenci), cyklus (iteraci) a větvení (selekci - viz dále). S použitím textového popisu se setkáme v následujícím příkladu.

Další kroky při návrhu Jacksonovou metodou jsou:

- *Specifikace funkcí.* Popíšeme funkce, které odpovídají akcím, zahrnutým do modelu. To znamená, že diagramy pro specifikaci systému rozšíříme o nově definované funkce (procesy), které propojíme s modelovým procesem pomocí datových proudů.

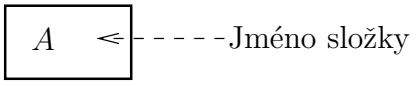
- *Popis časových vztahů.* V tomto kroku specifikujeme časová omezení, kladená na systém. V předchozích krocích jsme získali model, složený ze sekvenčních procesů, které pro vzájemnou komunikaci využívají jednak datových proudů a jednak přímého přístupu ke svým stavovým vektorům. Nyní je třeba určit časové vazby mezi nimi a případně synchronizační mechanismy pro komunikaci mezi jednotlivými procesy.

- *Implementace.* Vytvoříme návrh hardwarové a softwarové implementace systému. Přitom základem našeho postupu bude rozklad hierarchických struktur na menší části, které lze vyjádřit jako některé ze základních řídicích konstrukcí (posloupnost, iterace, selekce).

### 10.2.1 Jacksonovy diagramy

Jacksonovy diagramy znázorňují základní řídicí algoritmické struktury (posloupnosti, cykly a podmínky, jak jsme o nich hovořili v kapitole o algoritmech, viz 1.1.4.). Základní složka struktury se označuje obdélníkem, ve kterém je vepsán název této složky - viz obr. 10.4.

**Posloupnost**




Obr. 10.4: Základní složka Jacksonova diagramu

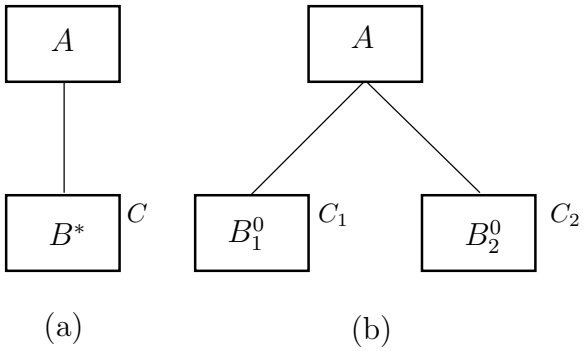
Jestliže se posloupnost operací $A$ skládá z operací $A_1, A_2, \ldots, A_n$, které provádíme postupně v uvedeném pořadí, vyznačíme to způsobem, který ukazuje obr. 10.5.

**Cyklus**

Jestliže je v cyklu $A$ opakování operace $B$ (těla cyklu) vázáno na podmínku $C$, znázorníme to způsobem, který vidíte na obr. 10.6(a). Podmínka opakování $C$ je samozřejmě nedílnou součástí cyklu $A$ jako celku; přesto ji zapisujeme k tělu cyklu.

**Selekce**

Selekce $A$ má obecně takovýto tvar: je-li splněna podmínka $C_1$, provede se operace $B_1$, jinak je-li splněna podmínka $C_2$, provede se akce $B_2, \ldots$, jinak je-li splněna podmínka $C_n$, provede se akce $B_n$. Takovouto selekci vyjádříme diagramem, který vidíte na obrázku 10.6(b). Také zde jsou podmínky $C_1, C_2, \ldots, C_n$ nedílnou součástí selekce $A$ jako celku, nikoli akcí $B_1, B_2, \ldots, B_n$. Přesto se zapisují ke složkám.
Podobné diagramy lze použít i k vyjádření struktury dat. Tyto diagramy se používají také při návrhu algoritmů (tj. při návrhu na nižší úrovni) Jacksonovou metodou na základě analýzy struktury dat.




Obr. 10.6: Znázornění iterace (a) a selekce (b);$C$ resp. $C_i$ jsou podmínky

**Příklad 10.3: místní doprava**

Tento příklad jsme v podstatě převzali z [12]. Jde o rozbor požadavků pro počítačem řízenou místní dopravu mezi dvěma objekty velké univerzity.

Vyjdeme od zadání, tedy od slovního popisu problému.

*Velká univerzita využívá dvou budov, které jsou od sebe vzdáleny více než 2 km. Protože studenti mají přednášky v obou budovách, chce univerzita vybudovat mezi nimi automaticky řízenou kyvadlovou dopravu.*

*Půjde o jeden vůz, jezdící po kolejích a řízený počítačem. Trat' bude mít dvě stanice, u každé budovy jednu. V každé ze stanic bude přivolávací tlačítko, jehož stisknutím si studenti mohou vyžádat transport do druhé stanice.*

*Pokud bude vůz již čekat ve stanici, studenti nastoupí a vůz odjede. Pokud je vůz na cestě, musí studenti počkat, až vůz dojede do opačné stanice, nastoupí případní cestující a vůz s nimi přijede. Pokud čeká vůz v opačné stanici, odjede a vezme studenty, kteří stiskli tlačítko.*

*Jinak bude vůz čekat ve stanici, dokud si jej někdo stisknutím tlačítka nevyžádá*[1].

**První krok: výběr objektů a akcí**. Přezkoumáme podstatná jména, která se v tomto popisu vyskytují, a tak určíme objekty návrhu. Zde by mohly připadat v úvahu tyto entity: *univerzita, budova, vůz, studenti, přednášky, transport, stanice.*

*Budova, univerzita, stanice, studenti, přednášky a transport* nesouvisí přímo s modelem a proto je ponecháme stranou. Nás se bezprostředně týká pouze *vůz a tlačítko.*

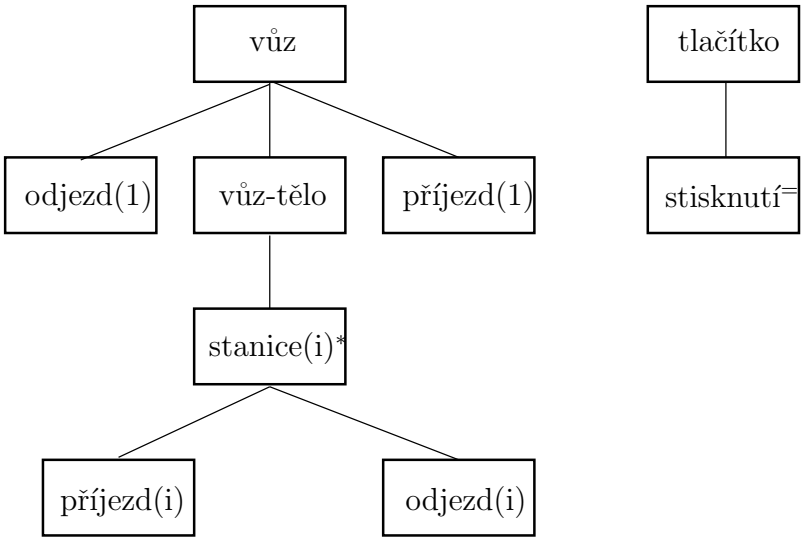
Dále si všimneme akcí, které se týkají zvolených objektů, tedy sloves. Půjde o slovesa *stisknout* (týká se tlačítka), *přijet* a *odjet* (týkají se *vozu*). Zamítneme *nastoupit*, neboť se týká především studentů, a *čekat*, neboť představuje spíše stav než akci. (Při další analýze se objeví jako příznak stavu, spolu s možností „transport“, tedy „vůz je na cestě“.) *Vyžádat si* znamená zde totéž co *stisknout* (tlačítko).

Poznamenejme, že později můžeme dospět k závěru, že potřebujeme určité objekty a akce přidat. Pokud bychom např. chtěli, aby náš program také sledoval, kolik studentů tuto dopravu využívá, museli bychom v naší analýze vzít v úvahu i objekt *student* a akci *nastoupit.*

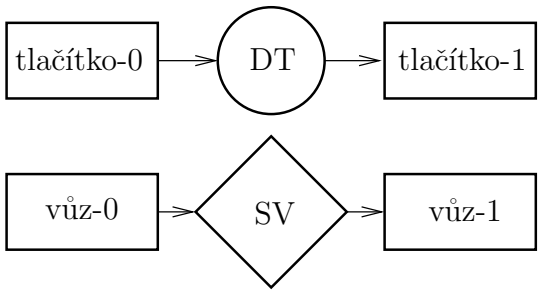
**Druhý krok: specifikace struktury objektů**. Vůz vyjede na počátku z první stanice, pak opakovaně přejíždí mezi první a druhou stanicí a skončí opět v první stanici. Přejíždění mezi stanicemi se skládá z příjezdu do $i$-té stanice a z odjezdu z $i$-té stanice. V diagramu to vyznačíme indexem $i$.

Jediná akce, která se týká tlačítka, je stisknutí. Tato akce se bude samozřejmě opakovat. Strukturální diagramy zvolených objektů vidíte na obr. 10.7.

[1] Nenechte se zmást skutečností, že by se u nás něco podobného nemohlo stát. Existují státy, kde mají univerzity dokonce prostředky na to, aby se snažily studentům usnadnit život, a stát to nepokládá za důvod k drastickým úsporným opatřením.




