

Piotr Wróblewski

Algoritmy

**Computer Press
Brno
2015**

Algoritmy

Piotr Wróblewski

Překlad: Jakub Goner

Obálka: Martin Sodomka

Odpovědný redaktor: Martin Herodek

Technický redaktor: Jiří Matoušek

Přeloženo z originálu: Algorytmy, struktury danych i techniki programowania

Copyright © Helion 2013

Translation © Jakub Goner, 2015

Objednávky knih:

<http://knihy.cpress.cz>

www.albatrosmedia.cz

eshop@albatrosmedia.cz

bezplatná linka 800 555 513

ISBN 978-80-251-4126-7

Vydalo nakladatelství Computer Press v Brně roku 2015 ve společnosti Albatros Media a.s.
se sídlem Na Pankráci 30, Praha 4. Číslo publikace 19 054.

© Albatros Media a.s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována
a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného
souhlasu vydavatele.

1. vydání

 **ALBATROS MEDIA** a.s.

Obsah

Předmluva	11
Čím se tato kniha liší od jiných příruček?	11
Proč C++?	12
Jak číst tuto knihu?	12
Čím se budeme zabývat?	13
Kapitola 1: Dříve než začneme	13
Kapitola 2: Rekurze	13
Kapitola 3: Analýza složitosti algoritmů	13
Kapitola 4: Třídicí algoritmy	13
Kapitola 5: Datové typy a struktury	13
Kapitola 6: Odstraňování rekurze a optimalizace algoritmů	13
Kapitola 7: Vyhledávací algoritmy	13
Kapitola 8: Prohledávání textů	14
Kapitola 9: Pokročilé programovací techniky	14
Kapitola 10: Prvky algoritmiky grafů	14
Kapitola 11: Numerické algoritmy	14
Kapitola 12: Mohou počítače myslet?	14
Kapitola 13: Kódování a komprese dat	14
Kapitola 14: Různé úlohy	14
Přílohy	14
Ukázkové programy	15
Typografické konvence a symboly	15
Stručné poděkování	16
Poznámky ke čtvrtému původnímu vydání	16
Zpětná vazba od čtenářů	17
Zdrojové kódy ke knize	17
Errata	17
KAPITOLA 1	
Dříve než začneme	19
Kdysi dávno aneb střípky z historie algoritmických strojů	20
– 1801 –	21
– 1833 –	21
– 1890 –	22
– třicátá léta –	22

Obsah

– čtyřicátá léta –	22
– poválečné období –	22
– současnost –	23
Nedávná historie aneb počátky metodologie programování	23
Proces koncepce programů	24
Úrovně abstrakce popisu a výběr jazyka	24
Správnost algoritmů	26
KAPITOLA 2	
Rekurze	29
Definice rekurze	29
Ukázka principu rekurze	30
Jak pracují rekurzivní programy?	32
Rizika rekurze	34
Fibonacciho posloupnost	34
Stack overflow!	35
Další nástrahy	37
Cesta na věčnost	37
Správná definice nestačí	37
Typy rekurzivních programů	39
Rekurzivní myšlení	40
Příklad 1: Spirála	40
Příklad 2: „Sudé“ čtverce	42
Praktické poznámky k rekurzivním technikám	43
Úlohy	44
Řešení a poznámky k úlohám	47
KAPITOLA 3	
Analýza složitosti algoritmů	51
Definice a příklady	52
Funkce faktoriál znova	55
Nulování části pole	59
Chytáme se do pasti	61
Různé typy výpočetní složitosti	62
Nový úkol: zjednodušení výpočtů	64
Analýza rekurzivních programů	64
Terminologie a definice	64
Vysvětlení metody na příkladu	66
Logaritmický rozklad	67
Změna definičního oboru rekurzivní rovnice	68
Ackermannova funkce aneb něco pro labužníky	69
Výpočetní náročnost není modla	70
Techniky optimalizace programů	71
Úlohy	72
Řešení a poznámky k úlohám	72

KAPITOLA 4

Třídící algoritmy	75
Třídění přímým vkládáním, algoritmus třídy $O(N^2)$	76
Bublinkové třídění, algoritmus třídy $O(N^2)$	77
Quicksort, algoritmus třídy $O(N \log N)$	79
HeapSort – třídění haldou	82
Slučování setříděných množin	85
Třídění slučováním	86
Vnější třídění	87
Praktické poznámky	91

KAPITOLA 5

Datové typy a struktury	93
Základní a složené typy	93
Znakové řetězce a texty v jazyce C++	95
Abstraktní datové struktury	96
Jednosměrné seznamy	97
Implementace seznamů pomocí polí	122
Zásobník	127
Fronty FIFO	131
Haldy a prioritní fronty	134
Stromy a jejich reprezentace	140
Množiny	153
Úlohy	156
Řešení úloh	156

KAPITOLA 6

Odstraňování rekurze a optimalizace algoritmů	159
Jak funguje kompilátor?	160
Trocha formalizmu neuškodí	162
Několik příkladů odstraňování rekurze v algoritmech	163
Odstraňování rekurze s využitím zásobníku	166
Eliminace lokálních proměnných	167
Metoda opačných funkcí	169
Klasické postupy odstraňování rekurze	171
Schéma typu while	172
Schéma typu if-else	173
Schéma s dvojitým rekurzivním voláním	175
Shrnutí	177

KAPITOLA 7

Vyhledávací algoritmy	179
Lineární vyhledávání	179
Binární vyhledávání	180

Obsah

Hešování	181
Hledání funkce H	182
Nejznámější funkce H	183
Řešení kolizí	185
Návrat ke kořenům	186
Návrat k polím	187
Lineární pokusy	187
Dvojité hešování	189
Využití hešování	190
Shrnutí metod hešování	190
 KAPITOLA 8	
Prohledávání textů	193
Algoritmus vyhledávání hrubou silou	193
Nové vyhledávací algoritmy	195
Algoritmus K-M-P	196
Boyer-Mooreův algoritmus	200
Karp-Rabinův algoritmus	202
 KAPITOLA 9	
Pokročilé programovací techniky	205
Programování typu „rozděl a panuj“	206
Vyhledávání minima a maxima v číselném poli	207
Násobení matic s rozdíly $N \times N$	209
Násobení celých čísel	212
Jiné známé algoritmy typu „rozděl a panuj“	213
„Hladové“ algoritmy	213
Problém batohu aneb těžký život pěšího turisty	214
Dynamické programování	216
Fibonacciho posloupnost	218
Rovnice s více proměnnými	218
Nejdří společný podřetězec	220
Jiné programovací techniky	222
Bibliografické poznámky	225
 KAPITOLA 10	
Prvky algoritmiky grafů	227
Základní definice a pojmy	228
Cykly v grafech	231
Způsoby reprezentace grafů	233
Reprezentace pomocí pole	233
Slovniky uzelů	235
Seznamy kontra množiny	235
Základní operace s grafy	236
Součet grafů	236
Kompozice grafů	236
Mocnina grafu	237

Roy-Warshallův algoritmus	238
Floyd-Warshallův algoritmus	240
Dijkstrův algoritmus	243
Bellman-Fordův algoritmus	245
Minimální rozpínavý strom	245
Kruskalův algoritmus	246
Primův algoritmus	247
Prohledávání grafů	247
Strategie „do hloubky“ (sestupné prohledávání)	248
Strategie „do šířky“	249
Jiné strategie prohledávání	251
Problém vhodného výběru	252
Shrnutí	256
Úlohy	256
 KAPITOLA 11	
Numerické algoritmy	259
Vyhledávání nulových bodů funkcí	259
Iterativní výpočet hodnot funkce	261
Interpolace funkcí Lagrangeovou metodou	262
Derivování funkcí	263
Integrování funkcí Simpsonovou metodou	265
Řešení soustav lineárních rovnic Gaussovou metodou	267
Závěrečné poznámky	269
 KAPITOLA 12	
Mohou počítače myslet?	271
Přehled oblastí zájmu umělé inteligence	272
Expertní systémy	272
Neuronové sítě	274
Reprezentace problémů	276
Cvičení 1	277
Hry pro dvě osoby a stromy her	277
Algoritmus mini-max	278
 KAPITOLA 13	
Kódování a komprese dat	285
Kódování dat a aritmetika velkých čísel	287
Symetrické šifrování	287
Asymetrické šifrování	289
Primitivní metody	294
Luštění šifer	296
Techniky komprese dat	296
Komprese pomocí matematického modelování	298
Komprese metodou RLE	299

Obsah

Komprese dat Huffmanovou metodou	299
Kódování LZW	305
Princip slovníkového kódování na příkladech	305
Popis formátu GIF	308
KAPITOLA 14	
Různé úlohy	311
Texty úloh	311
Řešení úloh	313
PŘÍLOHA A	
Seznámení s jazykem C++	317
Prvky jazyka C++ na příkladech	317
První program	317
Direktiva #include	318
Podprogramy	318
Procedury	319
Funkce	319
Aritmetické operace	320
Logické operace	321
Ukazatele a dynamické proměnné	322
Odkazy	323
Složené typy	323
Pole	323
Záznamy	324
Příkaz switch	324
Iterace	325
Rekurzivní struktury	326
Parametry programu main()	326
Operace se soubory v jazyce C++	326
Objektové programování v jazyce C++	327
Terminologie	328
Objekty na příkladech	328
Statické složky tříd	331
Konečné metody tříd	332
Dědičnost vlastností	332
Podmíněný kód v jazyce C++	335
PŘÍLOHA B	
Úvod do číselných soustav	337
Několik definic	337
Dvojková soustava	337
Aritmetické operace s dvojkovými čísly	339
Logické operace s dvojkovými čísly	339
Osmičková soustava	341

Šestnáctková soustava	341
Proměnné v paměti počítače	342
Kódování znaků	343
PŘÍLOHA C	
Kompilování ukázkových programů	347
Obsah archivu ZIP ke stažení	347
Bezplatně dostupné kompilátory C++	347
Kompilace a spouštění	348
GCC	348
Visual C++ Express Edition	349
Dev C++	353
Literatura	355
Rejstřík	357

Předmluva

V posledních několika desetiletích přinesl vědní obor zvaný algoritmika mnoho efektivních nástrojů, které umožňují řešit nejrůznější problémy pomocí počítačů. Někteří k algoritmice přistupují jen jako k jistému druhu kuchařské knížky, kde mohou v případě potřeby najít vhodné „recepty“. Jiní uživatelé díky ní rozvíjejí své schopnosti efektivního řešení problémů a učí se kreativně uvažovat. Při psaní této knihy jsem se snažil oba tyto přístupy spojit a představit dostatečně širokou škálu algoritmických témat, ale zároveň nezůstávat pouze na povrchu. Na popisovaných problémech jsem chtěl čtenářům ukázat vhodnou perspektivu možných aplikací počítačů a neomezovat se přitom na matematické vzorce a suchý kód ukázkových programů.

Čím se tato kniha liší od jiných příruček?

V předchozích vydáních sklidila tato kniha značný úspěch a získala mnoho věrných čtenářů. Ze zpětného pohledu vidím, že za své pozitivní přijetí vděčí prostému jazyku (pokud možno jsem se snažil využíbat složitým matematickým vzorcům a zároveň jsem teoretické poučky ilustroval příklady i skutečným kódem v jazyce C++). Zejména zmíněné ukázky kódů představovaly příloženou „trefu do černého“, protože čtenáři už měli dost příruček plných pseudokódu, který se nedal snadno přizpůsobit specifikám komplikátorů a požadavkům operačních systémů. V tomto vydání jsem se rozhodl aktualizovat zdrojové kódy ukázkových programů, aby šly snadno spustit v oblíbených bezplatně dostupných komplikátorech (Microsoft Visual C++ Express Edition a GCC/Dev C++). Doplnil jsem také řadu praktických rad týkajících se vlastního procesu komplikace, které by měly pomoci začínajícím adeptům programátorského řemesla (příloha C).

Samozřejmě si uvědomuji, že v rámci jedné knihy nelze prezentovat vše, co je v oboru algoritmiky klíčové. S ohledem na rozsah celé problematiky nemůže být výběr témat úplný. Knihu jsem sice koncipoval jako logický celek, ale může se stát, že některé čtenáře rozčaruje, zatímco jiné naopak značným rozsahem řešených otázek odradí. Snažil jsem se sestavit přehled algoritmických problémů tak, aby byl dostatečně reprezentativní a užitečný tém čtenářům, kteří se chtějí programováním zabývat profesionálně. Pokud si po přečtení této knihy budou chtít prostudovat jiné podobné publikace, najdou jejich seznam na konci knihy. Nesmí se však divit, když narazí na publikace plné matematických rovnic nebo se spoustou pseudokódu místo skutečných programů, které by se daly snadno komplikovat a testovat. Nicméně doufám, že po přečtení tohoto průvodce si snáze poradí i s objemnějšími svazky, které bohužel bývají dosti hermetickým jazykem.

Svou knihu doporučuji zejména osobám, které se zajímají o programování, ale zatím nemají dostatečné teoretické základy. Vzhledem k tomu, že zahrnuje poměrně početnou skupinu otázek z oblasti informatiky, dobře poslouží rovněž jako repetitorium pro ty, kdo se programováním již živí. Kniha je každopádně určena lidem, kteří se s programováním již alespoň setkali a rozumí

Předmluva

základním pojmem jako *proměnná*, *program*, *kompilace* atd. Na termínech tohoto typu je založen celý výklad a jejich vysvětlováním se nebudeme zdržovat.

Proč C++?

Rozhodně musíme věnovat několik slov programovacímu jazyku, který v této knize slouží k prezentaci algoritmů. Výběr padl na moderní a módní jazyk C++, který se díky svému přesnému zápisu a modularitě hodí k programování praktických aplikací. Jazyk C++ se samozřejmě vyznačuje i řadou nedostatků. Zejména postrádá mechanizmy, které by vynucovaly dobrý programátorský styl (programátoři mají občas příliš velkou svobodu volby). Z druhé strany žádný jiný jazyk nenabízí zároveň neomezené možnosti objektového programování a v případě potřeby i využití nízkourovňových mechanismů. Tato volnost se hodí například v systémovém programování nebo programování na úrovni zařízení (při tvorbě ovladačů), nemluvě již o hrách.

Na tomto místě bychom však měli zdůraznit, že samotný jazyk prezentace algoritmu nemá na jeho fungování zásadní vliv – jedná se pouze o nástroj, který tvoří jakýsi vnější obal a může se měnit v závislosti na současné módě. Jazyk C++ byl pro účely této knihy vybrán díky své značné oblibě mezi programátory. Ti, kdo se s tímto jazykem setkali teprve nedávno, tedy mají dokonalou přiležitost, aby se seznámili s jeho možnostmi. Programátoři, kteří zatím pracovali výhradně s jazykem Pascal, najdou v příloze rychlokurz jazyka C++, s jehož pomocí by si měli základní rozdíly obou jazyků rychle osvojit. Syntaxe jazyka C++ se natolik podobá jiným jazykům, že s přizpůsobením algoritmů jistě nebudou mít potíže ani osoby, které dávají přednost jazyku Java či nedávným „vynálezům“ typu C# (podle názoru autora tento jazyk, který kombinuje prvky několika jiných, vnáší pouze zbytečný chaos).

Natolik komplexnímu jazyku, jako je C++, se samozřejmě nelze naučit jen z krátké přílohy, kterou jsme pro tento účel vyhradili. Příloha by měla pouze překonat případnou syntaktickou bariéru, aby byly publikované výpisy kódů dostatečně srozumitelné. Čtenáři, kteří chtějí své schopnosti programování v C++ zdokonalit, by měli sáhnout po jedné z příruček uvedených v seznamu literatury. S ohledem na srozumitelnost výkladu lze doporučit zejména knihu [Eck00]. Ambičiozní programátoři by si měli povinně prostudovat dílo [Str97], jehož autorem je samotný tvůrce jazyka C++ Bjarne Stroustrup a které představuje kompletní zdroj informací o tomto jazyce.

Jak číst tuto knihu?

Čtenáři, kteří se v popisované problematice již orientují, mohou při čtení podle momentálních potřeb libovolně přeskakovat.

Začínajícím programátorům lze doporučit, aby procházeli jednotlivé kapitoly v původním pořadí. Kvůli snadnějšímu vyhledávání potřebných informací kniha obsahuje podrobný rejstřík.

Na konci mnoha kapitol se nacházejí úlohy, které tematicky souvisejí s právě popisovanou problematikou. U většiny úloh je uvedeno také jejich řešení s podrobnými komentáři.

Poslední kapitola zahrnuje sadu různorodých úloh, které se nevešly do předechozího výkladu. Než se pustíte do jejich řešení, důkladně si projděte obsah všech předechozích kapitol.

Čím se budeme zabývat?

Kvůli snazší orientaci v různých tématech, o nichž se kniha zmiňuje, shrňme nyní hlavní motivy jednotlivých kapitol. Následující popis vychází z obsahu knihy a navíc uvádí, co lze v každé kapitole najít.

Kapitola 1: Dříve než začneme

Rozsáhlý úvod, kde se můžeme pozvolna připravit na praktické programování. V této kapitole se seznámíme s mnoha důležitými historickými fakty, které se týkají vývoje algoritmiky, a porozumíme tomu, z čeho vychází současný postup v této oblasti.

Kapitola 2: Rekurze

Tato kapitola se věnuje jednomu z nejdůležitějších mechanizmů, které se při programování používají – rekurzi. Kromě výhod této programovací techniky objasňuje také její vady, které nebývají zcela zjevné. Problematiku přibližují jednodušší i obtížnější příklady a čtenáři mohou vyřešit zajímavé grafické úlohy.

Kapitola 3: Analýza složitosti algoritmů

Přehled nejpopulárnějších a nejjednodušších metod, které umožňují určit výpočetní náročnost algoritmů, aby je bylo možné porovnat a vybrat z nich ten nejfektivnější. Tato kapitola je určena hlavně studentům informatiky, i když by se na ni měly alespoň zběžně podívat i osoby, které se zajímají o programování obecně.

Kapitola 4: Třídicí algoritmy

Představuje nejoblíbenější a nejznámější třídicí postupy. Celou tematiku samozřejmě nelze poštihnout v jediné kapitole. Měla by však čtenáře povzbudit k dalšímu studiu mimořádně zajímavé oblasti třídicích algoritmů, která má kromě svého praktického uplatnění také značný význam didaktický. Podrobně představíme jak prosté metody typu třídění vkládáním či bublinkového třídění, tak i složitější algoritmy, kde se soustředíme na podrobný popis metody *Quicksort*.

Kapitola 5: Datové typy a struktury

Pojednává o běžných datových strukturách (seznamy, fronty, binární stromy atd.) a o jejich programové implementaci v jazyce C++. Zvláštní pozornost věnujeme ukázkám možného uplatnění představených datových struktur.

Kapitola 6: Odstraňování rekurze a optimalizace algoritmů

Předkládá způsoby, kterými lze převádět rekurzivní programy na jejich iterativní verze. Tato kapitola má vysoký technický charakter a je určena programátorům, kteří se zajímají o problematiku optimalizace programů.

Kapitola 7: Vyhledávací algoritmy

Tato kapitola využívá několik již dříve představených metod, které souvisejí s vyhledáváním prvků ve slovníku, a následně podrobně popisuje metodu hešování (ang. *hashing*).

Kapitola 8: Prohledávání textů

S ohledem na význam tématu byly algoritmy prohledávání textů soustředěny do samostatné kapitoly. Konkrétně jsou vysvětleny metody prohledávání *hrubou silou*, *K-M-P*, *Boyer-Moorova* a *Karp-Rabinova*.

Kapitola 9: Pokročilé programovací techniky

Dlouholetý výzkum na poli algoritmiky vedl k objevu skupiny metod obecného typu: dynamického programování, postupů typu „rozděl a panuj“ a „hladových“ algoritmů (ang. *greedy*). Tyto *metaalgoritmy* značně rozšiřují oblast možného nasazení počítačů při řešení problémů.

Kapitola 10: Prvky algoritmiky grafů

Popis jedné z nejzajímavějších datových struktur, se kterými se v informatice můžeme setkat. Grafy usnadňují (a občas vůbec umožňují) řešit mnoho úloh, které by při nasazení jiných datových struktur vypadaly neřešitelně.

Kapitola 11: Numerické algoritmy

Tato kapitola představuje několik zajímavých výpočetních problémů, které ukazují možnosti využití počítačů v matematice, konkrétně v přibližných výpočtech nulových bodů funkcí, interpolaci, derivování, integrování, *Gaussově* metodě atd.

Kapitola 12: Mohou počítače myslit?

Úvod do značně rozsáhlé oblasti, která se označuje jako *umělá inteligence*. Vymezíme oblast zájmu tohoto oboru a ukážeme příklad programové implementace algoritmu *Mini-Max*, který je oblíbený v teorii her.

Kapitola 13: Kódování a komprese dat

Podrobně rozebereme populární metody šifrování a komprese dat. V této kapitole se seznámíme s pojmy symetrického a asymetrického šifrování a podrobně také popíšeme systém šifrování s veřejným klíčem (*RSA*). Při té příležitosti také předvedeme, jak se provádějí aritmetické operace na značně velkých celých číslech. Při rozboru komprese dat začneme od teoretických základů a jednoduchých metod. Detailně také popíšeme slavné algoritmy komprese, které využívají *Huffmanovu* metodu a metodu *LZW*.

Kapitola 14: Různé úlohy

Sada různorodých úloh, které se nevešly do hlavního textu knihy. Nechybí samozřejmě ani jejich řešení.

Přílohy

Příloha A popisuje základy syntaxe jazyka C++ (formou srovnání s jazykem Pascal, který se často využívá k vyjádření algoritmů). Příloha B vysvětluje počítání v jiných číselných soustavách (dvojkové, osmičkové a šestnáctkové). Tyto znalosti se hodí každému zájemci o programování.

Příloha C obsahuje pokyny týkající se komplikování a spouštění ukázkových programů pomocí kompilátoru jazyka C++, který je součástí sady GCC (GNU Compiler Collection), a v prostředí Microsoft Visual C++ Express Edition.

Ukázkové programy

Kódy programů z této knihy jsou k dispozici ke stažení na stránce knihy: <http://knihy.cpress.cz/K2114>. Zdrojové soubory jsou oproti své tištěné podobě zpravidla delší a rozvinutější, protože kódy v knize kvůli úspoře místa často nezahrnují podrobné komentáře ani standardní hlavičkové soubory. Pokud tedy během výkladu představíme nějakou funkci a explicitně neuvedeme, jakým způsobem se používá, určitě najdete reprezentativní příklad jejího uplatnění ve zdrojovém souboru (s ukázkovou funkcí `main` a sadou hlavičkových funkcí). Při čtení knihy je proto vhodné porovnávat přetisklé verze programů se staženými soubory.

Programy byly otestovány pomocí kompilátoru GCC (ověřeny v systémech Windows Vista i Linux – varianta Ubuntu) a také v prostředí Microsoft Visual C++ Express Edition (praktické programovací prostředí pro systém Windows, které je k dispozici zdarma). Všechny programy by měly fungovat rovněž s kompilátorem Dev C++, protože v praxi se jedná pouze o nadstavbu (prostředí IDE) kompilátoru GCC.

Grafické programy jsou kompatibilní pouze se systémem Windows, ale obsahují podrobné komentáře, takže by zkušenému programátorovi nemělo jejich přenesení do jiného grafického systému činit žádné potíže. (V textu se nachází tabulka s vysvětlením použitých grafických instrukcí. S její pomocí by analýzu ukázkových programů měly zvládnout i osoby, které s kompilátorem Visual C++ nikdy nepracovaly.)

Typografické konvence a symboly

Uvedme nyní několik standardních symbolů a konvencí, které se používají na stranách této knihy.

V prvé řadě platí pravidlo, že všechny výpisy a texty zobrazené na monitoru jsou od standardního textu knihy odlišeny neproporčním písmem. Toto pravidlo se vztahuje i na příkazy operačního systému, jsou-li uvedeny.



Poznámka: Tento text má zásadní význam pro pochopení vysvětlované problematiky.



Upozornění: Co nedělat, abychom se nedostali do problémů.

Definice nebo tvrzení.

Stručné poděkování

První vydání této knihy vzniklo díky mému několikaletému pobytu ve Francii, kde jsem dostal neopakovatelnou příležitost čerpat z bohatých fondů několika technických knihoven. Většina titulů, jejichž přečtení mě inspirovalo k napsání této knihy, byla tehdy v Polsku jen velmi těžko dostupná (pokud vůbec). Knihu jsem tvořil hlavně s cílem vyplnit jistou mezeru na místním vydavatelském trhu.

Pracovní verzi prvního vydání knihy prohlédl a opatřil mnoha cennými poznámkami Zbyszek Chamski. Druhé vydání pečlivě recenzoval Tomasz Zieliński a k jeho rozsáhlé korektuře jsem z větší části přihlédhl i ve třetím vydání. Toto čtvrté originální vydání (v češtině druhé) zohledňuje četné komentáře čtenářů, které se týkaly hlavně kvality kódu C++ a jeho kompatibility s oblíbenými kompilátory. Všem, kteří kontrole knihy věnovali svůj čas, srdečně děkuji.

Poznámky ke čtvrtému původnímu vydání

Celý text aktuálního vydání knihy jsem kompletně prohlédl a přepracoval. Snažil jsem se odstranit nahlášené nejasnosti, přepsal jsem nesrozumitelné pasáže, vylepšil jsem některé obrázky a opravil jsem také chyby v obsahu i výpisech, kterých si všimli pozorní čtenáři. Některé kódy jsem zkorigoval s ohledem na čistotu jazyka C++, aby je bylo možné bez problémů kompilovat pomocí různých kompilátorů i operačních systémů. Část opravených nedostatků vyplývala z historického vývoje (první vydání se objevilo již v roce 1995) a další část vznikla mou nepozorností nebo kvůli problémům při sazbě (text druhého vydání jsem převáděl z formátu LaTeX na formát Word a při přenosu dat došlo k chybám). Všem čtenářům předchozích vydání se za potíže se zdrojovými kódy upřímně omlouvám. Jako malé odškodnění jim v tomto vydání nabízím pomoc při vlastním programování – v příloze vysvětlují práci s kompilátorem GCC a s prostředím Visual C++ Express Edition.

Struktura knihy sice zůstala stejná, ale do několika kapitol jsem doplnil nové pasáže. Největšími změnami prošly kapitoly 3 (analýza složitosti algoritmů), 4 (přibylo vnější a systémové trídění), 5 (několik praktických pokynů ohledně kódu C++ užívaného v kontextu datových struktur) a 10 (nové grafové algoritmy).

Rozšířil jsem také sadu ukázkových programů, které jsou dostupné ke stažení. Kromě jiného jsem grafické programy z kapitoly 2 připravil i ve verzích pro systém Windows a s ohledem na rozšíření obsahu knihy jsem přidal několik nových programů: <http://knihy.cypress.cz/K2114>.

Doufám, že provedené změny přispějí ke kvalitě knihy, a přeji příjemnou a užitečnou četbu.

Piotr Wróblewski
květen 2009

Zpětná vazba od čtenářů

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press, které pro vás tuto knihu přeložilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

*Computer Press
Albatros Media a.s., pobočka Brno
IBC
Příkop 4
602 00 Brno*

nebo

sefredaktor.pc@albatrosmedia.cz

Computer Press neposkytuje rady ani jakýkoli servis pro aplikace třetích stran. Pokud budete mít dotaz k programu, obraťte se prosím na jeho tvůrce.

Zdrojové kódy ke knize

Z adresy <http://knihy.cypress.cz/K2114> si po klepnutí na odkaz Soubory ke stažení můžete přímo stáhnout archiv s ukázkovými kódy.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nelze. Pokud v některé z našich knih najdete chybu, ať už chybu v textu nebo v kódu, budeme rádi, pokud nám ji oznámíte. Ostatní uživatele tak můžete ušetřit frustrace a pomocí nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese <http://knihy.cypress.cz/K2114> po klepnutí na odkaz Soubory ke stažení.

1

KAPITOLA

Dříve než začneme

V této kapitole:

- Kdysi dávno aneb střípky z historie algoritmických strojů
- Nedávná historie aneb počátky metodologie programování
- Proces koncepce programů
- Úrovně abstrakce popisu a výběr jazyka
- Správnost algoritmů

Než začneme pracovat s pojmy, jako je v úvodu zmíněný „algoritmus“, měli bychom se pečlivě zamyslet nad jejich významem¹.

Algoritmus je:

- konečná posloupnost či sekvence pravidel, která je aplikována na konečnou množinu dat a umožňuje řešit třídu problémů podobného typu;
- sada pravidel, která je typická pro jisté informatické výpočty nebo činnosti.

Uvedené definice vypadají jasně a srozumitelně, avšak zahrnují natolik širokou oblast lidských aktivit, že postrádají dostatečnou přesnost. Od významu termínu algoritmus na chvíli odbočme k jeho původu, který byl dlouho záhadou. Teprve odborníci na historii matematiky našli nejpravděpodobnější vysvětlení jeho etymologie: pojem je odvozen od jména perského matematika Muhammada ibn Mūsā al-Chwārizmīho² (žil v 9. století n. l.). Tento učenec sestavil jednoznačná pravidla, která krok po kroku vysvětlovala zásady aritmetických operací s desetinnými čísly.

Slovo algoritmus se často spojuje se jménem řeckého matematika Eukleida (365–300 př. n. l.) a jeho slavným návodem na výpočet největšího společného dělitele (NSD) dvou čísel a a b:

vstupní data: a a b , pomocné proměnné: c , **zbytek**

dokud $a > 0$ **proved'**:

nahrad' číslo c zbytkem po dělení čísla a číslem b ;

nahrad' číslo b číslem a ;

nahrad' číslo a číslem c ;

nahrad' číslo **zbytek** číslem b ;

výsledek: **zbytek**.

¹ Definice je převzata ze slovníku *Le Nouveau Petit Robert* (Dictionnaires le Robert – Paris 1994) – vlastní překlad autora.

² Jeho jméno mělo v latině podobu *Algorismus*.

KAPITOLA 1 Dříve než začneme

Eukleidés samozřejmě svůj algoritmus nenavrhl přesně takto (místo funkce zbytku po dělení se využívalo následné odčítání). Jeho myšlenku však můžeme zapsat výše uvedeným způsobem, aniž by to ovlivnilo výsledek, který bude v každém případě stejný. Není to samozřejmě jediný algoritmus, s nímž jsme se v životě setkali. Každý z nás nepochybňě dokáže uvařit čaj:

- zapnout rychlovárnou konvici;
- přivést potřebné množství vody k varu;
- vložit do hrnku sáček s čajem;
- zalít sáček vroucí vodou;
- osladit podle chuti;
- počkat, až se čaj dostatečně vyluhuje.

Uvedený postup sice funguje, ale obsahuje několik slabých míst: co to znamená „potřebné množství vody“? Co se přesně myslí pokynem „osladit podle chuti“? Postup přípravy čaje má vlastnosti algoritmu (jak jej chápeme ve smyslu výše citovaných slovníkových definic). Chybí mu však přesnost, aby jej bylo možné uložit do nějakého stroje, který by si pak v každé situaci dokázal poradit s pokynem „připrav mi silný čaj“. (Jak například v praxi definovat podmínu, že se čaj „dostatečně vyluhoval“?)

Jaké vlastnosti by tedy měly splňovat algoritmy v kontextu informatiky? Na toto téma bychom mohli diskutovat značně dlouho, ale pokud se spokojíme s jistým zjednodušením, mohou nás uspokojit následující konkrétní požadavky.

Každý algoritmus:

- se vyznačuje vstupními daty (v množství nulovém nebo větším), která pocházejí z dobře definované množiny (např. Eukleidův algoritmus pracuje se dvěma celými čísly);
- dává určitý výsledek (nemusí být nutně číselný);
- je přesně definovaný (každý krok algoritmu musí být jednoznačně určen);
- je konečný (algoritmus musí někdy poskytnout výsledek – máme-li algoritmus A a vstupní data D , musíme být schopni přesně zjistit čas provedení $T(A)$);
- lze jej aplikovat na řešení celé třídy úloh, a nikoli jen jedné konkrétní úlohy.

Kromě toho se kvůli své netrpělivosti snažíme hledat efektivní algoritmy, tj. takové, které svou práci dokončí co nejrychleji a využívají co nejmenší množství paměti (k tomuto tématu se ještě vrátíme v kapitole 3). Než se však vrhneme na klávesnici, abychom začali do paměti počítače zadávat programy splňující uvedené požadavky, podívejme se ještě na algoritmiku z historické perspektivy.

Kdysi dávno aneb střípky z historie algoritmických strojů

Oba matematiky, jejichž jména jsme zmínili na začátku této kapitoly, dělí více než tisíc let. Mohli bychom se tedy snadno domnívat, že tento obor se dynamicky rozvíjel už v davných dobách. Po krok v oblasti algoritmiky však ve skutečnosti úzce souvisí s technickou revolucí, ke které došlo během posledních sotva dvou set let. Pokud však budeme informatiku a algoritmiku považovat za nedílný celek, který přirozeně vychází z početních systémů, pak bychom měli zmínit přínos Sumerů, kteří vynalezli počítací tabulky, vlastní kalendář i šedesátkový měrný systém (rozdělení dne na 24 hodin jsme převzali právě od nich). Víme také, že abakus neboli nejslavnější počítadlo v lidské historii vyvinuli Číňané. Obvykle si však neuvědomujeme, že k nějakému systému, který

usnadňoval počítání, dospěla téměř každá vyspělejší civilizace a těžko někomu přiznat palmu vítězství. Při zkoumání faktů tohoto typu se navíc snadno dostaneme do éry kolem 2–3 tisíc let př. n. l., takže se z dnešního pohledu jedná spíše o historické zajímavosti.