Obr. 10.7: Strukturální diagramy pro objekty vůz a tlačítko




Obr. 10.8: Strukturální diagramy pro vůz a tlačítko

Tento diagram představuje časově uspořádané akce, které se týkají zvolených objektů. Je-li třeba, můžeme jej doprovodit vysvětlivkami a poznámkami - *např. index i smí nabývat pouze hodnot 1 a 2.*

**Třetí krok: vytvoření počátečního modelu**. Nyní „propojíme model s reálným světem“. Proces *tlačítko-1* může číst data (údaje o stisknutí tlačítka) z vyrovnávací paměti.
Na druhé straně proces *vůz*-1 musí mít přístup k okamžitým hodnotám přepínačů, které řídí funkci skutečného vozu. To znamená, že bude přímo číst data ze stavového vektoru *vozu*-0. Při jeho zpracování narazíme na příznaky ČEKÁNÍ a TRANZIT, které vyjadřují okamžitý stav vozu. V obou případech musíme neustále kontrolovat, zda nedošlo ke změně stavu.

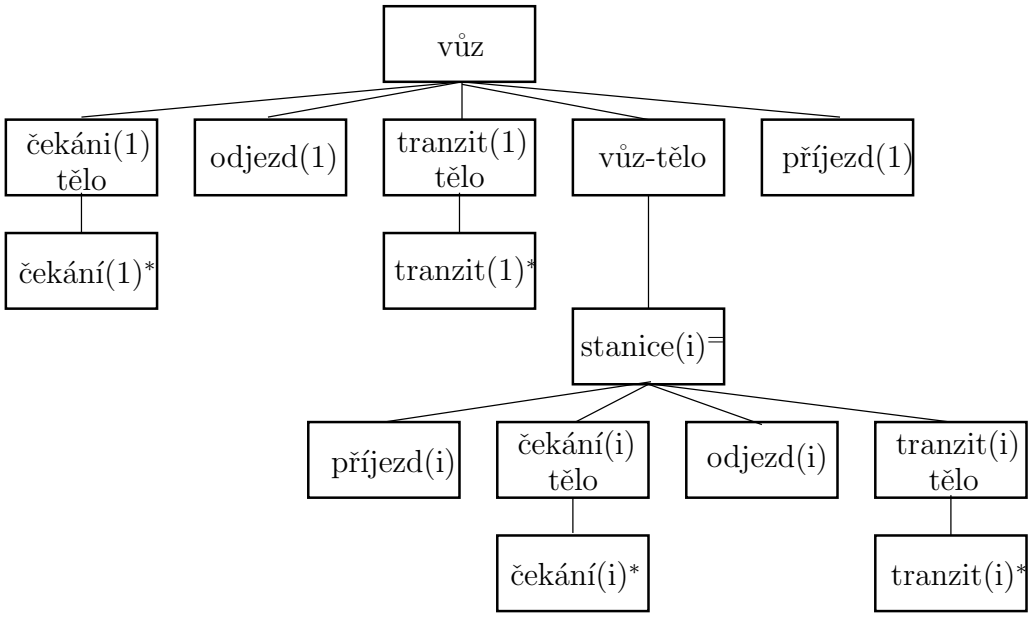
Systémové specifikace pro zkoumanou kyvadlovou dopravu, ke kterým jsme doposud dospěli, vidíte na obrázku 10.8.

Textový popis pro proces *tlačítko*-1 bude mít tvar

```
TLAČÍTKO-1
 čti DT;
 STISKNUTÍ-tělo: itr pokud DT
   STISKNUTÍ
   čti DT;
 STISKNUTÍ-tělo: konec
TLAČÍTKO-1: konec
```

Tento textový popis struktury *tlačítko*-1 plně odpovídá strukturálnímu diagramu; navíc specifikuje vztah k reálnému světu (čtení dat DT prostřednictvím vyrovnávací paměti) a upřesňuje podmínku iterace.

Podobně popíšeme strukturu *vůz*-1 (viz strukturální diagram na obr. 10.9).




Obr. 10.9: Strukturální diagram pro objekt vůz po dalším zpřesnění

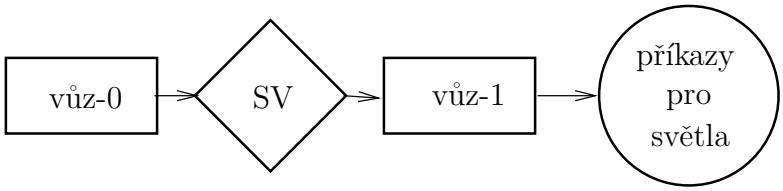
```
VŮZ-1 seq
  čti SV;          // čtení stavového vektoru
  ČEKÁNÍ-tělo: itr pokud ČEKEJ1
      čti SV;
  ČEKÁNÍ-tělo: konec
  ODJEZD(1);
  TRANZIT-tělo1: itr pokud TRANZIT1
      čti SV;
  TRANZIT-tělo1: konec
  VŮZ-tělo1 itr
    STANICE seq
      PŘÍJEZD(i);
      ČEKÁNÍ-tělo: itr pokud ČEKEJi
        čti SV;
      ČEKÁNÍ-tělo: konec
      ODJEZD(i);
      TRANZIT-tělo itr pokud TRANZITi
        čti SV;
      TRANZIT-tělo: konec
     STANICE: konec
  VŮZ-tělo: konec
  PŘÍJEZD(1);
VŮZ-1: konec
```

**Čtvrtý krok: specifikace funkcí**. Ve voze jsou signální světla, která se rozsvítí při příjezdu do i-té stanice. Předpokládejme, že k rozsvícení resp. zhasnutí slouží funkce ZAPSV(i) resp. VYPSV(i).

Příkazy k rozsvícení či zhasnutí těchto světel musí dát proces *vůz*-1. To znamená, že strukturální diagram tohoto procesu upravíme způsobem, který vidíte na obr. 10.10.
Vedle toho se musíme postarat o příkazy pro motory. Zavedeme proto nový funkční proces *motory*, který bude napojen na proces *vůz*-1. Příslušný datový tok označíme V1D.

Příkazy pro motory budou mít tvar např. START a STOP. Příkaz STOP je třeba vydat v okamžiku, kdy senzory oznámí příjezd do stanice, a start po prvním stisknutí tlačítka, jestliže vůz čeká ve stanici.

Samozřejmě je nezbytné zabezpečit, aby proces vůz-1 četl stavový vektor vozu a aby proces *motory* četl




Obr. 10.10: Upravený strukturální diagram pro objekt vůz

(s dostatečnou frekvencí opakování) informace o možném příjezdu do stanice, aby stihl vůz včas zastavit. Tím ovšem předbíháme - časové aspekty budeme rozebírat v následujícím kroku.

Podívejme se na zpřesněný popis vozu (změny označíme pro snazší orientaci čtenáře vykřičníkem):

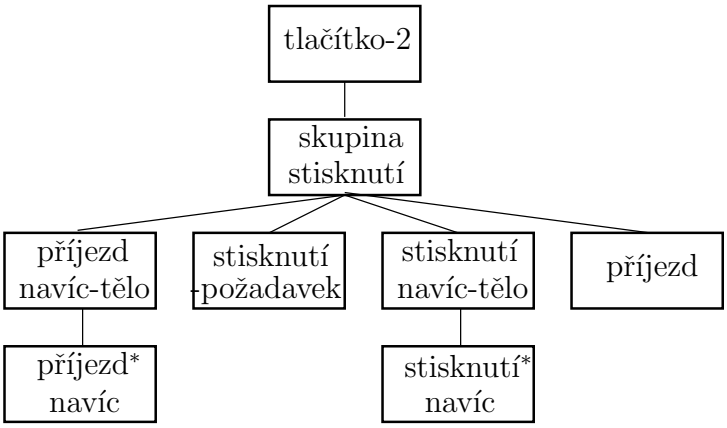
```
VŮZ-1 seq
  ZAPSV(1);       // !
  čti SV;         // čtení stavového vektoru
  ČEKÁNÍ-tělo: itr pokud ČEKEJ1
      čti SV;
  ČEKÁNÍ-tělo: konec
  VYPSV(1);       // !
  ODJEZD(1);
  TRANZIT-tělo1: itr pokud TRANZIT1
      čti SV;
  TRANZIT-tělo1: konec
  VŮZ-tělo1 itr
    STANICE seq
      PŘÍJEZD(i);
      zapiš příjezd do V1D  // ! data pro motory
      ZAPSV(i);             // !
      ČEKÁNÍ-tělo: itr pokud ČEKEJi
        čti SV;
      ČEKÁNÍ-tělo: konec
      VYPSV(i)              // !
      ODJEZD(i);
      TRANZIT-tělo itr pokud TRANZITi
        čti SV;
      TRANZIT-tělo: konec
     STANICE: konec
  VŮZ-tělo: konec
  PŘÍJEZD(1);
  zapiš příjezd do V1D
VŮZ-1: konec
```

Nakonec se ještě vrátíme k procesu *tlačítko*. Nyní je nezbytné rozlišovat mezi prvním stisknutím, které znamená opravdu požadavek na přivolání vozu, a mezi dalšími stisknutími, která jsou již bezvýznamná, neboť vůz je již na cestě. Popíšeme tedy tento proces znovu, podrobněji, a nový popis označíme *tlačítko*-2.

Proces *motory* informuje proces *tlačítko* o vyřízení požadavku, tj. o tom, že vůz přijel do stanice. Zde ale musíme opět rozlišovat mezi příjezdem vyžádaným stisknutím tlačítka a příjezdem nevyžádaným („navíc“, tj. přejezdem, vyžádaným v opačné stanici).