Abychom z této kapitoly neudělali učebnici dějepisu, odpustíme si úvahy na téma sčítacího stroje Blaise Pascala (kolem roku 1645) či obdobného mechanizmu, který téměř ve stejné době zkonstruoval G. W. Leibniz. Podívejme se jen na okolnosti vzniku několika vynálezů, které jednak usnadnily počítání, ale navíc umožňovaly i základní programování – tedy něco, co nám jednoznačně připomíná počítače a algoritmy.

– 1801 –

Francouz Joseph Marie Jacquard vynalezl tkalcovský stav, který umožňoval „programovat“ tkaný vzor pomocí speciálních děrovaných karet. Postup tkání závisel na algoritmu, který byl zakódován v podobě řady otvorů v příslušné kartě. Vynález byl výsledkem mnohaletého Jacquardova úsilí a konkrétní podobu získal vlastně náhodou – díky účasti ve státním konkuru, kde vynálezce představil stroj na výrobu rybářských sítí.

Jacquardova koncepce zaujala francouzského matematika L. M. Carnota, který konstruktéra povolal do Paříže, aby tam mohl pokračovat ve vývoji a snáze získat vládní stipendium. Na začátku prací vynálezce málem přišel o život, protože se jej rozrušení tkalcí pokusili utopit v řece Rhôně. Tušili totiž, že jeho stroj ohrozí jejich zaměstnání (přesněji řečeno zaměstnání jejich dětí, které do té doby pomáhaly přenášet nitky, aby bylo možné utkat odpovídající vzor pomocí příčně se pohybující cívky s nitkou – Jacquardův vynález odstranil pět pracovních míst u jednoho stavu!). Jacquardův nápad odpovídal tehdejším technickým možnostem, ale stojí za povšimnutí, že děrovaná karta se zakódovanou dvoustavovou logikou (v závislosti na přítomnosti či nepřítomnosti otvoru tkalcovský stroj provedl či neprovedl příslušnou mechanickou akci) představuje předchůdce současných pamětí, kde se jednotlivé kroky algoritmu kódují pomocí binárních číslic.

– 1833 –

Angličan Charles Babbage částečně dokončil stroj na počítání některých matematických funkcí. Žil v době, kdy se začaly široce uplatňovat matematické poznatky (například v astronomii a navigaci), ale přitom chyběly metody, které by umožnily počítání automatizovat. Babbage je autorem principu takzvaného *analytického stroje*, který vycházel z jeho předchozího výtvoru, ale kromě toho umožňoval měnit programy obdobně jako Jacquardův stav.

Zjednodušeně řečeno se tento stroj měl skládat z úložiště (forma paměti realizované jako souosé disky, později nahrazené bubnem), mlýna (výpočetní jednotka, která měla provádět operace díky otáčení disků a soukolí) a řídicího mechanizmu, který by využíval děrované karty (podle Jacquardova nápadu). Nepřipomíná to náhodou schéma dnešních počítačů?

Babbage svou koncepci rozvinul natolik důkladně, že matematicka Ada Lovelace, dcera lorda Byrona, pro toto zatím neexistující zařízení vytvořila první teoretické „programy“, a stala se tak první uznávanou programátorkou v historii informatiky³!

Kvůli mechanické náročnosti návrhu tohoto stroje byl jeho první prototyp postaven teprve dvacet let poté, co Babbage s jeho myšlenkou přišel, a kompletní stroj spatřil světlo světa teprve roku 1992. Koncem 20. století se samozřejmě jednalo spíše o kuriozitu, která k pokroku oboru už nijak nepřispěla.

³ Z jejího jména pochází název programovacího jazyka ADA.

– 1890 –

Prakticky poprvé byly veřejně a v rozsáhlém měřítku nasazeny přístroje založené na děrných štítcích. Šlo o stroj na zpracování statistických dat, jehož autorem byl Američan Herman Hollerith a který se uplatnil při zpracování dat ze sčítání lidu. (Stojí za povšimnutí, že Hollerithův podnik se roku 1911 transformoval na společnost *International Business Machines Corp.*, která je známější pod zkratkou IBM a dodnes patří mezi nejvýznamnější počítačové firmy.)

– třicátá léta –

Na rozvoji teorie algoritmů se podílela plejáda známých matematiků, mj. Turing, Gödel a Markow. Z té době pochází slavná otázka, kterou pruský⁴ matematik David Hilbert položil roku 1928 na mezinárodním matematickém kongresu. Zeptal se, zda existuje metoda, která by umožnila rozhodnout o pravdivosti libovolného matematického tvrzení pouze na základě mechanických operací se symboly. Studentům informatiky je asi povědomá koncepce tzv. *Turingova stroje*, což je abstraktní výpočetní stroj, který se skládá z čtecí a zapisovací hlavice a nekonečné pásky se symboly (například čísla nebo matematickými operátory). Tento matematický model posloužil jako teoretický základ při konstrukci moderních počítačů. Pro naše účely si stačí zapamatovat, že zařízení označované poněkud zavádějícím termínem *stroj* představuje pouze *model schématu fungování* podle zadaného algoritmu.

– čtyřicátá léta –

Vznikají první univerzální počítače (hlavně kvůli výpočtům, které vyžadoval tehdejší válečný konflikt: souvisely s dešifrováním kódů či konstrukcí první atomové bomby).

Prvním zařízením, které můžeme označit za „počítač“ v současném smyslu, byl automatický kalkulátor MARK 1, který byl postaven roku 1944 (ještě pomocí relé, tedy jako elektromechanické zařízení). Jeho autorem byl Američan Howard Aiken z Harvardovy univerzity. Aiken vycházel z myšlenek Charlese Babbage, které se své praktické realizace dočkaly teprve po 100 letech! O dva roky později se objevil první elektronický počítač ENIAC⁵ (vytvorili jej J. P. Eckert a J. W. Mauchly z Pensylvánské univerzity), který měl původně usnadňovat balistické výpočty.

Obecně se však za první počítač v plném smyslu tohoto slova považuje EDVAC⁶, který byl zkonstruován na Princetonské univerzitě. Odlišoval se tím, že prováděný program spolu s výsledky výpočtu byl kompletně umístěn v paměti počítače. S touto zásadní myšlenkou přišel matematik John von Neumann (Američan pocházející z Maďarska)⁷.

– poválečné období –

Vývoj počítačů probíhá souběžně v mnoha zemích. Firmy se začínají zajímat o vstup na nově vzniklý slibný trh počítačů (končí totiž éra, kdy se na univerzitách stavěly unikátní prototypy). Na trhu se objevují kalkulačky IBM 604 a BULL Gamma 3 a poté velké vědecké počítače, jako například UNIVAC 1 nebo IBM 650. Počátečním náznakům dominance některých producentů se snaží čelit výzkumy, které probíhají v mnoha zemích (s různou systematicností a s nestejnou politickou podporou), ale to už je téma na samostatnou knihu.

⁴ Narodil se ve městě Königsberg (česky Královec), které se v současnosti nazývá Kaliningrad.

⁵ Anglicky *Electronic Numerical Interpreter And Calculator*.

⁶ Anglicky *Electronic Discrete Variable Automatic Computer*.

⁷ Koncepce počítače pochází z roku 1946, ale poprvé byla prakticky realizována teprve v roce 1956.

– současnost –

Bouřlivý rozvoj elektroniky vede k masové a stále probíhající komputerizaci všech oblastí života. Počítače jsou nyní všudypřítomné a nedokážeme se bez nich obejít. Plní natolik různorodé úkoly, jaké si jen lidská mysl dokáže představit.

Nedávná historie aneb počátky metodologie programování

Vývoj popsaný v předchozí části kapitoly dospěl do fáze, kdy programátoři získali přístup k opravdovým počítačům. Velké investice vkládané do rozvoje hardwaru nepochyběně poskytly působivé výsledky, jak se můžeme v současnosti přesvědčit prakticky v každé kanceláři i ve stále větším počtu domácností.

V šedesátých letech se začaly budovat první skutečně velké informační systémy – poměřováno rozsahem kódu (hlavně v assembleru), který si tvorba dané aplikace vyžádala. Protože se však programování stále považovalo za činnost, která je založena zejména na intuici a citu, docházelo při tvorbě programů k docela vážným karambolům: systémy buď vznikaly rychle, ale nebyly příliš spolehlivé, nebo náklady na vývoj produktu značně převyšovaly původní rozpočet, což zpochybňovalo smysl celého projektu. Chyběly metody i nástroje, které by umožňovaly kontrolovat správnost programů. Vytvořené programy se obvykle testovaly tak dlouho, než byly kompletně odladěné⁸. Všimněme si, že oba zmíněné faktory (spolehlivost systémů i finanční náklady) mají v praxi mimořádný význam. Pokud bankovní informační systém nefunguje stoprocentně správně, neměla by jej banka vůbec nasadit! Na druhou stranu investice věnované na vývoj programů by neměly ohrozit finanční zdraví podniku.

V jisté fázi se situace natolik zhoršila, že se začalo otevřeně mluvit o krizi vývoje softwaru. Roku 1968 proběhla v německém městě Garmisch-Partenkirchen konference NATO, kde mohli odborníci nastalou situaci prodiskutovat. O rok později vznikla v rámci organizace IFIP (ang. *International Federation for Information Processing*) speciální pracovní skupina, která měla vyvinout metodologii programování.

Z historického hlediska se však diskuze na téma dokazování správnosti algoritmů začala odvijet od článku Johna McCarthyho „A basis for a mathematical theory of computation“ (Základy matematické teorie výpočtu), ve kterém se objevila věta: „Místo testování počítačových programů metodou pokusů a omyleů až do okamžiku jejich kompletního odladění bychom měli dokazovat, že tyto programy mají požadované vlastnosti.“ Jména lidí, kteří se zabývali teorií metodologie programování, zůstávají v odborném povědomí. Patřili k nim Dijkstra, Hoare, Floyd, Wirth a další (ještě na ně v této knize budeme nejednou odkazovat).

Z krátkého přehledu v předchozích dvou odstavcích vyplývá, že algoritmika je poměrně mladý vědní obor. Měli bychom si také uvědomit, že tento obor nevznikl „na zelené louce“. V současnosti ji sice musíme považovat za samostatnou sféru výzkumu, ale přitom bychom neměli přehlížet přínos generací matematiků, kteří algoritmice kromě nástrojů na popis problémů poskytli i mnoho užitečných teoretických koncepcí. (Podobně tomu bylo při rozvoji mnoha jiných oblastí lidského poznání.)

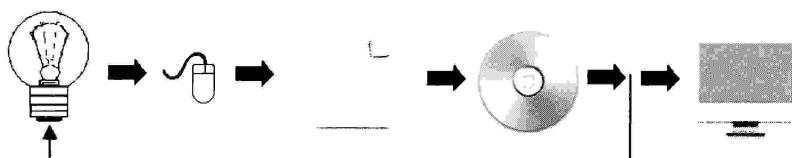
Když jsme již definovali hlavní téma této knihy – totiž algoritmy, podívejme se na několik způsobů, kterými je lze popsat.

8 Proces odstraňování chyb z programu.

Proces koncepce programů

Před historickou odbočkou jsme vyjmenovali několik typických vlastností, které by měl mít algoritmus, jak tento pojem chápeme v informatice. Zdůrazňovali jsme hlavně přesnost zápisu. Tento požadavek vychází z omezení, jaká kladou současné počítače a komplátory. Nedokáží totiž porozumět příkazům, které nejsou formulovány přesně a nejsou sestaveny podle pevných syntaktických pravidel.

Obrázek 1.1 zjednodušeným způsobem znázorňuje fáze procesu programování počítačů. Velká žárovka na začátku symbolizuje etapu, kterou programátoři občas vyneschávají (a obvykle toho později litují) – jedná se o REFLEXI.



Obrázek 1.1: Fáze tvorby programu

Následně vzniká *zdrojový kód* nového programu ve formě textového souboru, jehož obsah se do počítače zadává v běžném textovém editoru. Většina stávajících komplátorů má takový editor již integrovaný, takže programátor v praxi nemusí opouštět *integrované vývojové prostředí (IDE)*, které kombinuje všechny nástroje, jež jsou během programování potřeba. Některá integrovaná prostředí navíc obsahují pokročilé grafické editory, které umožňují sestavit uživatelské rozhraní programu prakticky bez napsání jediného řádku kódu. Když odhlédneme od takových detailů, obecně platí, že výsledkem programátorovy práce je soubor nebo skupina souborů, kde je v symbolické formě popsáno chování výsledného programu. Tento popis je zakódován v *programovacím jazyce*, který je zpravidla podmnožinou přirozeného jazyka⁹. Kompilátor provede jednodušší či složitější analýzu správnosti, a pokud je vše v pořádku, vyprodukuje *spustitelný kód*, který je zapsán v podobě srozumitelné počítači. Soubor se spustitelným kódem lze později spustit pod kontrolou operačního systému počítače (samotný systém mimochodem také sestává z mnoha jednotlivých programů). Spustitelný kód je specifický pro konkrétní typ operačního systému. V posledních letech se rozšířil jazyk Java, který umožňuje tvořit programy nezávislé na operačním systému. Přitom se však využívá určitého triku: spustitelný (bajtový) kód se nespouští přímo v operačním systému, ale ve speciálním běhovém prostředí. Toto prostředí jej izoluje od hardwaru i operačního systému a slouží jako mezivrstva, která po spuštění bajtového kódu dokončuje jeho překlad na výsledný kód.

Na které místo tohoto procesu patří tematika této knihy? Z celého komplikovaného procesu tvorby programů se zaměříme na fázi, která alespoň prozatím ještě není automatizovaná: návrhem algoritmů, jejich kvalitou a metodami programování, které se aktuálně v informatice používají. Upozorníme na konkrétní problémy, které lze řešit pomocí počítačů, a poté ukážeme, jak přitom postupovat efektivně. V knize se nebudeme zabývat tematikou tvorby vnějšího vzhledu programů, tedy tzv. *uživatelského rozhraní*.

Úrovně abstrakce popisu a výběr jazyka

Při popisu algoritmů se musíme rozhodnout, jak je prezentovat navenek. Za tímto účelem bychom mohli zvolit jeden z protichůdných přístupů:

⁹ V praxi se jedná o angličtinu.

- přiblížit se počítači (jazyk symbolických adres neboli assembler však není srozumitelný pro nezasvěceného čtenáře);
- přiblížit se člověku (slovní popis se nachází na nejvyšším stupni abstrakce, která u adresáta předpokládá takovou úroveň inteligence, jakou nelze současné stroje vybavit¹⁰⁾).

Pokud bychom se rozhodli prezentovat algoritmy v assembleru, museli bychom svůj výklad v zásadě spojit s konkrétním typem počítače, takže by naše úvahy postrádaly obecnou platnost a popisy algoritmů by se těžko analyzovaly. Na druhou stranu slovní popis vnáší riziko nejednoznačnosti, která by se nám nemusela vyplatit: program přeložený do podoby srozumitelné počítači by občas nefungoval.

Abychom se tomuto problému vyhnuli, obvykle algoritmy prezentujeme kompromisním způsobem:

- pomocí existujícího programovacího jazyka;
- užíváme programovacího pseudokódu (směsi přirozeného jazyka a syntaktických forem převzatých z několika reprezentativních programovacích jazyků).

V této příručce se setkáme s oběma formami a výběr jedné z nich se řídí kontextem popisované problematiky. Například: pokud lze příslušný algoritmus srozumitelně představit pomocí programovacího jazyka, dáváme přednost této možnosti. Občas se však dostaneme do situací, kdy je zbytečné uvádět kód v úplném tvaru, jaký lze zadat do počítače (například proto, že jsme podobným tématem již zabývali dříve), nebo by kód nebyl dostatečně čitelný (vzhledem k počtu řádků by se kód nevešel na jednu stránku). V každém případě by však střídání obou forem nemělo čtenářům činit větší potíže.

Už v úvodu jsme prozradili, že ukázkové programy budou napsány v jazyce C++. Je načase podrobněji vysvětlit důvody pro výběr tohoto jazyka. C++ patří mezi programovací jazyky označované jako *strukturální*. Obecně platí, že takové jazyky usnadňují psaní čitelného a srozumitelného kódu. Vzhledem ke své příbuznosti s klasickým jazykem C jazyk C++ samozřejmě umožňuje tvořit absolutně necitelné programy, ale něčemu takovému se budeme pečlivě vyhýbat. Abychom zachovali jednoduchost výpisů, často záměrně vynecháme možné optimalizační postupy. Hlavním důvodem použití jazyka C++ je však to, že dovoluje programovat na mnoha úrovních abstrakce. Díky třídám a všem objektovým vlastnostem tohoto jazyka lze zároveň velice snadno skrývat detaily implementace i rozšiřovat již definované moduly (bez jejich nákladného „přepisování“). Takovým výhodám je těžké odolat.

Měli bychom však zdůraznit, že vybraný programovací jazyk má jen pomocnou roli. Mnozí programátoři se vyžívají v diskuzích o tom, který jazyk je lepší. Tyto debaty jsou stejně neplodné jako hádky mezi fanoušky fotbalových klubů. Programovací jazyk je nakonec pouhým nástrojem, který se navíc během let značně proměňuje. Při práci na určitých částech této knihy jsem musel občas potlačovat silné pokušení vyjádřit některé algoritmy v jazycích jako LISP či PROLOG.

Tím by se sice zjednodušily všechny úvahy o seznamech a rekurzi, ale bohužel by se také okruh potenciálních čtenářů této knihy zúžil na osoby, které se profesionálně zabývají informatikou.

Uvědomuji si, že část čtenářů jazyk C++ nezná, a proto jsem pro ně v příloze A připravil rychlou kurz tohoto jazyka. Jsou zde souběžně uvedeny syntaktické struktury v jazycích C++ a Pascal, aby se čtenáři porovnáváním úryvků kódu naučili číst programové výpisy v této knize. Několikastránková příloha samozřejmě nenahradí příručku věnovanou výhradně jazyku C++, ale zpřístupní knihu i lidem, kteří se zajímají o problematiku algoritmů a plánují programovat v jiném jazyce.

10 Nemluvně snadno zvládá problémy, s nimiž zápasí specialisté na umělou inteligenci, když se je snaží řešit pomocí počítačů! (Jedná se o efektivitu učení, rozpoznávání tvarů atd.)

Správnost algoritmů

I když program zadáme do počítače, zkompilujeme a spustíme, není jisté, že později neselže (ať už to v praxi znamená cokoli). V případě domácích programátorských pokusů to nemá zásadní význam (doplátíme na to totiž jen my sami). Pokud bychom však program chtěli prodávat, situace se poněkud komplikuje. Kromě toho, že utrpí naše programátorská pověst, přichází v úvahu také naše odpovědnost za eventuální škody, které program svým uživatelům způsobil.

Chyb ve svých produktech se nedokáží vyvarovat ani velké programátorské koncerny – měsíc po reklamní kampani produktu X se potichu objevují „bezplatné“ aktualizované verze (pro legální uživatele) s odstraněnými chybami, které před vydáním produktu unikly pozornosti. Rozšířené operační systémy jako například Windows nebo Mac OS obsahují mechanizmy automatické aktualizace pomocí Internetu, které producentům systému umožňují opravovat chybné systémové funkce nebo reagovat na aktuální hrozby (typu nových virů).

Firmy totiž spěchají, aby předešly konkurenci, a jejich manažeři proto nařizují uvádět na trh nedokončené produkty. Trpí tím uživatelé, kteří se takovým praktikám nemohou nijak bránit. Na druhou stranu problém eliminace programových chyb vůbec není banální a představuje téma vážných odborných výzkumů¹¹.

Zaměřme se však nyní na situaci, která je běžnému programátorovi bližší: píše program a chce dostat odpověď na otázku: „Bude tento program správně fungovat v každé situaci a pro každou možnou konfiguraci vstupních dat?“ Odpověď je tím těžší, čím složitější jsou procedury, které hodláme analyzovat. Dokonce i v případě programů, které jsou zdánlivě krátké, může v praxi nastat tolik možných situací, že vylučují ruční testování. Zbývá tedy aplikovat důkazy matematické povahy, které zpravidla bývají dosti komplikované. Jednu z možných cest, kterými lze ověřit formální správnost algoritmů, představuje metoda *invariantů* (někdy označovaná jako *Floydova metoda*). Ve zkoumaném algoritmu můžeme snadno rozpoznat určitá klíčová místa, kde se děje něco, co je z hlediska algoritmu zajímavé. Obvykle není těžké taková místa najít: důležité jsou okamžiky inicializace proměnných, s nimiž bude procedura pracovat, testy ukončení algoritmu, hlavní cyklus atd. V každém z těchto bodů je možné stanovit jisté podmínky, které jsou vždy pravdivé – jedná se o tzv. *invarianty*. Poté lze důkaz formální správnosti algoritmu zjednodušit na kontrolu pravdivosti invariantů pro libovolná vstupní data.

V praxi se metoda obvykle používá dvěma způsoby:

- testování stavu kontrolních bodů pomocí *ladicího programu* (zjišťujeme hodnoty jistých důležitých proměnných a kontrolujeme, zda se pro určitá reprezentativní vstupní data chovají správně¹²),
- formální důkaz, že hodnota invariantů zůstane zachována pro libovolná vstupní data (např. pomocí matematické indukce).

Uvedené postupy mají jednu zásadní vadu: jsou pracné a snadno nás dokáží připravit o veškeré potěšení, jaké nám dává efektivní řešení problémů pomocí počítače. Měli bychom si však pamatovat, že programování má i tuto stránku. Čtenářům, kteří se zajímají o formální teorii programování, metody generování algoritmů a zkoumání jejich vlastností, lze doporučit jednu z přístupnějších

11 Správnost algoritmických systémů lze formálně zkoumat pomocí specializovaných jazyků.

12 Tvrzení jako „důležité proměnné“, „správné“ chování programu, „reprezentativní“ vstupní data atd. vesměs nejsou příliš přesná a úzce souvisejí s konkrétním programem, který analyzujeme.

(ale značně obsáhlých) knih o těchto tématech: [Gri87]. Nadšený úvod k ní napsal sám Dijkstra¹³, což je pro takovou práci asi to nejlepší doporučení. Jiný titul podobného typu, [Kal90], jistě oceň milovníci formálních důkazů a matematického myšlení. Metody matematického dokazování správnosti algoritmů jsou v těchto knihách představeny v jistém smyslu mimochodem. Autoři se snaží hlavně nabídnout nástroje, které by umožnily kvaziautomatické generování algoritmů.

Každý program vytvořený těmito metodami je automaticky správný – za předpokladu, že se během celého procesu nedopustíme nějaké chyby. Algoritmus lze vygenerovat teprve poté, co jej správně zapíšeme podle schématu:

{vstupní podmínky¹⁴} **hledaný program** {koncové podmínky}

Máme-li potřebné zkušenosti, dokážeme sestavit takovou posloupnost instrukcí, která zajistí přechod od „vstupních podmínek“ ke „koncovým podmínkám“. Správnost algoritmu pak již nemusíme formálně dokazovat. K problému můžeme také přistupovat z jiné strany. Pokud máme určitou sadu vstupních podmínek a konkrétní program, můžeme se ptát, zda budou po provedení programu pokaždé splněny požadované koncové podmínky.

Někteří čtenáři se možná pozastavují nad tím, že předchozí výklad je příliš obecný. Nutí nás však k tomu rozsah tématu, které by vyžadovalo prakticky samostatnou knihu! Nezbývá nám tedy než doporučit přečtení některé z výše zmíněných publikací, které však bohužel zatím nevyšly v českém překladu.

13 Když zmiňujeme tohoto autora, doporučujeme alespoň zběžné pročtení knihy [DF98], která představuje velmi dobrý úvod do metodologie programování.

14 Hodnoty proměnných, určité s nimi svázané logické podmínky atd.

KAPITOLA 2

Rekurze

V této kapitole se budeme zabývat jedním z nejdůležitějších mechanizmů, které se v informatice používají – jedná se o *rekurzi*. Rekurzi se sice můžeme vyhnout, ale nikomu, kdo tento styl programování alespoň jednou vyzkoušel, není potřeba její výhody vysvětlovat. Navzdory prvnímu dojmu není mechanizmus rekurze vůbec jednoduchý a řada jeho aspektů vyžaduje hlubokou analýzu. Rekurze však usnadňuje popis mnoha problémů (dovoluje například snadno definovat struktury založené na fraktálech) a u mnoha informatických úloh (například u „stromových“ struktur) se bez ní vzhledem k jejich charakteru neobejdeme.

Tato kapitola má klíčový význam pro zbývající část knihy, jejíž kapitoly lze sice procházet téměř v libovolném pořadí, ale bez dobrého porozumění samotné povaze rekurze nelze pochopit mnoho později prezentovaných algoritmů a programovacích metod.

V této kapitole:

- Definice rekurze
- Ukázka principu rekurze
- Jak pracují rekurzivní programy?
- Rizika rekurze
- Další nástrahy
- Typy rekurzivních programů
- Rekurzivní myšlení
- Praktické poznámky k rekurzivním technikám
- Úlohy
- Řešení a poznámky k úlohám

Definice rekurze

Rekurze se často klade do protikladu k iterativnímu přístupu, čili *n*-násobnému provádění algoritmu takovým způsobem, aby výsledky získané při předchozích iteracích (označují se také jako „průběhy“) mohly posloužit jako vstupní data následných iterací. Iterace jsou řízeny instrukcemi cyklu (např. `for` či `while`). Rekurze funguje podobně, ale místo cyklu je založena na tom, že stejná procedura (funkce) opakovaně volá sebe samu s jinými parametry. I v těle rekurzivní procedury samozřejmě můžeme najít klasický cyklus, který však plní jen pomocnou roli a nepředstavuje klasifikační kritérium algoritmu.

Stojí za zmínu, že programy zapsané v rekurzivním tvaru lze převést na klasickou iterativní podobu (někdy snáze, jindy obtížněji). Tomuto tématu věnujeme celou kapitolu 6. Zatím se však soustředíme na pochopení mechanizmu rekurze. Ukazuje se totiž, že to není tak snadné, jak se na první pohled zdá.

S rekurzivním přístupem se seznámíme na příkladu. Představme si malé dítě ve věku kolem tří let. Rodiče mu nařídí, aby do krabice posbíralo všechny dřevěné kostky, které předtím „neúmyslně“ rozsypalo na podlahu. Na kostkách není nic složitého, jsou to obyčejné dřevěné krychličky, jaké se skvěle hodí k budování primitivních staveb. Zadání je velmi jednoduché: „Všechno to posbírej a ulož zpátky do krabice.“ Takto vyjádřený úkol je pro dítě nesmírně náročný: kostek je spousta

KAPITOLA 2 Rekurze

a dítě neví, ze které strany se do toho pustit. Zatím sice není příliš šikovné, ale každopádně zvládne následující činnost: *vzít z podlahy jednu kostku a vložit ji do krabice*. Malé dítě se nenechává odradit složitosti problému, kterou si možná ani neuvědomuje, dává se do práce a rodiče potěšeně pozorují, jak se uklizená část podlahy každou minutou zvětšuje.

Zamysleme se na chvíli nad tím, jakou metodu dítě zvolilo: dítě ví, že rodiče po něm vůbec nechtějí, aby „sebralo všechny kostky“ (protože to se ve skutečnosti najednou udělat nedá), ale má „vzít jednu kostku, položit ji do krabice a poté sebrat do krabice ty zbývající“. Jakým způsobem může provést tu druhou část úkolu? Jednoduše – úplně stejně, jako tu první: „bereme jednu kostku...“ atd. a postupujeme tak až do chvíle, kdy na zemi nezůstala žádná kostka.

Podívejme se na obrázek 2.1, který symbolickým způsobem znázorňuje zvolený postup při řešení problému „úklidu rozsypaných kostek“.



Obrázek 2.1: „Úklid kostek“ aneb rekurze v praxi

Dítě si nejspíš nedokáže uvědomit, že postupuje rekursivním způsobem, i když jej opravdu zvolilo. Když se na výše popsaný problém podíváme pozorněji, můžeme si všimnout, že jeho řešení se vyznačuje následujícími vlastnostmi, které jsou pro rekursivní algoritmy typické:

- Je jasné určeno, kdy má algoritmus skončit („v okamžiku, kdy na podlaze nebudu žádné kostky, víš, že jsi úkol splnil“).
- „Velký“ a složitý problém byl rozložen na elementární problémy (které umíme řešit) a na problémy nižšího stupně náročnosti než ten, před nímž jsme stáli na začátku.

Všimněme si, že jsme pojmem „algoritmus“ použili poměrně volným způsobem. Má smysl rozebírat výše uvedenou úlohu v kategorických algoritmu? Můžeme vůbec tříletému dítěti připisovat znalosti, které si samo neuvědomuje?

Příklad, na kterém jsme předvedli princip rekursivního algoritmu, je nepochybně kontroverzní. Každý specialist na dětskou psychologii by se po přečtení předchozích úvah nejspíš zoufale chytil za hlavu. Proč jsme však zvolili právě takový, a nikoli jiný – možná více informatický – příklad? Snažili jsme hlavně dokázat, že rekursivní způsob myšlení dokonale odpovídá lidské povaze a k mnoha problémům, které svým rozumem řešíme, podvědomě přistupujeme právě rekursivním způsobem. Toto odvážené tvrzení můžeme dále rozvinout: Když se rozhodneme, že budeme na rekursivní algoritmy pohlížet intuitivně, přestanou nám připadat záhadné, i když si zpočátku možná nebudeš plně uvědomovat, na jakých mechanizmech jsou založeny.

Výše uvedené vysvětlení podstaty rekurze by mělo být značně srozumitelnější než typický přístup, který končí nepříliš jasným tvrzením, že rekursivní algoritmus je takový, který odkazuje sám na sebe.

Ukázka principu rekurze

Program, který budeme analyzovat v této části kapitoly, hodně připomíná problém kostek, s nímž jsme se setkali před chvílí. Využívá stejné rekursivní schéma, jen svým účelem se poněkud více blíží informatické praxi.

Naším úkolem je vyřešit následující problém:

- Dostaneme pole n celých čísel $tab[n] = tab[0], tab[1], \dots, tab[n - 1]$.
- Máme zjistit, zda pole tab obsahuje číslo x (zadané jako parametr).

Jak by postupovalo dítě z příkladu, který nám posloužil k vyjasnění pojmu rekurze (přitom budeeme pochopitelně předpokládat, že se dítě již seznámilo se základy informatiky)? Je velmi pravděpodobné, že by uvažovalo následujícím způsobem:

- vezmi první neprozkoumaný prvek n -prvkového pole;
- pokud se aktuálně analyzovaný prvek pole rovná x , pak:
oznam nalezení prvku a skonči;
v opačném případě:
prozkoumej zbylou část pole.

Uvedli jsme podmínu, za které může program ukončit svou práci s pozitivním výsledkem. V případě, kdy jsme prohlédli celé pole a prvek x jsme nenašli, musíme program samozřejmě ukončit nějakým dohodnutým způsobem – například zprávou o neúspěšném hledání.

Podívejme se nyní na jeden z možných příkladů, jak tento program realizovat:

```
rek1.cpp
#include <iostream>
using namespace std;
const int n=10;
int tab[n]={1, 2, 3, 2, -7, 44, 5, 1, 0, -3};

void hledej(int tab[], int left, int right, int x)
{
    // left, right = levá a pravá hranice prohledávané oblasti
    // tab           = pole
    // x             = hledaná hodnota
    if (left>right)
        cout << "Prvek " << x << " nebyl nalezen\n";
    else
        if (tab[left]==x)
            cout << "Byl nalezen hledaný prvek " << x << endl;
        else
            hledej(tab,left+1,right,x);
}

int main()
{
    hledej(tab,0,n-1,7);
    hledej(tab,0,n-1,5);
}
```

Podmínkou ukončení programu je buď nalezení hledaného prvku x , nebo také opuštění prohledávané oblasti. Přes svou jednoduchost tento program dobře ilustruje základní vlastnosti typického rekurzivního programu, které jsme již uvedli. Ostatně podívejme se pozorně:

- Je jasné určeno, kdy program končí.
 - Je nalezen hledaný prvek.
 - Dojde k překročení rozsahu pole.
- Velký problém jsme rozložili na elementární problémy, které umíme řešit (viz výše), a na analogický problém, pouze s nižším stupněm složitosti (z pole velikosti n přecházíme na pole velikosti $n-1$).

Na základě uvedených postřehů se pokusme určit, jakých dvou základních chyb se programátoři dopouštějí při tvorbě rekurzivních programů:

- Špatné vymezení koncové podmínky programu
- Nevhodná (neefektivní) dekompozice problému

V další části kapitoly se pokusíme společně dojít k určitým „bezpečnostním zásadám“, které jsou při psaní rekurzivních programů nezbytné. Mezitím však musíme pečlivě rozebrat, jak tyto programy fungují.

Jak pracují rekurzivní programy?

Zvídavý čtenář může na tomto místě prohlásit: „Dobре, na příkladu jsem viděl, že TO funguje, ale nyní bych se chtěl podrobněji dozvědět, JAK to funguje!“ Nezbývá tedy než tomuto oprávněnému požadavku vyhovět.

Odpověď poskytuje právě tato část kapitoly. Použitý příklad je sice možná poněkud banální, avšak umožňuje dokonale ilustrovat způsob, jakým se rekurzivní program provádí.