Zpřesněný popis tlačítka ukazuje diagram na obr. 10.11.
Textový popis tlačítka může mít tvar

```
TLAČÍTKO-2: seq
 požadavek := ne;  // zápis do stavového vektoru
 čti DT a DM;      // navíc čte data motoru
 TLAČÍTKO-tělo: itr
```




Obr. 10.11: Zpřesněný strukturální diagram pro objekt tlačítko

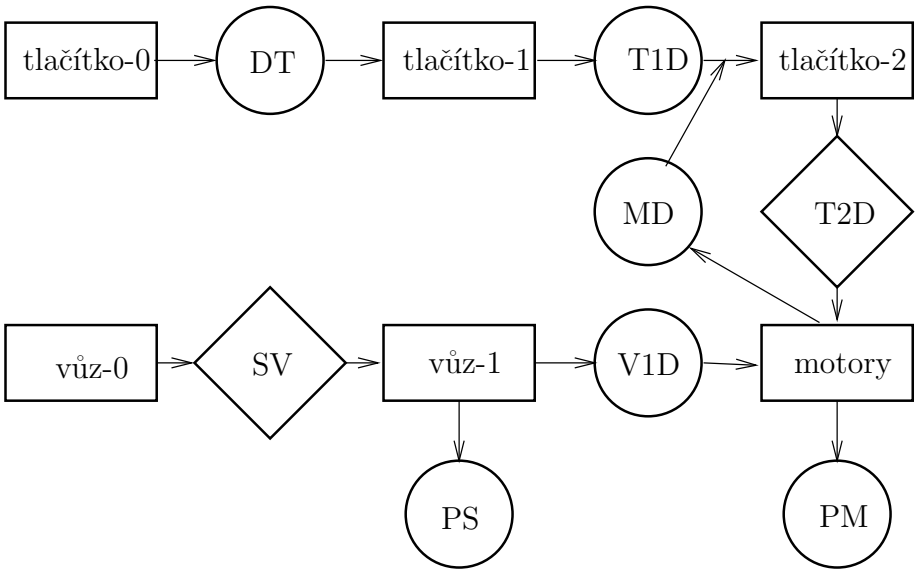
```
    SKUPINA-STISKNUTÍ: seq
      PŘÍJEZD-NAVÍC-tělo: itr pokud (PŘÍJEZD)
        čti DM a DT;
      PŘÍJEZD-NAVÍC-tělo: konec
      STISKNUTÍ-POŽADAVEK: seq
        požadavek := ano;
        čti DT a DM;
      STISKNUTÍ-POŽADAVEK: konec
      STISKNUTÍ-NAVÍC: itr pokud (PŘÍJEZD)
        čti DT a DM;
      STISKNUTÍ-NAVÍC: konec
      PŘÍJEZD seq
        požadavek := ne;
        čti DT a DM;
      PŘÍJEZD: konec
    SKUPINA-STISKNUTÍ: konec
  TLAČÍTKO-tělo: konec
TLAČÍTKO-2: konec
```

Vstup procesu *tlačítko*-2 se skládá ze dvou datových proudů - od skutečného tlačítka a od motorů. Zde postačí „hrubé sloučení“ těchto proudů (proces čte ta data, která má právě k disposici); existují ovšem i jiné způsoby zpracování více vstupních proudů - viz [14].

Vzájemné propojení těchto procesů vidíte na obr. 10.12.
**Pátý krok: určení časových vztahů**. Zde musíme určit například okamžik, ve kterém je třeba vydat příkaz STOP, a to v závislosti na rychlosti vozu a na kvalitě brzd. Dále musíme stanovit doby odezvy při přepnutí světel apod. Jejich hodnoty budou samozřejmě záviset na technických parametrech použitých zařízení.

Je zřejmé, že se program bude skládat z několika paralelně běžících procesů. Z provedené analýzy ale plyne, že není třeba zavádět žádné speciální synchronizační mechanismy.

**Šestým krokem, implementací**, se zde již zabývat nebudeme.




Obr. 10.12: Strukturální diagram pro vůz a tlačítko

# Kapitola 11

# Objektově orientovaný návrh

Objektově orientovaný návrh, podobně jako např. návrh založený na Jacksonově metodě, vytváří programovou reprezentaci reálného světa. Výsledkem je ovšem systém složený z *objektů*, programových struktur, které jsou - nebo spíše mohou být - odrazem objektů reálného světa a které modularizují zároveň informace i jejich zpracování (zatímco „klasické“ metody modularizovaly pouze zpracování dat).

Objekty navzájem komunikují prostřednictvím *zpráv*, které si posílají. Dále si vysvětlíme, co to vlastně znamená.

## 11.1 Základní pojmy objektově orientovaného programování

Objektově orientované programování (OOP) vychází z představy *objektu* jako základní programové struktury. Objekt v programu představuje obvykle model nějaké složky reálného světa. Z hlediska toku informací v programu představuje objekt zpravidla buď zdroj informací nebo jejich příjemce. Může ovšem také představovat informaci samu o sobě.

Objekt se skládá zpravidla z datové struktury, která bývá *soukromá* (nepřístupná jiným složkám programu) a z operací, které lze s těmito daty provádět. Složky datové struktury, která tvoří objekt, obvykle označujeme jako *atributy*[1]; procedury, funkce a operátory, které implementují operace s daty, označujeme jako *metody*. Metody smějí pracovat s daty objektu. Část z metod může být také soukromá.

Každý objekt má své *rozhraní*, přes které přijímá *zprávy*.

Každá zpráva vlastně představuje žádost, aby objekt provedl některou z možných operací. Pokud objekt zprávu přijme, zavolá některou z metod. Zdůrazněme, že posláním zprávy - tedy zpravidla voláním metody - říkáme, kterou operaci má objekt provést, neříkáme však, *jak* ji má provést. To je vnitřní záležitost objektu.

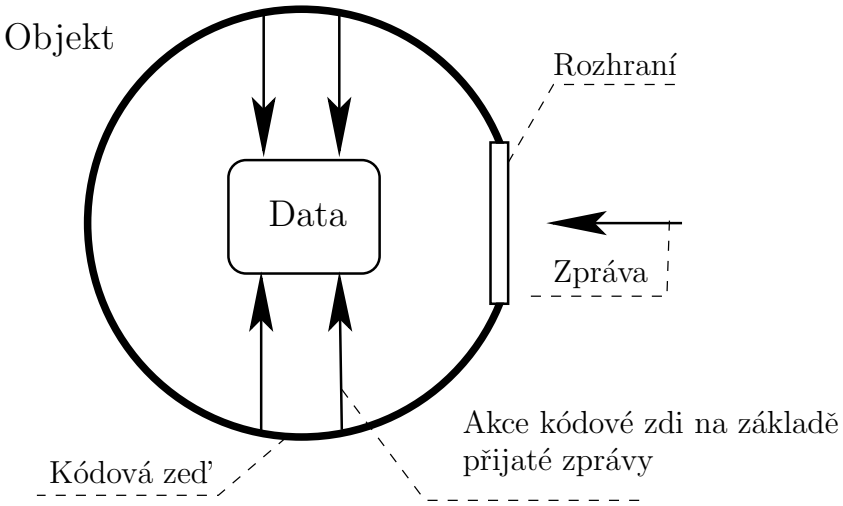
Tím, že část objektu (obvykle všechna data a část metod) označíme za soukromé, dosáhneme *ukrytí informace*, nebo přesněji *ukrytí implementace*. Podrobnosti implementace zůstanou skryty před ostatními částmi programu.

Výsledkem je, že k datům mají přístup pouze určité části kódu. Jinými slovy, data objektu jsou programovým kódem metod chráněna před neoprávněným použitím. Kód data identifikuje a zabezpečuje oprávněné operace s nimi.

Tento aspekt OOP bývá občas vyjadřován metaforou o *kódové zdi okolo každého kousku dat* a znázorňován obrázky podobnými jako obr. 11.1.

Objektový návrh vede k rozkladu programu na přirozené moduly, které lépe než jiné konstrukce odpovídají objektům reálného světa a shrnují jak data tak i akce, které se k nim váží.

[1] Rád bych čtenáře upozornil, že v oblasti OOP vládne v odborné literatuře neuvěřitelný terminologický zmatek. Názvy podobných konstrukcí se mohou lišit nejen u různých programovacích jazyků, ale i u různých autorů. Typickým příkladem může být právě *atribut*: zde tento termín používáme ve smyslu ”datová složka objektu”. V literatuře o programovacím jazyku Simula 67 se takto označují datové složky spolu se složkami funkčními (tedy metodami) a např. v dokumentaci k programovacímu jazyku *Actor* se tak označují data, popisující okna programu (tedy něco, co téměř vůbec nesouvisí s OOP).




Obr. 11.1: Objekt jako kódová zeď kolem každého kusu dat

### 11.1.1 Třída

Objekty v programu jsou představují modely objektů reálného světa. Objekty v programu jsou *instancemi*[2] datových typů, které se zpravidla označují jako *třídy* (*objektové typy*). Třída jako datový typ tedy představuje abstrakci společných vlastností jisté třídy objektů reálného světa (proto se pro ně občas používá poněkud matoucí označení *abstraktní datové typy*).
Ne každý datový typ, který vznikne abstrakcí společných vlastností skupiny objektů vnějšího světa, lze ovšem považovat za objektový typ. Jako objektové typy, tedy třídy, budeme označovat datové typy, které umožňují využívat následující tři vlastnosti: *zapouzdření, dědičnost a polymorfismus.*

Rozhlédneme-li se po odborné literatuře, snadno zjistíme, že tyto termíny - stejně jako řada dalších - je používána různými autory v lehce odlišných významech. Proto si zde uvedeme pouze nejobvyklejší interpretace.

#### Zapouzdření

Termínem *zapouzdření* (*encapsulation*) označujeme skutečnost, že v objektovém typu definujeme datové složky spolu metodami - tedy operacemi, které lze s datovými složkami třídy provádět.

Obvykle ovšem se pod zapouzdřením rozumí také skutečnost, že některé složky třídy mohou být soukromé, tj. že k nim - kromě metod třídy - nemají přístup žádné jiné součásti programu. Poznamenejme ale, že možnost omezit přístup k některým složkám instancí nebo tříd poskytují jen některé programovací jazyky (částečně např. Turbo Pascal od verze 6.0, v plné míře např. C++; naprosto chybí např. v Actoru, i když to je „čistě objektový“ jazyk).