Už na střední škole (nebo snad dokonce na základní?) se v hodinách matematiky často objevuje funkce označovaná jako „faktoriál“ – součin všech přirozených čísel od 1 do n včetně. Tento užitečný symbol se definuje následujícím způsobem:

$$\begin{aligned} 0! &= 1, \\ n! &= n \cdot (n-1)!, \text{ kde } n \in N, n \geq 1 \end{aligned}$$

Když budeme za n dosazovat přípustné hodnoty (tedy přirozená čísla větší než 1), dokážeme snadno vypočítat počáteční hodnoty faktoriálu. Pro větší čísla n už výpočet není tak prostý, ale k čemu máme počítače – nic nám přece nebrání napsat jednoduchý program, který bude počítat faktoriál rekurzivním způsobem:

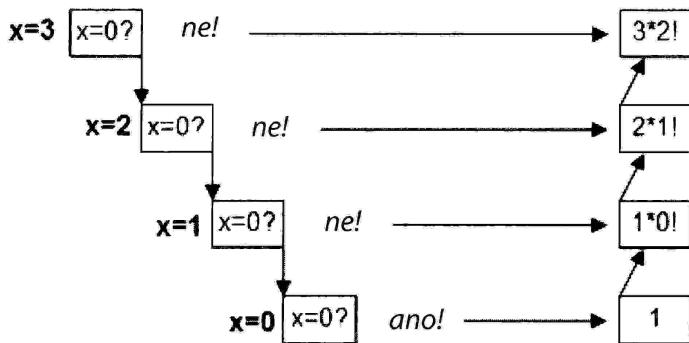
```
rek2.cpp
unsigned long int faktorial(int x)
{
    if (x==0)
        return 1;
    else
        return x*faktorial(x-1);
}

int main()
{
    cout << "faktorial(5)=" << faktorial(5) << endl;
```

Sledujme příklad, kdy program počítá hodnotu $3!$. Obrázek 2.2 znázorňuje následné fáze volání rekurzivní procedury a kontrolu podmínky pro elementární případ.

Na obrázku se používají tyto konvence:

- Šipky směřující dolů znamenají, že program z úrovně n přechází na nižší úroveň $n-1$ atd., aby se dostal k elementárnímu případu $0!$.
- Vodorovná šipka označuje výpočet částečných výsledků.
- Šikmá šipka symbolizuje proces předávání částečného výsledku z nižší úrovně na vyšší.



Obrázek 2.2: Strom volání funkce faktorial(3)

Co však znamenají ty tajemné úrovně, předávání parametrů a další pojmy? Prozatím nám mohou znít poněkud exoticky. Popišme tedy poněkud podrobněji způsob výpočtu funkce faktorial(1), abychom přiblížili, jak program funguje:

- Funkce faktorial přijímá jako parametr svého volání číslo 1 a zjišťuje: „Rovná se číslo 1 číslu 0?“ Odpověď zní: „Nikoli.“ Funkce tedy předpokládá, že jejím výsledkem je výraz $1 \cdot \text{faktorial}(0)$, čili jednoduše faktorial(0).
- Hodnotu výrazu faktorial(0) bohužel nezná. Funkce proto vyvolá svou další instanci, která bude počítat hodnotu výrazu faktorial(0). Zároveň pozastaví vyhodnocování výrazu $1 \cdot \text{faktorial}(0)$.
- Vyvolání funkce faktorial(0) vrací sice částečný, ale už konkrétní číselný výsledek (1), pomocí kterého lze určit hodnotu výrazu $1 \cdot \text{faktorial}(0)$ neboli faktorial(1).

Technicky je předávání parametrů zajištěno pomocí **zá sobníku**, což je speciální místo v operační paměti, které umožnuje ukládat informace potřebné během činnosti programů a při dynamickém přidělování paměti. Programátor se však o tyto nízkoúrovňové procesy nemusí vůbec zajímat. Parametr je sice vrácen pomocí zásobníku, ale mohl by být předán třeba telefonicky. Výsledek, který se projevuje zprávou „Výpočet je dokončen!“, je totiž v každém případě úplně stejný a na způsobu realizace nezávisí.

Kde ale najdeme zmíněné úrovně rekurze? Podívejme se ještě jednou na obrázek 2.2. Vlevo od rámečku, který znázorňuje funkci faktorial, je uvedena aktuální hodnota testovaného parametru x . Vzhledem k tomu, že některé instance funkce faktorial vyvolávají svou další kopii (kvůli výpočtu částečného výsledku), je potřeba jednotlivé instance nějak rozlišit. Nejjednodušší je přitom použít hodnotu proměnné x , která udává aktuální hloubku rekurze.

Rizika rekurze

Použití rekurze občas přináší jisté nevýhody. Dvě klasická nebezpečí představíme v následujících příkladech.

Fibonacciho posloupnost

Naším prvním úkolem je napsat program, který bude počítat prvky *Fibonacciho posloupnosti*. Tato posloupnost se často objevuje v přírodě a lze ji definovat takto:

$$\begin{aligned} fib(0) &= 0, \\ fib(1) &= 1, \\ fib(n) &= fib(n - 1) + fib(n - 2), \text{ kde } n \geq 2 \end{aligned}$$

Prvky této posloupnosti jsou přirozená čísla, která se vyznačují tím, že každý následující prvek (s výjimkou dvou počátečních) představuje součet dvou předchozích (tj. 1, 1, 2, 3, 5, 8, 13, ...). Název pochází od jména Leonarda z Pisy zvaného Fibonacci, který tuto posloupnost publikoval jako první¹.

Vztah k přírodním dějům není náhodný. Představme si, že chováme králíky, jejichž populace neustále roste podle těchto pravidel:

- Na začátku máme jeden pár.
- Každá samice králíka měsíc po kopulaci porodí potomstvo – jednoho samce a jednu samici.
- Měsíc po narození králík dospívá a může se začít rozmnožovat.

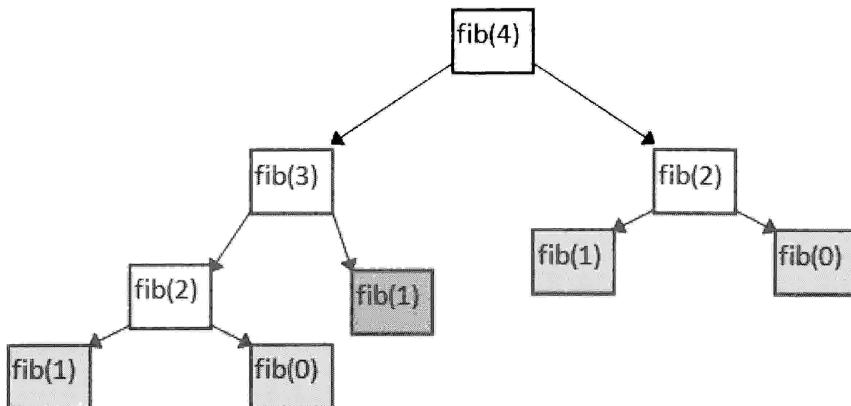
Jak bude za těchto podmínek vzrůstat počet našich zvířat? Na konci prvního měsíce můžeme očekávat první oplodnění u prvního páru králíků. Koncem druhého měsíce samice porodí pár mláďat a na farmě budou žít dva páry. Ve třetím měsíci už budeme mít tři páry, protože první samice porodí další potomky. Zároveň se začnou rozmnožovat dva králíci, kteří se narodili před měsícem... Jednoduše dokážeme spočítat, že počet králíků v jednotlivých měsících bude roven 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... A tato čísla představují prvky Fibonacciho posloupnosti².

Následující program asi nikoho nepřekvapí, protože se jedná o téměř přesný překlad uvedeného postupu do programovacího jazyka:

```
rek3.cpp
unsigned long int fib(int x)
{
    if (x<2)
        return x;
    else
        return fib(x-1)+fib(x-2);
}
```

Zkusme podrobně sledovat jednotlivá rekurzivní volání. Jednoduchým rozbořem získáme následující strom (obrázek 2.3):

1 Už méně se ví o tom, že právě Fibonacci přinesl z Asie do Evropy znalosti moderních aritmetických koncepcí, mimo jiné záporných čísel a pozičního systému zápisu číslic.
 2 Kvůli přesnosti dodejme, že existuje iterativní verze výpočtu, která využívá existenci čísla φ (čti *f*). Toto číslo, označované jako zlatý řez, získáme, když libovolný prvek posloupnosti vydělíme prvkem předchozím (hodnota φ tedy činí přibližně 1,61804).



Obrázek 2.3: Výpočet čtvrtého prvku Fibonacciho posloupnosti

Stínované rámečky označují elementární případy. Problém s rozměrem $n \geq 2$ je rozdělen na dva problémy jednoduššího typu: $n-1$ a $n-2$. U elementárních případů se proces dekompozice zastavuje.

Proč má tedy tato část kapitoly tak pesimistický název? Podívejme se na obrázek 2.3 pozorněji. Na první pohled si můžeme všimnout, že značná část výpočtů se provádí vícekrát (opakuje se například celá větev, která začíná výrazem fib(2)!). Funkce fib to nedokáže nijak zjistit. Nakonec je to jen program, který provádí to, co po něm chceme. V kapitole 9 popíšeme zajímavou programovací techniku (tzv. *dynamické programování*), díky níž se můžeme takovým problémům vyhnout.

Stack overflow!

Nadpis této části kapitoly česky znamená „přetečení zásobníku“. V praxi se ukazuje, že psaní programů neprobíhá podle našich záměrů, ale jedná se spíše o činnost, která patří do sféry magie. Po spuštění programu počítač často „zatuhne“ (program přitom nereaguje na žádné vstupy z klávesnice a musíme jej natvrdo ukončit³). Stává se to i těm nejpozornějším programátorům a neoddělitelně to patří k programátorskému řemeslu.

Programy typicky přestávají reagovat z několik příčin:

- narušení stability operačního systému kvůli nekorektnímu použití jeho prostředků;
- „nekonečné“ cykly;
- nedostatek paměti;
- nesprávná nebo nejasná definice podmínek, za jakých má program ukončit svou činnost;
- programátorská chyba (např. příliš pomalý algoritmus).

Rekurzivní programy obvykle vyžadují hodně paměti: při každém rekurzivním volání je nutné uchovat určité informace⁴, aby bylo možné obnovit stav programu před příslušným voláním. Tyto informace vždy zaberou část cenné paměťové kapacity. Můžeme se setkat s rekurzivními programy, u kterých lze maximální úroveň hloubky rekurze během jejich činnosti určit docela snadno. Při analýze programu, který počítá hodnotu $3!$, ihned vidíme, že sám sebe vyvolá jen třikrát. V případě funkce fib nám však zběžná prohlídka programu stejně jednoznačný závěr již neposkytne.

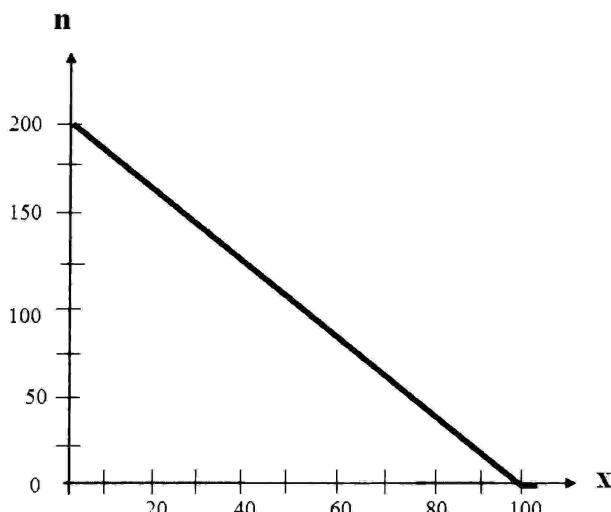
Často není snadné dospět ani k přibližnému odhadu. Asi nejlépe to dokládá MacCarthyova funkce (viz dále):

³ *Ctrl+Alt+Del* v systému DOS nebo Windows, instrukce *kill* v systému Unix atd.

⁴ Nebudeme zabíhat do podrobností, protože toto téma pro nás momentálně není příliš důležité.

```
rek4.cpp
unsigned long int MacCarthy(int x)
{
    if (x>100)
        return (x-10);
    else
        return MacCarthy(MacCarthy(x+11));
}
```

Už na první pohled vidíme, že funkce je jaksi „divná“. Kdo dokáže orientačně popsat, jak závisí počet jejich volání na parametru x uvedeném v prvním volání? Málodko by asi dokázal okamžitě určit, že tato závislost má tvar, který je znázorněn na obrázku 2.4.



Obrázek 2.4: Závislost počtu volání MacCarthyovy funkce na hodnotě parametru volání

Snazší je postřehnout, že pro všechna x větší než 100 se funkce provede pouze jednou. Ukázkový program umožnuje zadávat vstupní parametr a zobrazuje výslednou hodnotu MacCarthyovy funkce spolu s počtem jejích volání.

Jistě se shodneme, že výsledky této funkce vůbec nejsou tak předvídatelné, jak bychom mohli očekávat.

Cvičení 2.1

Prozkoumejte MacCarthyovu funkci na větším číselném intervalu než na obrázku. Jaká rizika lze přitom najít?

Cvičení 2.2

Nakreslete strom rekurzivních volání MacCarthyovy funkce pro hraniční parametr $x = 100$ i pro jeho hodnotu $x = 99$. Poté zkompilujte a spusťte ukázkový program, abyste předpokládané výsledky ověřili.

Další nástrahy

Jako by rekurzivní programy bez tak neměly dost negativních stránek, musíme ještě doplnit ty, které nevyplývají ze samotné povahy rekurze, ale spíše z programátorských chyb. Na tomto místě bychom možná měli zdůraznit, že rozbor „stinných stránek“ rekurze by nás neměl odradit od jejího využívání. Snažíme se spíše ukázat typická rizika a způsoby, jak se jim vyhnout. Takové preventivní metody máme k dispozici vždy (musíme však vědět, ČEMU se chceme vyhnout). Pusťme se tedy do čtení následujících odstavců.

Cesta na věčnost

V mnoha na první pohled správně sestavených rekurzivních funkcích se může snadno ukrývat chyba, která spočívá v tom, že aktivují nekonečný počet rekurzivních volání. Takový záladný příklad představíme v následujícím výpisu (`std.cpp`).

Když se program `std.cpp` pokusíme spustit s hodnotou `m=2`, v závislosti na operačním systému se objeví hlášení typu *Stack overflow* (přetečení zásobníku) či *Segmentation fault* (chyba segmentace). Stručně řečeno: došla volná paměť a stačilo k tomu jen několik sekund činnosti programu!

```
std.cpp
int CestaNaVecnost(int n)
{
    if (n==1)
        return 1;
    else
        if ( (n %2) == 0 ) // n sudé
            return CestaNaVecnost(n-2)*n;
        else
            return CestaNaVecnost(n-1)*n;
}
```

Co je příčinou potíží? Při pohledu na zdrojový kód programu těžko najdeme nějaké problematické místo. Zdá se, že rekurze je definována správně: máme elementární případ, kterým řetězec volání končí, a problém velikosti `n` se zjednoduší na problém s rozdílem `n-1` nebo `n-2`. Do pasti jsme se ovšem chytili právě kvůli svému naivnímu předpokladu, že proces zjednodušování vede k elementárnímu případu (čili k `n=1`). Důkladnější analýzou však můžeme zjistit, že pro parametr `n≥2` končí všechna rekurzivní volání *sudou* hodnotou `n`. Z toho vyplývá, že se nakonec dostaneme k případu s parametrem `n=2`, který je zredukován na případ typu `n=0`, který je zredukován na případ typu `n=-2`, který... Tak můžeme pokračovat až do nekonečna a nikdy nenarazíme na žádný elementární případ!

Závěr se sám nabízí: je potřeba pozorně kontrolovat, zda rekurze někdy skončí pro všechny hodnoty vstupních parametrů, které patří do definičního oboru funkce.

Správná definice nestačí

Na předchozím příkladu jsme ukázali problémy, které souvisejí s konvergencí rekurzivního procesu. Mohli bychom se domnívat, že když nám matematika poskytne správnou rekurzivní definici, můžeme si být jisti, že odpovídající rekurzivní program bude také správný (tzn. nezacyklí se, bude poskytovat očekávané výsledky atd.). Tato důvěra je však doslova naivní a nepodložená. Matematik má totiž veškeré možnosti, jaké mu poskytuje jeho obor: dokáže popsat definiční obor funkce,

KAPITOLA 2 Rekurze

dokázat, že je konečná, a dokonce vyhodnotit její výpočetní složitost. Neumí však zohlednit jeden faktor: jak danou funkci provede konkrétní kompilátor. Většina kompilátorů sice funguje podobně, ale vyskytuje se u nich drobné rozdíly, kvůli nimž mohou identické programy dávat odlišné výsledky. Následující příklad se týká právě takového případu.

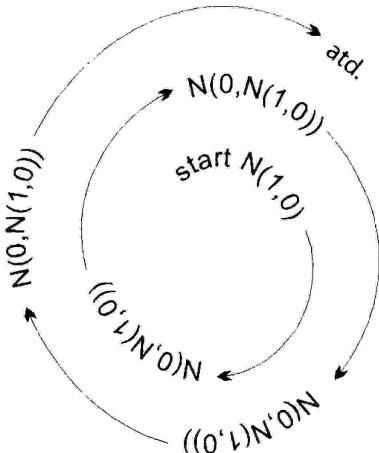
Podívejme se na tuto funkci:

```
int N(int n, int p)
{
    if (n==0)
        return 1;
    else
        return N(n-1,N(n-p,p));
}
```

Můžeme provést matematický důkaz⁵, že uvedená definice je správná v tom smyslu, že pro libovolné hodnoty $n \geq 0$ a $p \geq 0$ poskytuje konkrétní výsledek rovný 1. Tento důkaz vychází z předpokladu, že hodnota argumentu volání funkce se počítá pouze tehdy, když je skutečně potřebná (což vypadá docela logicky). Jak se ale bude chovat typický kompilátor jazyka C++?

Platí pravidlo, že všechny parametry rekursivní funkce se počítají jako první a poté je volána samotná funkce. (Tento způsob provedení se označuje jako *volání hodnotou*.)

Problém se pak může projevit tehdy, když se do volání funkce pokusíme dosadit ji samotnou. Vyzkoušejme, co se stane v případě naší funkce, např. pro $N(1, 0)$ (viz obrázek 2.5).



Obrázek 2.5: Nekonečná řada rekurzivních volání

Program se zacykluje kvůli výpočtu parametru p , i když druhé volání není k dokončení funkce vůbec potřeba! Funkce totiž obsahuje podmínu, která se týká elementárního případu: pokud $n=0$, vrátí hodnotu 1. Kompilátor o tom bohužel neví a snaží se druhý parametr vyčíslit, což vede k zacyklení programu.

Uvedený příklad můžeme považovat za jistou kuriozitu, nicméně stojí za zapamatování, že problém tohoto typu může nastat.

⁵ Viz [Kro89].

Typy rekurzivních programů

Po přečtení předchozích odstavců bychom mohli odvodit několik obecných závěrů ohledně programů, které využívají rekurzivních metod: bývají náročné na paměť, někdy přestávají reagovat... Takový pohled však naštěstí není objektivní. Rekurzivní programy mají jednu zásadní výhodu: jsou snadno srozumitelné a vyznačují se menším počtem řádků kódu. Díky druhé uvedené vlastnosti se v těchto programech snáze hledají případné chyby. Vraťme se však k našemu tématu.

Všimli jsme si, že rekurzivní programy mohou spotřebovat hodně paměti a pracovat dosti pomalu. Položme si otázku: Existují nějaké programovací techniky, pomocí nichž bychom mohli uvedené vady rekurzivních programů odstranit (nebo alespoň omezit)? Tuto otázku lze naštěstí zodpovědět kladně. Jistou třídu problémů rekurzivní povahy můžeme totiž formulovat dvěma způsoby, které poskytují přesně stejný konečný výsledek, ale poněkud se liší svou praktickou realizací. Rekurzivní metody si pro zjednodušení rozdělme na dva základní typy:

- „přirozená“ rekurze,
- rekurze „s pomocným parametrem“⁶.

S prvním typem jsme se setkali při analýze dosavadních příkladů a nyní se podívejme na ten druhý. Uvažujme ještě jednou příklad funkce na výpočet faktoriálu. Zatím jsme ji znali v této podobě:

```
rek5.cpp
unsigned long int faktorial1(unsigned long int x)
{
    if (x==0)
        return 1;
    else
        return x*faktorial1(x-1);
}
```

Funkci na výpočet faktoriálu lze však konstruovat i jinak. Dokládá to její následující verze:

```
unsigned long int faktorial2(unsigned long int x, unsigned long int tmp=1)
{
    if (x==0)
        return tmp;
    else
        return faktorial2(x-1, x*tmp);
}
```

Princip druhé funkce nemusí být zřejmý na první pohled. Stačí však vzít papír a tužku a na několika případech ověřit, že svou úlohu plní stejně dobře jako předchozí verze. Těm, kdo nejsou záběhlí v jazyce C++, je asi potřeba strukturu funkce faktorial2 vysvětlit. Libovolná funkce v C++ může zahrnovat výchozí parametry.

Díky tomu lze funkci s hlavičkou řekněme:

```
int FunDom(int a, int k=1)
```

vyvolat dvěma způsoby:

⁶ Dočasně se spokojme s tímto nepřesným názvem. Tímto typem rekurze se budeme znova zabývat v kapitole 6, avšak v jiném kontextu.

KAPITOLA 2 Rekurze

- S uvedením hodnoty druhého parametru, např. `FunDom(12, 5)`. V tom případě parametr `k` nabývá hodnoty 5.
- Bez zadání hodnoty druhého parametru, např. `FunDom(12)`. Parametru `k` je pak přiřazena výchozí hodnota, která se rovná hodnotě uvedené v hlavičce funkce, tedy 1.

Právě tuto užitečnou vlastnost jazyka C++ využívá druhá verze funkce na výpočet faktoriálu. Z jakých zásadních důvodů bychom měli používat tuto programovací metodu, která působí poněkud podivně? Nelze tady argumentovat lepší čitelností programu, protože funkce `faktorial2` je už na první pohled mnohem méně srozumitelná než funkce `faktorial1`.

Podstatná výhoda rekurze „s pomocným parametrem“ spočívá ve způsobu provádění programu. Představme si, že rekurzivní program „bez pomocného parametru“ při počítání výsledku vyvolá sám sebe desetkrát. To znamená, že částečný výsledek z desáté a nejhlubší úrovni rekurse bude nutné předat přes následných deset úrovní nahoru až do první instance funkce.

Každá „zmrazená“ úroveň, která čeká na předání částečného výsledku, vyžaduje určitou část paměti, aby bylo možné obnovit mj. hodnoty proměnných této úrovni (tzv. kontext). Navíc obnova kontextu sama o sobě zabírá cenný procesorový čas, který by bylo možné využít třeba k jiným výpočtům.

Nejspíše už si domýslíte, že rekurzivní program „s pomocným parametrem“ přitom postupuje poněkud efektivněji. Pomocný parametr slouží k předání prvků konečného výsledku. V programu, který disponuje tímto parametrem, tedy není nutné předávat výsledek výpočtu nahoru po jednotlivých úrovních. Ve chvíli, kdy program zjistí, že výpočet skončil, volající procedura dostane tuto informaci přímo z poslední aktivní úrovni rekurse. Díky tomu vůbec není potřeba zachovávat kontext jednotlivých mezilehlých úrovní. Záleží pouze na poslední aktivní úrovni, která dodá požadovaný výsledek.

Rekurzivní myšlení

Ačkoli jsme již ukázali, že rekurze je pro člověka zcela přirozená, někteří programátoři mají s jejím používáním jisté problémy. Potíže se zvládnutím podstaty této programovací techniky mohou vyplývat z nedostatku dobrých a názorných příkladů na její aplikaci. Na základě této úvahy předkládáme několik jednoduchých rekurzivních programů, které generují známé grafické motivy. Jestliže dobře pochopíme, jak tyto programy fungují, máme dostatečný důkaz o tom, že umíme přemýšlet rekurzivně (nikomu však rozhodně neuškodí, když si vyzkouší řešení úloh na konci kapitoly).

Příklad 1: Spirála

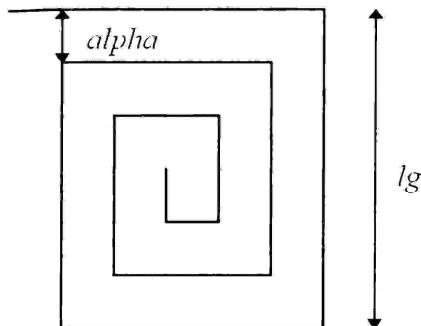
Zamysleme se, jak lze rekurzivně nakreslit spirálu jedním tahem – viz obrázek 2.6.

Program má tyto parametry:

- rozestup mezi rovnoběžnými čarami: `alpha`;
- délka strany kreslené jako první: `lg`.

Iterativní algoritmus by také nebyl složitý (stačil by běžný cyklus), ale řekněme, že jsme na něj na chvíli zapomněli a stejný úkol chceme zvládnout rekurzivně. Při rekurzi je podstatné hlavně najít vhodnou dekompozici problému. U této úlohy je znázorněna na obrázku, což by nám její případný převod do programu v jazyce C++ mělo hodně usnadnit.

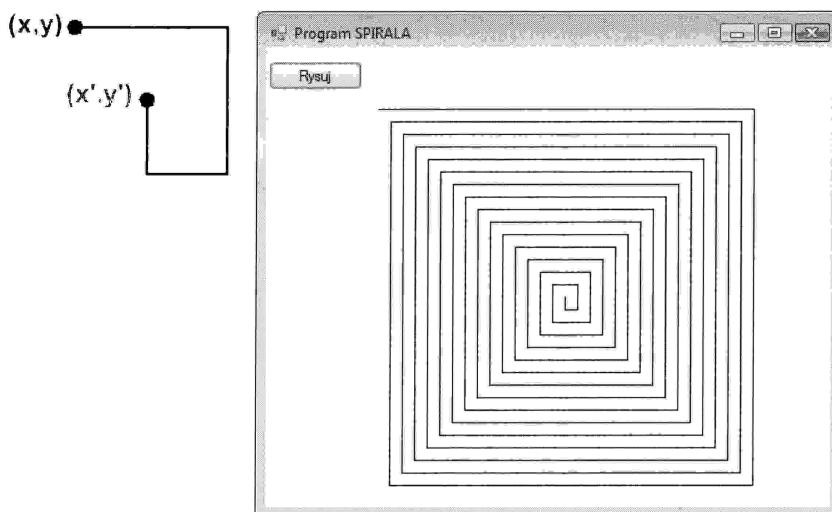
Rekurzivita naší úlohy je patrná na první pohled, protože program bude v zásadě opakovat stejné činnosti (kreslit rovnoběžné a svislé čáry, i když různě dlouhé). Nyní potřebujeme najít rekurzivní schéma a podmínky zakončení rekurzivních volání.



Obrázek 2.6: Spirála vykreslená pomocí rekurze

Jak lze toto zadání vyřešit? Nejdříve se poněkud přiblížíme k „realitě obrazovky“ a vybereme počáteční bod pomocí určité dvojice souřadnic (x, y) . Princip řešení spočívá ve kreslení čtyř vnějších úseků spirály tak, abychom se dostali do bodu (x', y') . V tomto novém počátečním bodě již můžeme vyvolat rekurzivní proceduru kreslení. Ta bude samozřejmě omezena konkrétními podmínkami, které zaručí, že korektně dokončí svou činnost.

Obrázek 2.7 představuje elementární případ řešení.



Obrázek 2.7: Rekurzivně vykreslená spirála – náčrt řešení + výsledek

Uveděme jednu z možných verzí programu, který funguje tak, jak jsme popsali výše. Program používá následující smluvené grafické instrukce⁷:

- `lineto(x, y)` – vykreslí úsečku od aktuální pozice do bodu (x, y)
- `moveto(x, y)` – přesune grafický kurzor do bodu (x, y)
- `getmaxx()` – vrací maximální vodorovnou souřadnici (závisí na rozlišení grafického režimu)
- `getmaxy()` – vrací maximální svislou souřadnici (viz výše)
- `getx()` – vrací aktuální vodorovnou souřadnici
- `gety()` – vrací aktuální svislou souřadnici

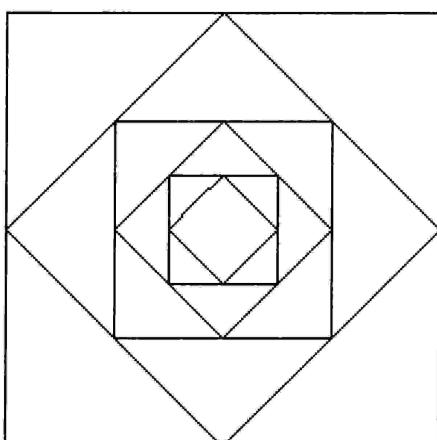
⁷ Jsou kompatibilní s knihovnami kompilátoru Borland pro systém DOS (soubory se nacházejí ve složce INNE v archivu ZIP se soubory ke stažení).

```
spiral.cpp
const double alpha=10;
void spiral(double lg,double x,double y)
{
    if (lg>0)
    {
        lineto(x+lg,y);
        lineto(x+lg,y+lg);
        lineto(x+alpha,y+lg);
        lineto(x+alpha,y+alpha);
        spiral(lg-2*alpha,x+alpha,y+alpha);
    }
}
int main()
{
    // inicializace grafického režimu
    moveto(90,50);
    spiral(getmaxx()/2,getx(),gety());
    getch(); // čekání na stisknutí klávesy
              // ukončení grafického režimu
    closegraph();
}
```

Poznámka: Grafický program ve verzi pro systém Microsoft Windows byl připraven v kompilátoru Microsoft Visual C++ Express Edition. Kompletní projekt určený pro grafické prostředí Windows je umístěn ve složce VISUAL\spiral. Vzhledem k tomu, že kód řešení pro systém Windows není příliš čitelný (obsahuje mnoho příkazů, které jsou specifické pro programování v tomto systému), rozhodli jsme se v knize neuvádět jeho plný výpis a ponechat originální stručný kód z předchozích vydání. Kódy pro Windows se nacházejí v souborech **spiral_win.cpp** a **spiral_win.h**.

Příklad 2: „Sudé“ čtverce

Zadání připomíná předchozí úlohu: Jak jedním tahem nakreslit obrazec, který vidíme na obrázku 2.8?



Obrázek 2.8: Rekurzivně vykreslené čtverce ($n = 3$)

Elementárním případem zde bude vykreslení jedné dvojice čtverců (vnitřní vzhledem k vnějšímu otočený o 45°). Tato úloha je ještě snazší než předchozí. Trik spočívá jen ve výběru vhodného místa rekurzivního volání⁸:

```
kwadraty.cpp
void ctverce(int n, double lg, double x, double y)
{
    // n - sudý počet čtverců
    // x, y - souřadnice počátečního bodu
    if (n>0)
    {
        lineto(x+lg,y);
        lineto(x+lg,y+lg);
        lineto(x,y+lg);
        lineto(x,y+lg/2);
        lineto(x+lg/2,y+lg);
        lineto(x+lg/2,y+lg/2);
        lineto(x+lg/2,y);
        lineto(x+lg/4,y+lg/4);
        ctverce(n-1,lg/2,x+lg/4,y+lg/4);
        lineto(x,y+lg/2);
        lineto(x,y);
    }
}

int main()
{
    // inicializace grafického režimu
    moveto(90,50);
    ctverce(5, getmaxx()/2, getx(), gety());
    getch();
    // ukončení grafického režimu
}
```



Poznámka: Grafický program ve verzi pro systém Microsoft Windows byl připraven v kompilátoru Microsoft Visual C++ Express Edition. Projekt najdeme ve složce VISUAL\kwadraty. Podobně jako v předchozím příkladu také zde vypouštíme plný výpis a ponecháváme originální stručný kód z předchozích vydání. Kódy pro Windows se nacházejí v souborech **kwadraty_win.cpp** a **kwadraty_win.h**.

Praktické poznámky k rekurzivním technikám

Při rozboru rekurzivních technik jsme zjistili, že mají své výhody i nevýhody. Zásadní výhodou je čitelnost a přirozenost zápisu algoritmů v rekurzivní formě, zejména tehdy, když je v rekurzivní podobě vyjádřen jak problém, tak i související datové struktury. Rekurzivní procedury obvykle bývají jasné a krátké a díky tomu v nich můžeme snadno najít eventuální chyby. Jak lze ale rekurzivní algoritmus zhodnotit? Aniž bychom používali komplikovaný matematický aparát, můžeme

⁸ Tento kód zůstal zachován z historických důvodů. Odpovídá kompilátoru Borland pro systém DOS a nachází se ve složce INNE archivu ZIP ke stažení.

KAPITOLA 2 Rekurze

jakost rekurzivního programu poměrně bezpečně posoudit podle toho, zda má správnou podmínu ukončení (elementární případ) a zda rekurzivní dekompozice vede ke *zmenšení* velikosti problému. Všechny bizarní konstrukce (viz MacCarthyovu funkci) obvykle patří spíše mezi akademické kuriozity než k příkladům správných procedur.

Velkou vadou mnoha rekurzivních algoritmů je jejich paměťová náročnost: mnohonásobná rekurzivní volání mohou snadno zabrat celou dostupnou paměť! Problém zde nepředstavuje ani tak samotné obsazení paměti, ale to, že rekurzivní algoritmy typicky neumožňují svou spotřebu snadno odhadnout. Můžeme k tomu sice využít metody, které slouží k analýze efektivity algoritmů (viz kapitolu 3), avšak příslušné výpočty bývají dosti pracné a občas je vůbec nelze provést.