Omezení přístupu k některým složkám objektů umožňuje programátorovi přesně vymezit, kdo může měnit datové složky, a tím zabránit některým chybám z nepozornosti. Vzhledem k tomu, že přístup ke složkám může ve značné míře kontrolovat překladač, můžeme tak zjistit řadu chyb již v době kompilace.[3]
Poznamenejme, že právě zapouzdření je základem metafory o kódové zdi, která chrání data před neoprávněným použitím, zabezpečuje oprávněné použití a v případě potřeby data také identifikuje (část této „kódové zdi“, tedy některé metody, mohou např. poskytovat informace o typu instance).

#### Protokol

Soubor zpráv, které může třída jako celek nebo instance určité třídy přijmout, spolu s popisem odezev na tyto zprávy, označujeme jako *protokol* dané třídy.

[2] V angličtině znamená slovo *instance* (vedle významů, známých v češtině) také *příklad* nebo *případ*. V souvislosti s programovacími jazyky a programováním se používá ve významu ”realizace abstraktního vzoru”. Označuje např. proměnnou (konstantu, formální parametr...) objektového typu, typy nebo funkce, vytvořené podle šablony (v C++) apod.

[3] Další nezanedbatelnou výhodou zapouzdření je, že řada jmen (identifikátorů) bude ukryta uvnitř třídy. Při týmové práci se tím snáze vyhneme kolizím, kdy dva programátoři použijí téhož jména pro dvě různé konstrukce v témže programu.

Chceme-li nějakou třídu používat, stačí znát její protokol; implementace metod, uložení dat apod. mohou uživateli zůstat skryty.

Často se v této souvislosti hovoří o *kontraktu*: Třída si na jedné straně klade určité požadavky a na druhé straně, jsou-li tyto požadavky splněny, zavazuje se poskytovat určité služby. (Podrobnější povídání na toto téma najdeme např. v Meyerově knize [35].)

### Dědičnost

*Dědičnost* (*inheritance*) je vlastnost, která umožňuje snadno od jednoho objektového typu - předka - odvodit typ další (potomka).[4]

Potomek zdědí všechny vlastnosti předka. To znamená, že bude mít stejné atributy a stejné metody jako třída, od které je odvozen. Obvykle však u potomka definujeme některé další datové složky nebo metody - specifikujeme nějaké další vlastnosti. Můžeme také v potomkovi překrýt některou z metod předka novou verzí.

Rozhraní potomka proto automaticky obsahuje rozhraní předka, může však být širší, neboť v potomkovi můžeme definovat nové metody. Implementace potomka v sobě obsahuje implementaci předka. To znamená, že potomek má všechny vlastnosti předka a nějaké další navíc. Z toho ale plyne, že odvozená třída představuje *podtřídu*[5] třídy rodičovské.

### Příklad 11.1

Vezměme třídu *lod'*, která bude reprezentovat obecné plavidlo. Atributy třídy *lod'* mohou být např. *okamžitá_rychlost, směr, poloha, výtlak, délka, rok_spuštění_na_vodu*. Metody třídy lod' mohou být např. *změň_směr, změň_rychlost, zakotvi nebo potop_se*.

Potomkem třídy *lod'* může být např. třída *plachetnice*, která bude mít navíc atribut *počet_stěžňů, celkový_počet_plachet a počet_vytažených_plachet*. Novou metodou může být např. *vytáhni_plachtu, sviň_plachtu* atd.

Jiným potomkem třídy *lod'* může být např. třída *parník*, která bude mít navíc atributy *počet_lodních_šroubů* a *zásoba_uhlí* a *metody doplň_uhlí* a *houkej*.

Je zřejmé, že každá *plachetnice* je *lod'* - jinými slovy třída plachetnic je podtřídou třídy lodí.

Třídy *plachetnice* a *parník* jsou mají stejného předka - označujeme je jako *sourozence*. U těchto tříd se budou nejspíš lišit metody *plav*, neboť parník nepoužívá plachty a plachetnice nemá parní stroj.

Jestliže je ale potomek podtřídou - tedy vlastně zvláštním případem - předka, znamená to, že instanci odvozené třídy můžeme kdykoli použít jako instanci rodičovské třídy. Odtud plyne, že pro objektové typy musí platit takováto pravidla:

- Proměnné typu ukazatel na rodičovskou třídu lze přiřadit hodnotu, která představuje ukazatel na instanci odvozeného typu.
- Instanci rodičovského typu lze přiřadit hodnotu odvozeného typu.
- Ukazatel na instanci odvozeného typu lze použít na místě formálního parametru typu ukazatel na rodičovskou třídu a podobně hodnotu odvozeného typu lze použít jako skutečný parametr při volání funkce nebo procedury, jejíž formální parametr je rodičovského typu.

K těmto pravidlům je ale třeba dodat, že v mnoha programovacích jazycích se při přiřazení hodnoty typu potomek instanci typu předek se přenesou pouze data, která lze v předkovi uložit, a tak se vlastně hodnota typu potomek transformuje na hodnotu typu předek. Podobné je to i při předávání parametrů hodnotou. Na druhé straně při přiřazování ukazatelů nebo při předávání parametrů odkazem k podobné změně nedojde.

[4] Jestliže od typu (třídy) **A** odvodíme typ **B**, označujeme typ **A** jako předka, bázovou třídu nebo rodičovskou třídu, typ **B** pak jako potomka, odvozenou třídu, dceřinnou třídu nebo podtřídu.

[5] Pozor na terminologické zmatky: Občas se setkáme s názorem, že *podtřída* je *předek*, nikoli potomek. Toto pojetí vychází ze skutečnosti, že instance potomka obsahuje vždy podobjekt, který je instancí předka. Proto se budeme raději tremínům *podtřída* a *nadtřída* vyhýbat.

Odvozený typ můžeme použít jako rodičovský typ pro další objektové typy; tak vznikne *dědická hierarchie tříd*. Pravidlo o tom, že potomek může vždy zastoupit předka, platí i pro všechny třídy v dědické hierarchii. To znamená, že předka může zastoupit i nepřímý potomek, tedy potomek, vzdálený v dědické hierarchii o několik úrovní.

Vedle *jednoduché dědičnosti*, kdy odvozený typ může mít pouze jednoho předka, se můžeme setkat i s *dědičností vícenásobnou*, kdy odvozený typ má dva nebo více předků. Vícenásobná dědičnost není běžnou součástí objektově orientovaných programovacích jazyků; najdeme ji např. v C++.

Vícenásobná dědičnost umožňuje snadno popsat objekty, které vznikly složením několika (v podstatě rovnocenných) složek. Ve většině případů však není nezbytná.

### Polymorfismus

*Polymorfismus* znamená v překladu mnohotvarost. V OOP tím vyjadřujeme skutečnost, že stejnou zprávu můžeme poslat instancím několika různých tříd, zpravidla ovšem tříd ze stejné dědické hierarchie. Přitom typ příjemce nemusíme v okamžiku odeslání zprávy znát (a nemusí jej znát překladač v době překladu programu). Příjemce může samozřejmě na přijatou zprávu reagovat různým způsobem v závislosti na svém skutečném typu.

Polymorfismus se uplatňuje především v souvislosti s pravidlem, které říká, že potomek může kdykoli zastoupit předka (viz předchozí odstavec). Z něj totiž plyne, že při operacích s instancí nemusíme znát její přesný typ - stačí vědět, že patří do určité dědické hierarchie a tedy že může přijmout danou zprávu.

### Příklad 11.2

V předchozím příkladu jsme zavedli třídu *lod'* a další odvozené třídy. V programu používáme proceduru *AkceSLodí*, jejímž formálním parametrem předávaným odkazem je objekt jménem *NějakáLod'* typu *lod'*. To znamená, že skutečným parametrem může být jak instance typu *plachetnice* tak instance typu *lod'* nebo *parník*.

V této proceduře pošleme objektu *NějakáLod'* zprávu *plav*. Pokud je typ *lod'* polymorfní, nemusíme se o typ skutečného parametru starat, použije se metoda odpovídající skutečnému typu instance.

Jestliže tedy byla skutečným parametrem instance typu *parník*, budou se lodi otáčet kolesa, zatímco pokud by byla skutečným parametrem instance typu *plachetnice*, budou se na lodi třepetat plachty.

### Časná a pozdní vazba

Polymorfismus předpokládá tzv. *pozdní vazbu*. To znamená, že skutečný typ instance, která je příjemcem zprávy, se vyhodnocuje až při běhu programu. Program pak ale musí při zpracování tohoto volání zpravidla prohledávat tabulky metod. To znamená prodloužení kódu a zpomalení běhu programu.

Proto se ve většině objektově orientovaných jazyků zpravidla implicitně používá *časná vazba*, při které se typ příjemce, a tedy také volaná metoda, vyhodnotí již při kompilaci. Pozdní vazba se používá jen pro vybrané metody, které označujeme (a deklarujeme) jako *virtuální*.[6]
V předchozím příkladu bychom tedy museli metodu *plav* deklarovat jak ve třídě *lod'* tak i ve třídách odvozených jako virtuální.

### Poznámka: implementace virtuálních metod v Turbo Pascalu

V této poznámce si povíme, jak se implementují virtuální metody v Turbo Pascalu. V řadě implementací jiných programovacích jazyků je postup podobný, nepředstavuje však jedinou možnost.

Pro každý objektový typ, který má (nebo zdědí) alespoň jednu virtuální metodu, zřídí překladač *tabulku virtuálních metod* (označujeme ji také zkratkou *VMT*, podle anglického *virtual method table*). Tato tabulka bude obsahovat adresy všech virtuálních metod třídy, ke které patří. programátorovi není přímo dostupná.

[6] Slovo *virtuální* znamená nejen zdánlivý, ale také *takový*, který má schopnost něco konat. (Slovník spisovného jazyka českého, Academia 1989.) Odtud zřejmě pochází označení virtuálních metod. (Někteří autoři, např. B. Stroustrup, mu připisují význam "řízený pomocí skrytých ukazatelů".)

Vedle toho do každé instance této třídy uloží překladač ukazatel na *VMT*. Adresu *VMT* do tohoto ukazatele uloží konstruktor. (Také tento ukazatel není programátorovi přímo dostupný.) Důležité je, že ukazatel na *VMT* je uložen v instancích všech typů v dané dědické hierarchii na stejném místě (v Turbo Pascalu je *VMT* uložena za atributy, deklarovanými v prvním členu dané objektové hierarchie, který obsahuje alespoň jednu virtuální metodu; jiné překladače mohou *VMT* ukládat např. jako úplně první datovou složku instance).

Při volání virtuální metody se nejprve z dané instance vezme ukazatel na tabulku virtuálních metod. V tabulce virtuálních metod se pak vyhledá adresa volané metody a ta se konečně zavolá.

Podívejme se na příklad. Vezmeme objektové typy *A* a *B*, deklarované takto:

```
type A = object
  i: integer;
  procedure p; virtual;
  procedure q; virtual;
end;
type B = object(A)
  j: integer;
  procedure p; virtual;
  procedure q; virtual;
end;
var bb: B;
    ua: ^A;
{ ... }
ua := @bb;
ua^.p;                              {zde voláme virtuální metodu}
```

Proměnné *ua* typu ukazatel na *A* můžeme přiřadit adresu instance *bb* typu *B*, který je potomkem typu *A*. Při volání virtuální procedury *p* se nejprve v instanci *ua*^ získá ukazatel na *VMT* (tím se vlastně určí skutečný typ této instance, na kterou *ua* ukazuje). V nalezené tabulce virtuálních metod se pak vyhledá adresa procedury *p* a ta se zavolá. Viz též obr. 11.1.

### 11.1.2 Složky instancí a složky tříd[7]

#### Atributy instancí

Již jsme si řekli, že datové složky objektů nazýváme atributy.

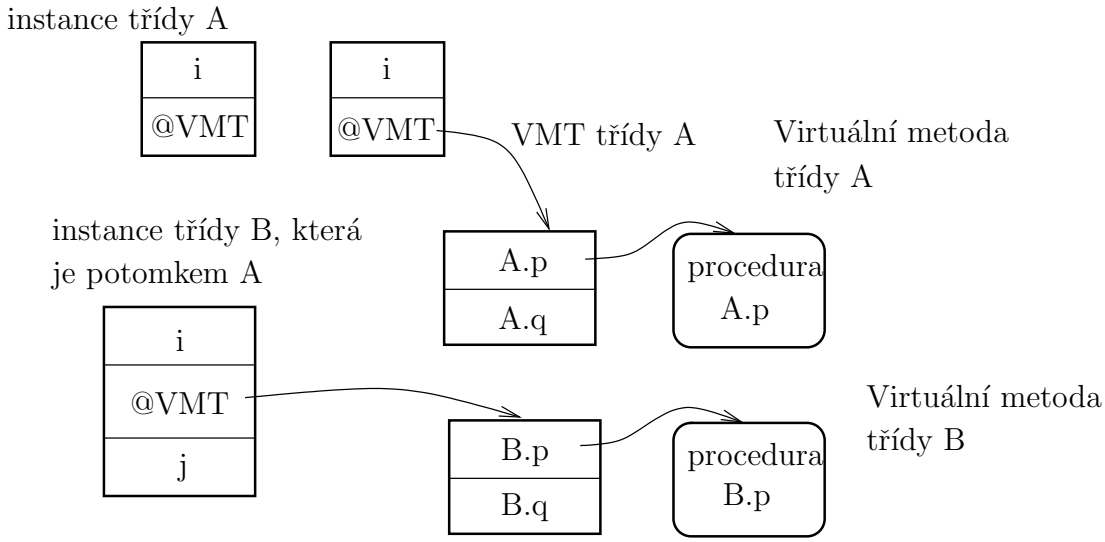
Zatím jsme ovšem hovořili pouze o atributech, které jsou individuálně vytvářeny pro každou jednotlivou instanci; označujeme je proto jako *atributy instancí*. Atributy instancí mohou mít v každé z existujících instancí jinou hodnotu, takže se hodí k vyjadřování individuálních vlastností různých instancí téže třídy.
Vedle toho může mít třída jako celek své vlastní atributy, které budeme označovat jako *atributy třídy*. Jde o datové struktury, které existují pouze jednou pro celou třídu a jsou společné pro všechny instance. (Atribut třídy je tedy vlastně globální proměnná, ukrytá uvnitř třídy.)

Atributy třídy obvykle vyjadřují skutečnosti, společné pro všechny instance, a proto nejsou na žádnou konkrétní instanci vázány. V programu mohou existovat i v případě, že jsme od dané třídy dosud nevytvořili ani jednu instanci.

#### Příklad 11.3

Zůstaneme stále u typu *parník*, který jsme zavedli v příkladu 11.1 v této kapitole, a předpokládejme, že *Vltava* a *Labe* jsou dvě instance této třídy. Atribut *počet_lodních_šroubů* lodi *Vltava* může mít hodnotu 2, zatímco týž atribut instance *Labe* může mít hodnotu 1. Tento atribut existuje pro každou instanci zvlášť a vyjadřuje individuální vlastnosti jednotlivých lodí.

[7] Turbo Pascal nabízí programátorovi pouze atributy a metody instancí. Atributy a metody třídy najdeme např. v C++ nebo v "čistě objektových" jazycích, jako je Actor nebo Smalltalk (metody třídy jsou také součástí Object Pascalu v borlandském produktu *Delphi*). Poznamenejme, že C++ se atributy a metody tříd označují jako "statické".




Obr. 11.2: Implementace pozdní vazby v Turbo Pascalu

Vedle toho můžeme definovat atribut třídy *parník*, který se bude jmenovat *počet_parníků* a bude evidovat aktuální počet parníků. Tato proměnná bude na počátku - před vytvořením první instance - mít hodnotu 0. Po zkonstruování každé z instancí se hodnota tohoto atributu zvýší o 1, po zrušení instance (dáme parník do šrotu) se jeho hodnota zmenší o 1.

**Metody instancí a metody tříd**

Dosud jsme hovořili pouze o metodách, které představují odezvu na zprávu poslanou instanci, a tedy pracují s jednotlivými instancemi.

Takové metody označujeme jako *metody instancí*. Metody instancí voláme vždy pro určitou konkrétní instanci a zavolaná metoda pak pracuje s atributy dané instance. Metody instancí mohou samozřejmě vedle atributů instancí používat i atributů třídy.

V určitých situacích je ale třeba poslat zprávu třídě jako celku, neboť třída plní mj. úlohu správce svých instancí. Odezvy na takovéto zprávy budou pak implementovat *metody třídy*, metody, které jsou sdruženy s třídou jako celkem a které nepracují s žádnou konkrétní instancí. Nemohou proto používat atributů instancí; smějí ale pracovat s atributy třídy. *Metody třídy* lze volat i tehdy, když žádná instance dané třídy neexistuje.

Jako typický příklad může posloužit metoda, která za běhu programu zkonstruuje novou instanci dané třídy. Instance, kterou chceme vytvořit, ještě neexistuje (nemusí existovat vůbec žádná instance dané třídy), takže jí žádnou zprávu poslat nemůžeme[8]. Adresujeme tedy zprávu třídě jako celku; tato zpráva bude vyjadřovat žádost, aby třída vytvořila novou instanci.

### 11.1.3 Poznámka k používání dědičnosti

Skutečnost, že předek je součástí potomka, tedy že instance odvozené třídy vždy obsahuje podobjekt, který je instancí předka, může svádět k nevhodnému užití dědičnosti.

Podívejme se na třídu plachetnice, definovanou v příkladu 11.2

Součástí plachetnice jsou i plachty - v programu pro ně definujeme zvláštní objektový typ, který výstižně pojmenujeme *plachta*. Co kdybychom definovali *plachetnici* jako potomka typu *plachta*?

[8]Nenechte se zmást skutečností, že např. v Turbo Pascalu instanci nejprve deklarujeme a pak zavoláme konstruktor, který se tváří jako metoda instancí. Deklarace pouze vyhradí volné místo, o kterém nelze ještě dost dobře hovořit jako o objektu. Teprve konstruktor udělá z vyhrazeného místa objekt.

Takto definovaná třída *plachetnice* by jistě mohla fungovat. Měla by ale řadu nežádoucích vlastností, které by byly přímým důsledkem logické chyby v návrhu: plachetnice *není* plachta.

Pro instance třídy *plachta* jistě má smysl volat metody *napni_se, třepetej_se* apod. Kdybychom definovali *plachetnici* jako potomka *plachty*, mohli bychom volat tyto metody i pro lodě, což zjevně nedává smysl.

Další problém, na který bychom narazili: co když bude plachetnice mít více plachet? Běžné programovací jazyky nedovolují, aby třída měla několik stejných předků. To znamená, že bychom jednu plachtu zdědili a ostatní museli definovat jako atributy; jedna plachta na lodi by tedy měla privilegované postavení oproti ostatním[9], což obvykle neodpovídá skutečnosti.