V podkapitole „Typy rekurzivních programů“ jste se dozvěděli, jak se můžeme vyhnout potížím s pamětí díky rekurzi „s pomocným parametrem“. Některé problémy se však tímto způsobem řešit nedají a kromě toho tato metoda poněkud zhoršuje čitelnost programů. Nedá se nic dělat – není přece růže bez trnů...

Kdy bychom rekurzi neměli používat? Konečné rozhodnutí je vždy na programátora, avšak v některých situacích můžeme své dilema vyřešit poměrně snadno. Rekurzivní řešení bychom neměli volit v těchto případech:

- Rekurzivní algoritmus lze nahradit čitelným nebo rychlým iterativním protějškem.
- Rekurzivní algoritmus je *nestabilní* (pro jisté hodnoty vstupních dat se například může začyklit nebo poskytovat podivné výsledky – často je to způsobeno specifiky komplilátoru nebo vlastnostmi hardwarové platformy).

Poslední poznámku uvádíme spíše pro úplnost. V literatuře se občas můžeme setkat s úvahami na téma nevýhodných vlastností tzv. *křížové rekurze*: podprogram *A* volá podprogram *B*, který následně volá podprogram *A*. Příklad takové podivnosti záměrně neuvádíme, protože přemíra špatných příkladů může být na škodu. Na základě analýzy rekurzivních programů plných neobvyklých konstrukcí můžeme dojít k jednomu praktickému závěru: VYHNĚME SE JIM, pokud si správností programu nejsme zcela jisti a intuice nám napovídá, že některé prvky dané procedury by mohly způsobit problémy.

Když používáme katalogy algoritmů a formální programovací metody, můžeme snadno zapomenout na to, že mnoho hezkých a elegantních metod vzniklo spontánně – jako záblesk geniality, intuice, umění... Nemohli bychom do této sbírky přispět také? Zkusme změřit své síly při řešení úloh, které mohou objektivně odpovědět na otázku, zda rekurzi jako programovací metodu do statečně ovládáme.

Úlohy

Vybrat reprezentativní sadu rekurzivních úloh nebylo vůbec snadné. Tato oblast je značně rozsáhlá a svým způsobem je zajímavé prakticky vše, co zde najdeme. Stejně jako na jiných místech knihy nakonec rozhodly praktické aspekty a jednoduchost.

Úloha 1

Předpokládejme, že chceme rekurzivním způsobem obrátit pole celých čísel. Navrhнěte algoritmus, který to provede pomocí „přirozené“ rekurze.

Úloha 2

Vraťme se k problému hledání určitého daného čísla x v poli, které je však tentokrát seřazeno od nejmenší hodnoty po největší. Velmi známá a efektivní metoda vyhledávání (tzv. *binární vyhledávání*) vychází z následujícího pozorování:

- Rozdělme pole velikosti n na poloviny: $t[0], t[1] \dots t[n/2-1], t[n/2], t[n/2+1], \dots, t[n-1]$.
 - Pokud $x = t[n/2]$, prvek x byl nalezen⁹.
 - Pokud $x < t[n/2]$, prvek x se možná nachází v levé polovině pole – analyzujme ji.
 - Pokud $x > t[n/2]$, prvek x se možná nachází v pravé polovině pole – analyzujme ji.

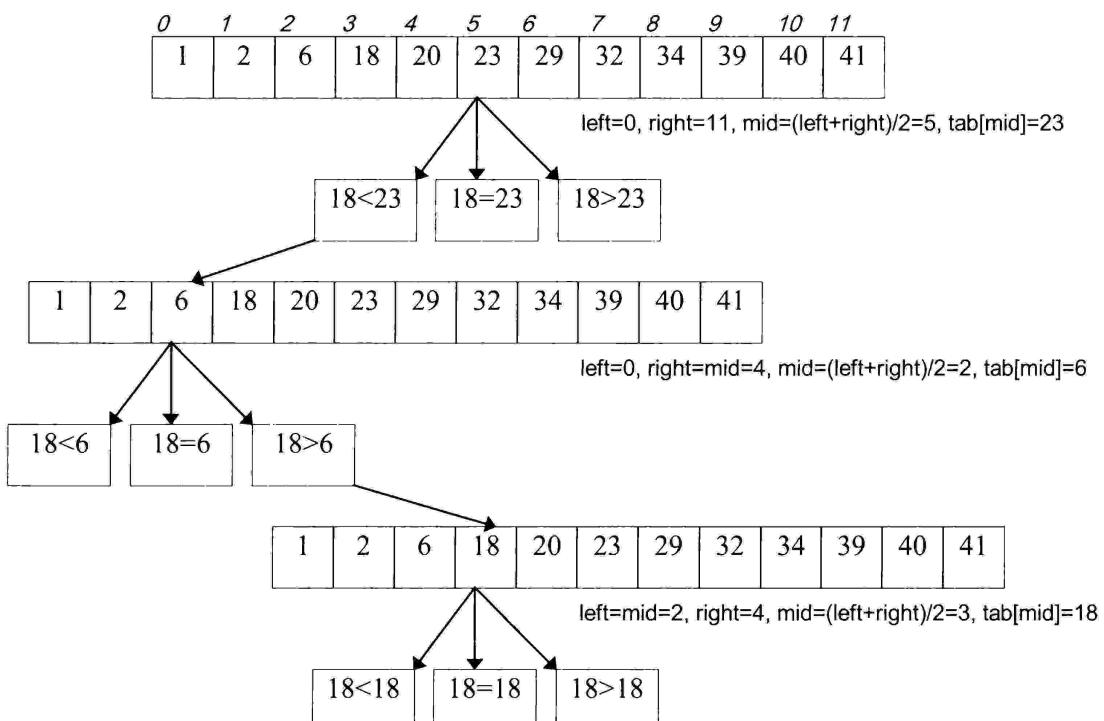
Slovo *možná* nám dává alibi, když se vyhledávání nezdáří. Naším úkolem je napsat dvě verze funkce, která realizuje uvedený algoritmus. Jedna funkce bude používat přirozenou rekurzi a druhá pro změnu nebude rekursivní.

Na obrázku 2.9 je znázorněna činnost algoritmu pro následující data:

- Pole s 12 prvků obsahuje tato čísla: 1, 2, 6, 18, 20, 23, 29, 32, 34, 39, 40, 41.
- Hledáme číslo 18.

Abychom mohli algoritmus podrobně prozkoumat, zavedeme několik pomocných proměnných:

- left – index prvku, který prohledávanou oblast pole ohraničuje zleva,
- right – index prvku, který prohledávanou oblast pole ohraničuje zprava,
- mid – index středního prvku aktuálně prohledávané oblasti pole.



Obrázek 2.9: Příklad binárního vyhledávání

⁹ Dělení v jazyce C++ poskytuje výsledek bez desetinné části (odpovídá funkci *div* v Pascalu).

KAPITOLA 2 Rekurze

Obrázek 2.9 představuje fungování algoritmu spolu s hodnotami proměnných `left`, `right` a `mid` během každé klíčové etapy. Vyhledávání skončilo úspěšně již po třech etapách¹⁰. Stojí za pozornost, že kdybychom stejnou úlohu řešili postupným prohlížením prvků pole zleva doprava, požadovaný prvek bychom našli teprve po čtvrté etapě. Minimální zlepšení oproti sekvenčnímu vyhledávání nejspíš nikoho neohromí, ale představme si, že pole by mohlo být několikrát větší než v tomto příkladu. Napište funkci, která binární vyhledávání realizuje rekurzivním způsobem.

Úloha 3

Napište funkci, která po zadání celého kladného čísla vypíše jeho *dvojkovou reprezentaci*. Využijte přitom známý algoritmus dělení základem číselné soustavy.

Ukažme si to při převodu čísla 13 na jeho dvojkovou formu:

$$13 : 2 = 6 + 1$$

$$6 : 2 = 3 + 0$$

$$3 : 2 = 1 + 1$$

$$1 : 2 = 0 + 1$$

↑

Konec algoritmu!

Problém spočívá v tom, že jsme sice dostali regulérní výsledek, ale pozpátku. Algoritmus poskytl hodnotu 1011, avšak správné pořadí číslic je 1101. Teprve na tomto místě úloha opravdu začíná:

Otázka 1

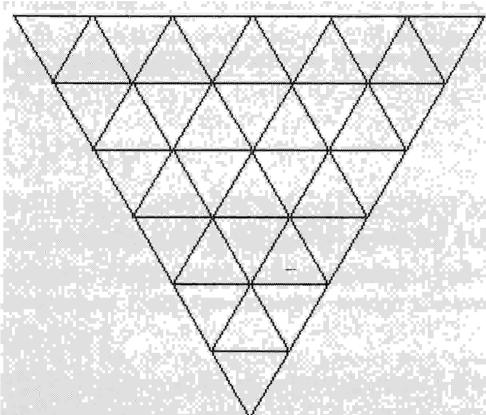
Jak pomocí rekurze obrátit pořadí vypisovaných číslic?

Otázka 2

Existuje nekomplikované a elegantní řešení tohoto úkolu, které by využívalo rekurze „s pomocným parametrem“?

Úloha 4

Zkuste napsat funkci, která by rekursivně vykreslila „kobereček“ znázorněný na obrázku 2.10.



Obrázek 2.10: Rekursivně vykreslené trojúhelníky

¹⁰ Za „etapu“ zde budeme považovat okamžik testování, zda jsme nalezli hledané číslo.

Úloha 5

Jako kondiční cvičení naprogramujte rekurzivním způsobem Eukleidův algoritmus – algoritmus vyhledávání největšího společného dělitele dvou čísel (zkráceně NSD nebo GCD podle anglického výrazu *greatest common divisor*). Největší společný dělitel NSD pro dvě přirozená čísla a, b : takové největší přirozené číslo c , že čísla a i b lze bez zbytku vydělit tímto číslem c .

Eukleidés si povšiml, že když od většího čísla odečteme menší, bude mít menší číslo i získaný rozdíl stejného největšího společného dělitele jako původní dvojice. Když při jednom z následných odečítání dostaneme dvojici stejných čísel, našli jsme hodnotu NSD.

Úloha 6

Napište program, který počítá faktoriál bez použití rekurze.

Řešení a poznámky k úlohám

Úloha 1

Řešení vychází z následující úvahy:

- Zaměňme krajní prvky pole (*elementární případ*).
- Obraťme zbývající část pole (*rekurzivní volání*).

Program založený na uvedeném principu vypadá takto:

```
rev_tab.cpp
// záměna proměnných
void swap(int& a, int& b)
{
    int temp=a;
    a=b;
    b=temp;
}

void obrat(int *tab, int left, int right)
{
    if(left<right)
    {
        swap(tab[left],tab[right]); // vyměňujeme krajní prvky
        obrat(tab,left+1,right-1); // obracíme zbytek
    }
}
int main()
{
    int tab1[8]={1,2,3,4,5,6,7,8};
    for(int i=0;i<8;i++)
        cout << tab1[i] << " ";
    obrat(tab1,0,7); // ukázka volání a ověření:
    for(int i=0;i<8;i++)
        cout << tab1[i] << " ";
}
```

Úloha 2

Dále ukážeme pouze rekurzivní verzi programu. Analogické iterativní řešení jistě snadno najdete sami!

```
binary_s.cpp
int hledej_rek(int * tab, int x, int left, int right)
{
    if(left>right)
        return -1; // prvek nebyl nalezen
    else
    {
        int mid=(left+right)/2;
        if(tab[mid]==x)
            return mid; // prvek byl nalezen!
        else
            if(x<tab[mid])
                return hledej_rek(tab,x,left,mid-1);
            else
                return hledej_rek(tab,x,mid+1,right);
    }
}
```

Úloha 3

Program nepatří mezi zvláště komplikované, i když vůbec není triviální. Zamysleme se nad tím, jak algoritmus donutit, aby uváděl výsledek v normálním pořadí, tj. zleva doprava. Kvůli tomu znovu rozeberme, jak funguje algoritmus založený na dělení základem číselné soustavy (zde 2). Číslo x dělíme dvěma, takže dostáváme hodnotu výrazu $[x \text{ div } 2]$ plus zbytek. Tento zbytek se samozřejmě rovná $[x \text{ mod } 2]$ a zároveň se jedná o poslední číslice dvojkové reprezentace, kterou chceme získat.

Existuje nějaký způsob, jak obrátit pořadí počítání dvojkových číslic a přitom stále používat tento jednoduchý algoritmus? Je to možné, ale musíme se na něj podívat trochu jinak. Všimněme si, jak lze symbolicky zapsat výpočet dvojkové reprezentace jistého čísla x s použitím operátorů, které patří do jazyka C++:

$$[x]_2 = [x \% 2] * \begin{bmatrix} x \\ 2 \end{bmatrix}_2$$

Tento zápis již naznačuje, jak je možné tento algoritmus vyjádřit rekurzivně:

$$[x]_2 = \text{elementární případ} * \text{rekurzivní volání}$$

Jestliže pomocí takového algoritmu počítáč požádáme, aby nejdříve dvojkově vypsal číslo $[x/2]$ a teprve poté výraz $[x \% 2]$ (který nabývá dvou hodnot: 0 nebo 1), objeví se číslice výsledku na monitoru v normálním pořadí¹¹, a nikoli obráceně jako v předchozím případě.

Tento trik stojí za zapamatování, protože se může hodit i v mnoha jiných programech.

¹¹ Případně viz kapitolu 7.

```
post_2.cpp
void post_dw(unsigned long int n)
{
    if(n!=0)
    {
        post_dw(n/2); // n modulo 2
        cout << n % 2; // zbytek po dělení číslem 2
    }
}
```

Co se týče druhé otázky, sám za sebe na ni mohu odpovědět: „možná“. Problém jsem sice díky rekurzi „s pomocným parametrem“ vyřešil, ale nepodařilo se mi najít natolik elegantní řešení, abych je zde mohl předvést. Možná tomu některý z čtenářů věnuje více času a bude úspěšnější? Rozhodně doporučuji, abyste to zkusili. I když se nedostanete do cíle, určitě se přitom naučíte více než čtením hotových řešení.

Úloha 4

Jedno z možných řešení vypadá takto:

```
trojkaty.cpp
void trojuhelniky(double n, double lg, double x, double y)
{
    // n = počet částí
    if (n>0)
    {
        double a=lg/n;
        double h=a*sqrt(3)/2.0;
        lineto(x-a/2.0,y-h);
        trojuhelniky(n-1,lg-a,x-a/2.0,y-h);
        lineto(x+a/2.0,y-h);
        for (double i=1;i<n;i++)
        {
            lineto(x+(i-1)*a/2.0,y-(i+1)*h);
            lineto(x+(i+1)*a/2.0,y-(i+1)*h);
        }
        lineto( x,y);
    }
}

int main()
{
    // inicializace grafického režimu
    moveto(getmaxx()/2,getmaxy()-10);
    trojuhelniky(6,getmaxx()/2,getx(),gety());
    getch();
    // ukončení grafického režimu
}
```

Úloha 5

Následuje jedno z možných řešení založené na rekurzi (v archivu ke stažení se nachází i klasičtější varianta):

```
nwd.cpp
int nsd (int a, int b)
{
    if (b==0)
        return a;
    else
        return nsd(b, a % b); // modulo
}
int main()
{
    cout << " nsd(12,3) =" << nsd(12,3) << endl;
    cout << " nsd(24,30) =" << nsd(24,30) << endl;
    cout << " nsd(5,7) =" << nsd(5,7) << endl;
    cout << " nsd(54,69) =" << nsd(54,69) << endl;
}
```

Úloha 6

Funkci faktoriál můžeme implementovat velmi snadno. Stačí si všimnout, že se jedná o řadu následných operací násobení (čtěme zprava doleva):

$$\begin{array}{ll} 0!=1 & \\ 1!=1*0! & \leftarrow \\ 2!=2*1! & \leftarrow \\ 3!=3*2! & \leftarrow \\ \dots & \leftarrow \end{array}$$

Jistě nám neujde, že chceme-li vypočítat následnou hodnotu faktoriálu, stačí mít k dispozici kumulovaný výsledek předchozích operací násobení, čehož dosáhneme pomocí jednoho cyklu v C++. Řešení najdete v programech, které patří ke knize, ale pokuste se na ně přijít sami bez nahlízení do hotového kódu.

KAPITOLA 3

Analýza složitosti algoritmů

Při výběru vhodného algoritmu se řídíme hlavně tím, v jakém kontextu jej hodláme používat. Jestliže chceme program přiležitostně spouštět v „domácím“ prostředí nebo bude jeho kód sloužit při výuce programování, budeme se obvykle rozhodovat podle *jednoduchosti algoritmu*.

Poněkud jiná situace nastává ve chvíli, kdy máme v úmyslu svůj program prodávat (případně jej zpřístupnit větší skupině osob).