Při návrhu třídy *plachetnice* nemá smysl přenášet na ni rozhraní třídy *plachta*. Třída *plachetnice* bude využívat vlastností plachty, ale bude to *lod'*, nikoli plachta. Proto definujeme plachtu jako atribut; budeme-li chtít napnout plachty na *plachetnici*, pošleme jí zprávu *napni_plachty* a plachetnice na základě toho pošle všem svým plachtám zprávu *napni_se*. (Možná, že přitom vezme v úvahu informace o síle větru a napne jen některé - to závisí na implementaci třídy *plachetnice*.)

Z předchozího výkladu plyne, že:

- Plachetnice *je* lod' - má tedy smysl definovat třídu *plachetnice* jako *potomka* třídy *lod'*.
- Plachetnice *není* plachta, plachetnice *má* plachtu. Plachtu má tedy smysl definovat jako *atribut* třídy *plachetnice*.
- Vztah potomka k předkovi se někdy označuje anglickým termínem *isa* (rozloženo *is a*, tj. *je čímsi)*. Potomek *je* zvláštním případem předka.
- Vztah třídy k atributu se potom označuje termínem *hasa* (*has a*, tj. *má cosi*). Třída *má* atribut, ale *není* jeho zvláštním případem - využívá pouze jeho služeb. Atribut poskytuje (některé) své služby dané třídě.

**Poznámka v poznámce: soukromí předkové**

V některých programovacích jazycích (mám na mysli opět zejména své oblíbené C++) můžeme při deklaraci odvozené třídy určit, zda bude předek soukromý nebo veřejně přístupný (veřejný).

Veřejně přístupné složky veřejného předka budou veřejné i v potomkovi; to znamená, že potomek zdědí jak rozhraní tak i implementaci (a jde tedy o dědičnost, jak jsme ji popsali v odstavci 11.1.1.).

Specifikujeme-li předka jako soukromého, budou všechny zděděné složky, jak atributy tak i metody, v potomkovi soukromé. V tomto případě potomek získává implementaci, nikoli však rozhraní. Pokud chceme, aby některé složky soukromého předka byly veřejně přístupné, musíme je v potomkovi explicitně zveřejnit. Postavení soukromě zděděného předka je proto spíše podobné postavení atributu (hasa). Potomek má předka, ale nechlubí se s ním.

## 11.2 Objektově orientovaný návrh

Objektově orientovaný návrh vychází z objektové analýzy zadání. Jeden z možných postupů při objektově orientovaném návrhu se skládá z následujících kroků:

1. Definujeme problém.
2. Sestavíme seznam požadavků.
3. Na základě požadavků navrhneme neformálně architekturu (vyvineme neformální strategii) softwarového modelu problému z „reálného světa“.

[9] To není jen metafora. V některých programovacích jazycích se může lišit zacházení se zděděnou složkou od zacházení s atributem např. v konstruktoru při inicializaci nebo v destruktoru při likvidaci instance. Např. v C++ konstruktor nejprve inicializuje zdřdřné podobjekty (volá se jejich konstruktory), pak inicializuje odkazy na VMT v dané instanci a teprve pak volá konstruktory atributů dané instance. To znamená, že "zděděná" plachta by byla inicializována v jiném prostředí než plachty, deklarované jako atributy. Stěží si lze představit rozumnou situaci, kde bychom něco takového potřebovali.

4. Architekturu postupně formalizujeme a zpřesňujeme v následujících krocích:
   (a) Určíme objekty a jejich atributy.
   (b) Určíme operace, které mohou objekty provádět nebo které lze na ně aplikovat.
   (c) Určíme rozhraní objektů tak, že vyšetříme vztahy mezi objekty a operacemi.
   (d) Tam, kde je to vhodné, uplatníme dědičnost.
   (e) Podrobnou analýzou dojdeme k návrhu implementace jednotlivých objektů.
5. Rekurzivním opakováním kroků 3, 4 a 5 dokončíme návrh.

Podívejme se nyní na tento postup podrobněji.

### Neformální popis architektury

První tři kroky při objektově orientovaném návrhu se v podstatě neliší od předchozích, „neobjektových" postupů. Začneme u specifikace problému a pak na základě popisu předběžných požadavků určíme neformální strategii implementace.

Dále budeme muset určit třídy, které k řešení daného problému použijeme, a jejich atributy a metody.

### Objekty a třídy

Při formalizaci strategie musíme na základě analýzy požadavků stanovit objekty a jejich třídy, které budeme v programu používat. Při určování tříd a jejich instancí si můžeme vypomoci tím, že v popisu odpovídají objektům zpravidla podstatná jména nebo fráze s významem podstatných jmen[10]. Při určování, co bude třída a co bude instance, si můžeme pomoci tím, že obecná podstatná jména, použitá v popisu, budou zpravidla představovat třídy objektů, zatímco konkrétní podstatná jména budou označovat instance. Při rozlišování těchto kategorií se samozřejmě musíme opírat o skutečný význam v daném kontextu (a ledacos si domyslet).

Jakmile jsme v popisu problému vyhledali všechna podstatná jména, sestavíme tabulku objektů. V ní vyznačíme, zda jde o objekt v prostoru problému (tedy v reálném světě) nebo v prostoru řešení (tedy v programu), a případně připojíme stručný komentář.

Při zjemňování řešení se může stát, že některé objekty vyloučíme jako nadbytečné nebo nesouvisející s problémem. Na druhé straně často musíme do návrhu přidat objekty nebo třídy, které se z popisu problému nedaly bezprostředně odvodit, ale jejichž potřeba vyplynula z analýzy problému.

### Atributy

Při určování vlastností objektů si můžeme pomoci tím, že v popisu strategie vyšetříme přídavná jména a mluvnické vazby s podobným významem a určíme, ke kterým objektům se vztahují. Vlastnostem budou zpravidla odpovídat atributy instancí. Často je ovšem třeba k atributům, popisujícím fyzikální a jiné vlastnosti předmětu z reálného světa, přidat další atributy, které usnadní softwarovou realizaci. Ty však obvykle specifikujeme až při podrobném návrhu.

### Metody

Dále musíme určit akce, které mohou objekty provádět, a operace, které na ně můžeme aplikovat. Je jasné, že v neformálním popisu strategie jim budou odpovídat slovesa a slovesné fráze. Přitom budeme brát v úvahu i predikáty (tvrzení jako *„je menší než cosi"*), neboť také odpovídají operacím s objekty - zjišťují jejich vlastnosti. Operace připojíme do tabulky k objektům.

Přitom se může stát, že se určitá operace vztahuje k více objektům. Jak potom určit, ke kterému ji připojit? Jako dostatečné vodítko obvykle poslouží následující pravidlo: operace bude součástí (metodou) toho objektového typu, jehož soukromé součásti využívá.

[10] Odvolávky na slovní druhy jsou samozřejmě jen pomůckou, která může usnadnit práci. Návrh samozřejmě nelze vytvořit (jen) na základě slovního rozboru zadání.

Jestliže narazíme na operaci, která vyžaduje přístup k soukromým částem několika tříd, znamená to nejspíš, že jsme udělali chybu ve specifikaci rozhraní některých tříd (typů) nebo ve specifikaci operace.

Výsledkem této fáze analýzy je tabulka objektů a operací. Přitom každému objektu by měla odpovídat alespoň jedna operace a každá operace by měla odpovídat nějakému typu. Pokud se stane, že se nějakého objektu nebude týkat žádná operace, nebo že nějakou operaci nebude možno přidělit žádnému z objektů, může to znamenat, že

- neformální strategie je neúplná a chybí v ní nějaký objekt nebo operace s ním;
- objekt nebo operaci, která patří do prostoru řešení, jsme zařadili do prostoru problému nebo naopak;
- přehlédli jsme, že některá operace, uvedená v tabulce, vyžaduje znalost „osamělého“ objektu;
- neformální strategie není popsána dobře - různé části popisu jsou na různé úrovni abstrakce.

V každém případě to znamená, že se musíme vrátit k popisu strategie a opravit jej.

#### Rozhraní: komunikace mezi objekty

Jakmile známe operace, které lze s objekty provádět, určíme zprávy, které si mohou objekty navzájem posílat. To znamená, že definujeme vztahy mezi metodami a zprávami, které metody volají. Zde již podrobnosti mohou záviset i na konvencích použitého programovacího jazyka.

(Představu komunikace modulů pomocí zpráv lze využít i v neobjektovém návrhu - usnadní např. testování návrhu pomocí scénářů.)

#### Podrobná analýza

Podrobný návrh, využívající OOP, je v mnoha ohledech velice podobný ostatním technikám návrhu. Navrhneme rozdělení programu do hlavních modulů. Dále vyjdeme od podrobného popisu rozhraní; zjemňujeme a zpřesňujeme datové struktury; navrhneme algoritmy pro jednotlivé moduly programu.

Objektově orientovaný návrh ovšem umožňuje kdykoli rekurzivně aplikovat výše uvedený postup, neboť objekt na určité úrovni abstrakce se může skládat z dalších objektů, které zabezpečují jeho funkčnost, podobně jako operace (metoda) se může skládat z řady jednodušších operací.

Často se uplatňuje takovéto pravidlo: Jestliže implementace určité operace vyžaduje příliš velké množství kódu (řekněme nad 200 řádků - to samozřejmě není závazná hodnota), vezmeme její popis jako nové zadání a opakujeme výše popsaný proces.

#### Testování návrhu

V tomto stádiu můžeme předběžně testovat schopnost produktu vyhovět požadavkům, které se na něj kladou. Používají se k tomu „scénáře“, který se skládá ze zpráv posílaných objektům. Pro různé úrovně abstrakce, a tedy různé úrovně podrobnosti návrhu, je samozřejmě třeba použít různé scénáře.

#### Dědičnost

Dědičnost umožňuje opakované používání již hotového kódu. Na možnost využití dědičnosti narazíme jak při návrhu shora dolů tak i při cestě opačné.

Při postupu „shora dolů“ přecházíme od abstraktnějších, méně specifických pojmů (např. *lod'*) k pojmům konkrétnějším, které přesněji určují vlastnosti objektů (např. *plachetnice*). Při definici potomka, odvozené třídy, si musíme všimnout, které metody může potomek zdědit a které je třeba překrýt novou verzí („předefinovat“).

Může se stát, že si při zjemňování návrhu všimneme společných operací a/nebo společných dat v několika třídách; v takovém případě se můžeme pokusit spojit je v návrhu nové třídy. Někdy se stane, že objekty

(instance) takto vzniklé třídy nemají v programu žádný význam - jediným smyslem takové třídy je, že poskytuje společná data a metody svým potomkům. Takovéto třídy označujeme jako *abstraktní*.[11]

V příkladu 11.1 na začátku této kapitoly bude loď abstraktní třída, neboť nemá smysl používat v programu obecnou loď; vždy to bude pouze *parník* nebo *plachetnice*.

### 11.2.1 Příklad: jednoduchý grafický editor

Počítačová grafika patří mezi nejjednodušší aplikace OOP. Proto se s těmito příklady setkáme téměř v každé publikaci, která se o OOP alespoň zmiňuje. Tento text nebude výjimkou. Na následujícím příkladu si ukážeme postupné zjemňování návrhu.

**Definice problému**: Vytvořit jednoduchý grafický editor.

**Specifikace požadavků**: *Editor má umožnit manipulovat na obrazovce s dvourozměrnými čarami, kuželosečkami a aproximačními křivkami. Uživatel bude pomocí myši (nebo jiného „ukazovátka“) přemísťovat a otáčet grafické objekty, měnit jejich velikost a barvu.*

Tato specifikace je velice neurčitá. Uživatelské rozhraní aplikace je specifikováno jen zhruba, o výstupu na jiná zařízení, o ukládání vytvořených obrázků apod. se v něm nehovoří vůbec. (Všechny tyto problémy si nyní dovolíme velkoryse pominout.) Lze ji však považovat neformální popis strategie implementace. Naznačuje, že obrázek se má skládat z jednotlivých „primitivních“ objektů, se kterými lze samostatně manipulovat.

Zpřesňování a formalizaci strategie provedeme v několika průchodech.

#### 1. zjemnění

Určení tříd: Mezi podstatnými jmény v popisu požadavků můžeme vynechat *uživatele* a *obrazovku*, neboť se bezprostředně netýkají implementace samotné - spíše určují její okolnosti. Také *myš* je na této úrovni analýzy nezajímavá.

*Editor* bude třída, která bude mít na starosti uživatelské rozhraní a která bude obsahovat seznam existujících grafických objektů. Nebudeme se s ní zde zabývat; za prvé zadání neobsahuje bližší specifikaci a za druhé příklad bude i tak dosti dlouhý.

Zbývají nám *čáry, kuželosečky a aproximační křivky*. Ty lze ale všechny na nejvyšší úrovni shrnout pod označení *grafický objekt*. Další podstatná jména, se kterými se v popisu setkáváme, *velikost, poloha a barva*, vyjadřují vlastnosti objektů.

*Grafický objekt* (*GO*) bude tedy představovat abstraktní třídu, která ponese vlastnosti společné všem třídám. (Vzhledem k tomu, že zatím uvažujeme o jediné třídě, nemá smysl pořizovat tabulku objektů.)

**Vyhledání atributů jednotlivých tříd**: GO je grafický objekt na obrazovce počítače. Je tedy zřejmé, že bude mít *barvu* a *polohu*. Barva bude určena jedním celým číslem, poloha dvojicí celých čísel (souřadnic referenčního bodu, např. středu, na obrazovce).

Další vlastnosti, které připadají v úvahu, jsou *orientace* (vyplývá z požadavku na otáčení objektů) a *velikost*. Z požadavků je zřejmé, že tyto vlastnosti budeme potřebovat. Jak orientace tak i velikost budou určeny jedním celým číslem.

**Určení operací s jednotlivými objekty (metod)**: Operace odvozujeme od sloves. Podle souvislosti ale musíme často přidat i operace, o kterých se v požadavcích přímo nehovoří.

GO budeme *přemísťovat, otáčet a měnit jejich velikost*. Vedle toho musíme samozřejmě mít možnost GO *vytvořit* a *zrušit* (odstranit). První tři operace představují vlastně změnu některého z atributů GO. Při práci

[11] V některých programovacích jazycích - např. v C++ - je termín *abstraktní třída* používán pro třídy, které mají alespoň jednu *čistě virtuální metodu*, tedy metodu, kterou (zhruba řečeno) sice deklarujeme, ale neimplementujeme. Taková metoda pouze "drží místo" pro metody stejného jména, které musíme implementovat v odvozených třídách. Podrobnější informace o čistě virtuálních metodách v C++ najdete např. v [18]. Současné verze Turbo Pascalu čistě virtuální metody neznají.

s GO musíme umět v programu také získat informace o GO - přesněji grafický objekt musí umět vrátit informaci o své velikosti, poloze a orientaci.

Přehled potřebných metod můžeme uspořádat do následující tabulky:

| Metoda | Význam |
|---|---|
| Vytvoř_GO | Vytvoří grafický objekt |
| Zruš_GO | Zruší grafický objekt |
| Nastav_polohu | Nastaví souřadnice x,y referenčního bodu GO |
| Zjisti_polohu | Zjistí souřadnice x,y referenčního bodu GO |
| Nastav_orientaci | Nastaví orientaci GO |
| Zjisti_orientaci | Zjistí orientaci GO |
| Nastav_velikost | Nastaví velikost GO |
| Zjisti_velikost | Zjistí velikost GO |
| Nastav_barvu | Nastaví barvu GO |
| Zjisti_barvu | Zjistí barvu GO |

Tato tabulka vlastně definuje protokol třídy GO.

**Komunikace mezi objekty (rozhraní)**: Předpokládejme pro určitost, že použijeme Turbo Pascal. Potom můžeme první dvě zprávy implementovat jako konstruktor a destruktor. Destruktor bude bez parametrů; parametry konstruktoru budou hodnoty atributů nového objektu.

Vzhledem k tomu, že objekty budou určitě vytvářeny a rušeny dynamicky, použijeme volání konstruktoru v proceduře *New* a volání destruktoru v procuduře *Dispose*.

Metody *Zjisti_velikost*, *Zjisti_barvu* a *Zjisti_orientaci* budou funkce bez parametrů, vracející velikost, barvu a orientaci grafického objektu[12]. Metody *Nastav_velikost*, *Nastav_barvu* a *Nastav_orientaci* budou procedury s jedním parametrem, vyjadřujícím odpovídající veličiny.

Metoda *Nastav_polohu* bude procedura s parametrem (parametry), vyjadřujícím polohu objektu na obrazovce; metoda *Zjisti_polohu* bude nejspíš - vzhledem k omezením Turbo Pascalu - také procedura, jejíž parametr (parametry), popisující polohu, se budou předávat odkazem.

**Test návrhu pomocí scénáře**: Ověříme, zda každému z požadavků, kladených na náš editor, odpovídá nějaká zpráva:

Vytvoření a zrušení GO (ugo je ukazatel na GO):

```
New(ugo, Vytvoř_GO(parametry));
Dispose(ugo, Zruš_GO);
```

Otočení GO:

```
ugo^.Nastav_orientaci(t);
```

Změna velikosti GO:

```
ugo^.Nastav_velikost(v);
```

Přesun GO:

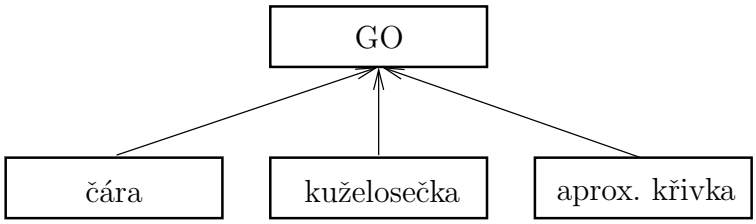
```
ugo^.Nastav_polohu(x,y);
```

Změna_barvy:

```
ugo^.Nastav_barvu(b);
```

**Úvahy o dědičnosti** jsou zatím nemístné, neboť na současné úrovni abstrakce máme jedinou třídu.

[12] V této fázi návrhu zatím nehovoříme o tom, jak budeme barvu, velikost, orientaci nebo polohu grafických objektů reprezentovat. Např. barva je pro nás zatím prostě jakýsi abstraktní typ, který popisuje barvu grafických objektů v navrhovaném editoru.




Obr. 11.3: Hierarchie tříd na 2. úrovni zjemnění

**2. zjemnění**

Nyní zopakujeme předchozí kroky a přitom zjemníme rozlišení typů grafických objektů. (Vlastně tak postoupíme na další úroveň abstrakce.)

**Třídy**: Čáry, kuželosečky a aproximační křivky jsou různé druhy grafických objektů. Definujeme je tedy jako samostatné třídy, které budou samozřejmě potomky třídy GO.

*V další analýze se budeme pro stručnost zabývat pouze kuželosečkami. Čtenář se může pokusit dokončit rozbor i pro zbývající dvě třídy.*

**Atributy**: Kuželosečka je obecně popsána kvadratickou rovnicí druhého stupně

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

Atributy kuželosečky tedy budou koeficienty $a, \ldots, f$. Změní-li se některý z těchto koeficientů, změní se kuželosečka - jinými slovy každá kuželosečka musí mít svou vlastní šestici koeficientů. To znamená, že půjde o atributy instancí.

V této fázi návrhu se již také můžeme rozhodnout, jak implementujeme barvu (budeme ji reprezentovat celými čísly, výčtovým typem ...) a další atributy grafických objektů.

**Metody**: K metodám, společným všem GO, musíme přidat metody pro nastavení a zjištění koeficientů. Protože třída kuželoseček má také jinou datovou strukturu než třída GO, musíme pro ni definovat zvláštní metody pro vytvoření a zrušení instance. V protokolu budou tedy navíc tyto zprávy:

| Metoda | Význam |
|---|---|
| Nastav_koeficienty | Nastaví koeficienty kuželosečky |
| Zjisti_koeficienty | Zjistí koeficienty dané kuželosečky |
| Vytvoř_kuželosečku | Konstruktor kuželosečky |
| Zruš_kuželosečku | Destruktor kuželosečky |

**Rozhraní**: Konstruktor koželosečky bude mít stejné parametry jako konstruktor obecného GO a k tomu navíc parametry $a, \ldots, f$.

Také metody *Nastav_koeficienty* a *Zjisti_koeficienty* budou mít parametry $a, \ldots, f$; v případě metody *Zjisti_koeficienty* je musíme předávat odkazem.

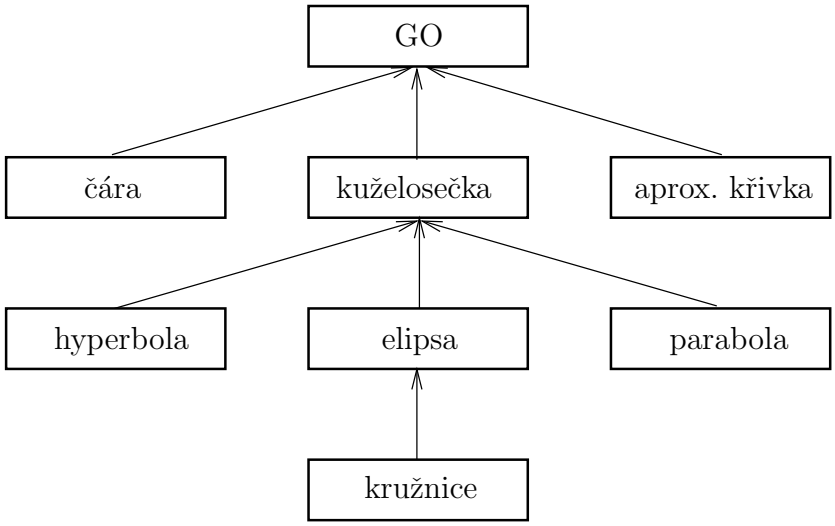
**Test pomocí scénáře** přenecháváme čtenáři.

**Dědičnost**: Je zřejmé, že *čára, kuželosečka* a *aproximační_křivka* budou třídy odvozené od *GO*. Mezi těmito třídami nemá smysl o dědičnosti uvažovat. Současný tvar dědické hierarchie[13] vidíte na obr. 11.3.

**3. zjemnění**

Dále se pro jednoduchost budeme zabývat pouze třídou kuželosečka.

[13] V diagramech, které popisují dědické hierarchie objektů, je obvyklé, že šipka směřuje od potomka k předkovi.




Obr. 11.4: Hierarchie tříd na 3. úrovni zjemnění

**Třídy**: Zadávání kuželoseček pomocí koeficientů $a, \ldots, f$ není nejpohodlnější. K vyjádření tvaru se spíše používají jiné parametry (např. souřadnice středu, velikosti a orientace poloos apod.), které se ovšem liší podle druhu křivky.

To znamená, že třídu *kuželosečka* bude rozumné dále rozdělit na podtřídy *kružnice, elipsa*, *parabola* a *hyperbola*; degenerované kuželosečky (např. dvojici různoběžek) nemá samozřejmě smysl brát v úvahu.

Dále své povídání zúžíme pouze na kružnici, rozbor ostatních kuželoseček přenecháme čtenáři.

**Atributy**: Kružnice je určena středem a poloměrem. Do třídy *kružnice* tedy přidáme nový atribut *poloměr*.

**Metody**: Protože jsme přidali nový atribut, budeme potřebovat i metody pro manipulaci s ním. Nazveme je *Nastav_poloměr* a *Zjisti_poloměr*. Z téhož důvodu definujeme také nový konstruktor a destruktor ( *Vytvoř_kružnici* a *Zruš_kružnici* ) a novou verzi metody pro určení koeficientů.

**Rozhraní**: Konstruktor kružnice bude mít jako vstupní parametry polohu středu, barvu a poloměr - vše celočíselné hodnoty. Destruktor nebude mít - podobně jako u ostatních uvažovaných tříd - žádné parametry. Funkce *Zjisti_poloměr* bude bez parametrů a bude vracet celé číslo; procedura *Nastav_poloměr* bude mít jeden celočíselný parametr, a to hodnotu poloměru.

**Test pomocí scénáře** přenecháváme opět čtenáři.

**Dědičnost**: Vzhledem k tomu, že kružnici lze považovat za zvláštní případ elipsy, může být vhodné definovat třídu *kružnice* jako potomka třídy *elipsa*. Výsledkem bude hierarchie tříd, znázorněná na obr. 11.5.
Nyní bychom mohli pokračovat ve zjemňování: elipsa je vlastně zvláštním případem eliptického oblouku, podobně jako třeba parabola je zvláštním případem parabolického oblouku atd.

Dále je třeba navrhnout algoritmy pro nakreslení oblouku kuželosečky, implementovat je atd. Z hlediska výkladu objektově orientovaného návrhu bychom se však nesetkali již s ničím novým.

# Literatura


[1] Wirth, N.: Algoritmy a štruktúry údajov. Alfa, Bratislava 1987

[2] Horowitz, E. - Sartaj, S.: Fundamentals of Computer Algorithms. Pittman Publishing Ltd., 1978

[3] Knuth, D.E.: The Art of Computer Programming. Vol II, Seminumerical Algorithms. Addison Wesley Publishing Comp., 1969.

[4] Knuth, D.E.: The Art of Computer Programming. Vol III, Sorting and Searching. Addison Wesley Publishing Comp., 1973. (Obě předchozí knihy, [3] a [4], existují též v ruském překladu Iskusstvo programirovanija dlja EVM. Vydal Mir, Moskva 1978)

[5] Sedgewick, R.: Algorithms in C++. Addison Wesley Publishing Comp., 1992. (Tato kniha existuje i ve verzi pro jazyky Pascal a C.)

[6] Barron, D.W.: Rekurzívne metódy v programovaní. Alfa, Bratislava 1973.

[7] McConnell, S.: Code Complete. Microsoft Press, 1993.

[8] Suchomel, J.: Technologie strukturovaného programování. Kancelářské stroje 1987.

[9] Znám, Š. a kol.: Pohľad do dejín matematiky. Alfa, Bratislava 1986.

[10] ČSN 36 4030. Značky vývojových diagramů pro systémy zpracování informací. ÚNM, Praha 1974

[11] Kernighan, B.W. - Plauger, B.W.: The Elements of Programming Style. McGraw - HillBook Comp., 1978

[12] Pressman, R.S.: Software engineering.A Practitioner's Approach. McGraw - Hill, 1987

[13] Maguire, S.: Writing Solid Code. Microsoft Press, 1993

[14] Jackson, M.A.: Principles of Program design. Academic Press, 1975

[15] Jackson, M. A.: System Development. Prentince Hall 1983

[16] Borland Visual Solutions Pack for Windows v.1.0. User's Guide. Borland International Inc., 1993

[17] Kašpárek, F. - Minárik, M. - Nikolov, V. - Pecinovský, R. - Virius, M.: Borland C++. Co v manálu nenajdete. Unis, Brno 1993

[18] Virius, M.: Programovací jazyky C/C++. G-Comp, Praha 1992

[19] Virius, M.: Základy programování (Úvod do Turbo Pascalu). ČVUT, Praha 1991 (skriptum)

[20] Stroustrup, B.: Design and Evolution of C++. Addison-Wesley, 1994

[21] Seige, V.: Kdy bude poražen Gari Kasparov? Softwarové noviny 1/1994, str. 26

[22] Shell, D. L.: A Highspeed Sorting Procedure. Communications of the ACM, 2, No. 7 (1959), s. 30

[23] Williams, J.W.J.: Heapsort (Algorithm 232). Communications of the ACM, 7, No. 6 (1964), s. 347

[24] Hoare, C.A.R.: Quicksort. Comp. Journal, 5, No. 1 (1962), s. 10

[25] Hoare, C.A.R.: Proof of a Programm: FIND. Communications of the ACM, 13, No. 1 (1970), s. 39

[26] Reiterman, J.: Analýza algoritmů. ČVUT Praha, 1991 (skriptum)

[27] Aho, A.V. - Hopcroft, J.E. - Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison - Wesley, Reading 1974; ruský překlad Mir, Moskva 1979

[28] Gilstadt, R. L.: Polyphase Merge Sorting - An Advanced Technique. Proc. AFIPS Eastern Jt. Comp. Conf., 18 (1960), 143

[29] ČSN 01 0181 Abecední řazení. Vydavatelství ÚNM, Praha 1977

[30] ANSI/IEEE std 754/1985: IEEE Standard for Binary Floating-Point Arithmetic. 1985

[31] Myers, G.: Composite Structured design. Van Nostrand, 1978

[32] Adelson-Velskij, G.M. - Landis, M.E.:Doklady Akademii nauk SSSR 146 (1962), 263.

[33] Cooley, J.M. - Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comp. 19, 1965, str. 297.

[34] Valášek, P.: Numerický koprocesor 8087.ČSVTS, Praha 1989

[35] Meyer, B.: Object-Oriented Software Construction. Prentince Hall, 1988

# Rejstřík