Klient, který dostane binární program připravený k instalaci a používání, se jen okrajově (pokud vůbec) zajímá o vnitřní estetiku programu, srozumitelnost a eleganci použitých algoritmů a podobné aspekty. Takový uživatel – který se občas označuje jako *koncový* – se soustřeďuje na to, k čemu má bezprostřední přístup: propracované systémy nabídek, kontextovou návodědu, kvalitu prezentace výsledků v grafické formě atd. Programátor, který tyto požadavky koncových uživatelů zapomíná zohlednit, riskuje, že jeho produkt na trhu komerčních programů neuspěje.

Konflikt zájmů, který se zde projevuje, je samozřejmě typický pro všechny vztahy typu producent – klient: ve vlastním zájmu toho prvního je vytvořit produkt (v tomto případě program) co *nejlevněji* a prodat co *nejdráze*, zatímco ten druhý by chtěl za málo peněz získat produkt *nejvyšší kvality*. Abychom celou problematiku pro účely našeho výkladu poněkud zjednodušili, můžeme uvést základní kritéria hodnocení programu. Jedná se o:

- způsob komunikace s uživatelem,
- rychlosť provádění základních funkcí programu,
- stabilitu.

V této kapitole se budeme zabývat výhradně aspektem efektivní činnosti programů. Problematiku komunikace, která je příliš široká, ponecháme najinou příležitost, a hledisko stability softwaru nebudeme rozebírat vůbec.

Tématem této kapitoly je tzv. výpočetní složitost algoritmů. Tato vlastnost nám umožňuje odpovědět na otázku: *Který ze dvou programů, které plní stejný úkol (ale odlišnými způsoby), je efektivnější?* Odpověď v mnoha případech překvapivě vůbec není tak jednoduchá a vyžaduje, abychom nasadili poměrně složitý matematický aparát. Čtenáři se však obejdou bez znalostí vyšší mate-

V této kapitole:

- Definice a příklady
- Nový úkol: zjednodušení výpočtů
- Analýza rekurzivních programů
- Výpočetní náročnost není modla
- Techniky optimalizace programů
- Úlohy
- Řešení a poznámky k úlohám

KAPITOLA 3 Analýza složitosti algoritmů

matiky – metody, které představíme, budou značně zjednodušené a namísto teoretických studií orientované spíše na praktické aplikace.

Poznámka pro osoby, které mají hlubší zájem o matematickou stránku představených problémů a důkazy použitých metod atd.: Prezentované matematické nástroje byly zvoleny hlavně s ohledem na jejich jednoduchost, nikoli úplnost či soulad s veškerými formálními postupy. Podrobnější informace jsou k dispozici v příručkách matematické analýzy. Každý programátor přece nemusí být matematikem.

Čtenáři, kteří si potrpí na matematický formalizmus, mohou sáhnout po specializovaných publikacích, např. [BB87], [Gri87] či [Kro89], nebo také po klasických titulech [Knu08], [Knu10] a [Knu98].

Užitečné jsou i standardní učebnice matematiky, ale musíme si uvědomit, že často bývají poměrně obsáhlé a vybrat z nich potřebné informace bývá mnohem těžší než v případě titulů, které jsou určeny programátorům.

Definice a příklady

Při analýze efektivity programů můžeme rozlišit dva důležité faktory, které přispívají ke spokojnosti uživatelů:

- čas provádění (*ještě něco počítá, nebo už se zase zakousl?*),
- obsazení paměti (už mě nebabí zprávy typu: *Insufficient memory – save your work*¹).

Vzhledem ke značnému poklesu cen paměti RAM v posledních letech již druhé kritérium v zásadě ztratilo svůj význam². S tím prvním se však musíme potýkat i nadále. Současné počítače zdaleka nejsou tak rychlé³, abychom tento aspekt mohli pustit z hlavy. Co z toho, že počítač X je dvanáctkrát rychlejší než počítač Y, když to v případě algoritmu A a problému P znamená, že výpočty místo za 12 let dokončíme za „pouhý“ jeden rok?! Odhlédneme přitom od toho, že by nikdo algoritmus A k tomuto úkolu nepoužil. Stejný problém lze často vyřešit jiným algoritmem, který poskytne výsledek třeba už za několik hodin.

Základním úkolem analýzy algoritmů je výběr vhodné *míry výpočetní složitosti*. Zvolené měřítko musí být natolik reprezentativní, aby se vztahovalo na všechny uživatele stejného algoritmu, ať už pracují s levným osobním počítačem nebo výkonnou pracovní stanicí. Pokud někdo prohlásí, že *jeho program je rychlý, protože skončil už po jedné minutě*, nemá tato informace sama o sobě žádnou vypovídací hodnotu. Museli bychom ještě položit další otázky, například:

- O jaký počítač se jedná?
- Kolik dat program zpracovával?
- Jakou taktovací frekvenci má procesor počítače?
- Byl program během své činnosti v paměti sám? Pokud nikoli, jakou měl prioritu?

1 *Nedostatek paměti – uložte svá data*: tato chyba se kdysi často objevovala i v mnoha profesionálních programech pro systém Windows. Díky velkým pevným diskům, které simulují operační paměť, se tato chyba v současnosti už totikdy nevyskytuje. Vděčíme za to ovšem spíše technologickému pokroku než vyšší jakosti programů.

2 Ve speciálních oblastech toto tvrzení neplatí. Některé algoritmy používané při zpracování obrazu spotřebují totikdy paměti, že se s nimi v osobních počítačích prakticky nesetkáme. Kromě toho si musíme uvědomit, že obsluha komplikovaných datových struktur obecně bývá dosti časově náročná. Obě kritéria se tedy vzájemně ovlivňují.

3 Pochopitelně máme na mysli osobní počítače.

- Kterým komplátorem byl zpracován zdrojový kód?
- Pokud to byl komplátor XYZ, byly přitom zapnuty možnosti optimalizace kódu?

Hned je nám jasné, že tímto způsobem se daleko nedostaneme. Potřebujeme *univerzální metriku*, která by nijak nesouvisela s detaily hardwarové povahy.



Poznámka: Ve „fyzickém“ světě, kde pracujeme s diskovými paměti, sítěmi a pomalými přenosy, se často operuje pojmem *výkonnosti softwaru*. Odběratel programu dodavateli popíše své požadavky na výkon (např. čas zpracování měsíčních sestav) a dodavatel se snaží tyto požadavky splnit. Bere přitom v úvahu všechny technické parametry. Koncepce výkonnosti softwaru je však značně nepřesná, protože ve skutečnosti příliš nevypovídá o aplikovaných algoritmech. Ve výše zmíněném příkladu by bylo mnohem zajímavější porovnat čas zpracování měsíční sestavy v případě zdvojnásobení objemu vstupních dat (např. počtu záznamů v databázi).

Čas dokončení daného algoritmu je nejčastěji určen rozměrem dat n , která tento algoritmus zpracovává⁴. Pojem *rozměru dat* není jednoznačný: pro funkci na třídění pole to bude jednoduše délka příslušného pole, zatímco v případě programu na výpočet funkce faktorial se bude jednat o velikost vstupní hodnoty.

Podobně funkce, která vyhledává data v seznamu (viz kapitolu 5), bude hodně „citlivá“ na délku seznamu. Ve všech uvedených případech se jedná o rozdíl vstupních dat. Vzhledem k tomu, že tento termín lze intuitivně dobře pochopit, nebudeme jej nadále zdlouhavě vysvětlovat a spokojíme se s jeho nepřesným významem.

Obecně můžeme říci, že n je nejvýznamnějším parametrem, který ovlivňuje čas dokončení algoritmu.

Vraťme se ještě k příkladu, který jsme zmínili v předchozích odstavcích. Nepřipravený čtenář může po zmínce o programu, který dokončí svou práci až za 12 let, poněkud zapochybavit: je to vůbec možné?! Díky technologickému rozvoji přece máme k dispozici stále rychlejší počítače. Kdysi možná programy mohly potřebovat 12 let, ale dnes?

Bohužel musíme zdůraznit, že uvedený čas vůbec není kdovíjak dlouhý. Podívejme se na tabulku 3.1.

Tabulka 3.1: Čas provádění programů pro algoritmy různé třídy

	10	20	30	40	50	60
n	0,000 01 s	0,000 02 s	0,000 03 s	0,000 04 s	0,000 05 s	0,000 06 s
n^2	0,000 1 s	0,000 4 s	0,000 9 s	0,001 6 s	0,002 5 s	0,003 6 s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
2^n	0,001 s	1,0 s	17,9 min	12,7 dní	35,7 roků	366 stol.
3^n	0,59 s	58 min	6,5 roku	3855 stol.	200 × 10 ⁶ stol.	1,3 × 10 ¹³ stol.
$n!$	3,6 s	768 stol.	8,4 × 10 ¹⁶ stol.	2,6 × 10 ³² stol.	9,6 × 10 ⁴⁸ stol.	2,6 × 10 ⁶⁶ stol.

Tabulka shrnuje několik časových údajů o provádění algoritmů⁵. Vychází se přitom z těchto předpokladů:

⁴ V dalším výkladu ukážeme, že čas provádění programu závisí i na jiných činitelích.

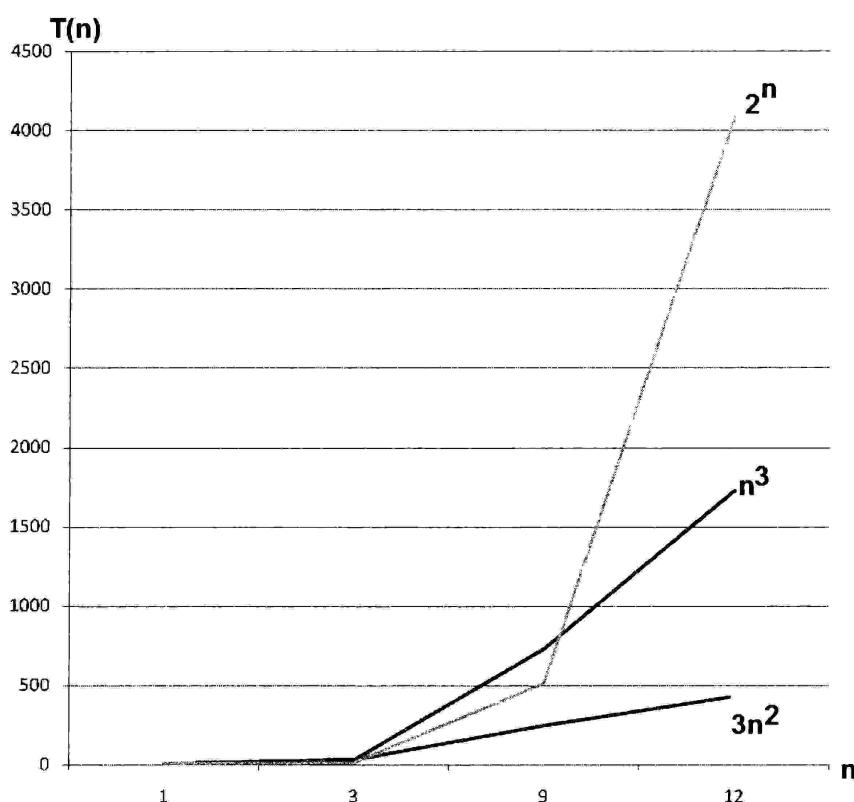
⁵ Legenda: s – sekunda, stol. – století.

KAPITOLA 3 Analýza složitosti algoritmů

- Čas provádění algoritmu A je úměrný určité vybrané matematické funkci, např. pro vstupní hodnotu velikosti x a funkci $n!$ čas provádění programu odpovídá hodnotě $x!$.
- Jedna elementární operace trvá jednu mikrosekundu (např. pro $n = 10$ a funkci faktoriál dostaváme $10! = 3628800/1000000 = 3,6$ s).

Za uvedených předpokladů můžeme dospět k výsledkům uvedeným v tabulce, které jsou zejména v její pravé části dosti šokující.

Všimněme si, jak rychle vzrůstá čas provádění programů u zdánlivě blízkých tříd funkcí (obrázek 3.1) dokonce i při malém zvětšení hodnoty n .



Obrázek 3.1: Grafické znázornění časů provedení algoritmů různé třidy

Záměrně jsme vybrali velmi malé hodnoty n , protože jen pro takové hodnoty se výsledky vůbec vejdu do grafu!

Na základě tohoto obrázku můžeme dojít k důležitému závěru: Pouze pro třídy s nízkou složitostí (blízké složitosti lineární) můžeme díky většímu výpočetnímu výkonu počítače v praxi řešit rozsáhlejší problémy. Stačí nám jednoduchá simulace, abychom se přesvědčili, že u časově „nákladných“ funkcí nezískáme zrychlením počítače tolik času, abychom mohli zpracovat více vstupních dat.

Toto tvrzení lépe objasníme na obrázku 3.2.

Dvě uvedené tabulky shrnují časy provádění programů tříd $100n$ a 2^n v intervalu n od 10 do 10 000 000. Souběžně jsou přitom uvedeny základní výsledky a hodnoty po tisícinásobném zvětšení výpočetního výkonu. Zajímá nás hraniční čas realizace kolem milionu (např. sekund) a chceme zjistit, jaký rozsah hodnot n dokáží oba programy v tomto vymezeném čase zpracovat.

Před zrychlením			Po zrychlení 1000x		
n	100n	2^n	100n	2^n	
10	1 000	1 024	1	1	
20	2 000	1 048 576	2	1 049	
30	3 000	1 073 741 824	3	1 073 742	
40	4 000	1 099 511 627 776	4	10 995 116 278	
50	5 000		5		
100	10 000		10		
1 000	100 000		100		
10 000	1 000 000		1 000		
100 000	10 000 000	...	10 000		
1 000 000	100 000 000	...	100 000		
10 000 000	1 000 000 000	...	1 000 000		

Obrázek 3.2: Ošidná důvěra v možnosti počítačů a realita

Ukazuje se, že počet problémů, které lze řešit v daném čase, vzrůstá v případě programu třídy $100n$ dramaticky (o tři řády), zatímco u druhého algoritmu jich dokážeme zpracovat přibližně jen o 50 % více (praktický rozsah 10–20 se rozšiřuje sotva na 10–30)!

Nyní by všichni pochybovači měli uznat, jaký význam mají znalosti z oboru složitosti algoritmů, díky nimž se můžeme vyhnout nekonečnému čekání na výsledek programu...

Abychom lépe porozuměli matematickým metodám, které se při analýze složitosti algoritmů používají, společně rozebereme několik typických početních úkolů. Jednotlivé postupy výpočtu složitosti algoritmů si přiblížíme na reprezentativních příkladech, které jsou názornější než suché definice.

Funkce faktoriál znovu

Na funkci faktoriál nás může překvapit, kolik otázek lze pomocí ní ilustrovat. Z předchozí kapitoly si nepochybňě ještě pamatujete rekurzivní definici:

$$\begin{aligned}0! &= 1, \\ n! &= n \cdot (n-1)! , n \geq 1, n \in N\end{aligned}$$

Odpovídající funkce v jazyce C++ má následující podobu:

```
int faktorial(int n)
{
    if (n==0)
        return 1;
    else
        return n*faktorial(n-1);
}
```

Pro zjednodušení budeme vycházet z toho, že časově nejnáročnější operací bude v tomto případě instrukce porovnání `if` (což je ostatně pro úlohy tohoto druhu typické). Za tohoto předpokladu můžeme také čas provedení programu zapsat rekurzivním způsobem:

KAPITOLA 3 Analýza složitosti algoritmů

$$T(0) = t_c,$$

$$T(n) = t_c + T(n - 1) \text{ pro } n \geq 1.$$

Výše uvedené vzorce můžeme přečíst takto: Pro vstupní hodnotu rovnou nule se čas provedení funkce označovaný jako $T(0)$ rovná času provedení jedné instrukce porovnání, kterou symbolicky označujeme jako t_c . Analogický čas pro vstupní data ≥ 1 je v souladu s rekurzivním vzorcem určen vztahem $T(n) = t_c + T(n - 1)$.

Zápis tohoto typu nám bohužel nijak nepomůže – těžko bychom například dokázali okamžitě říci, jak dlouho bude trvat výpočet faktorial(100). Nyní je jasné, že problém musíme řešit poněkud jinak. Zamysleme se, jak z této soustavy rovnic vyjádřit $T(n)$, abychom dostali nějakou nerekurzivní funkci, která by nám ukázala závislost času provedení programu na vstupní hodnotě n . Za tímto účelem zkusme rozepsat rovnice:

$$T(n) = t_c + T(n - 1),$$

$$T(n - 1) = t_c + T(n - 2),$$

$$T(n - 2) = t_c + T(n - 3),$$

$$T(1) = t_c + T(0),$$

$$T(0) = t_c.$$

Když nyní sečteme levé a pravé strany, získáme vztah:

$$T(n) + T(n - 1) + \dots + T(0) = (n + 1)t_c + T(n - 1) + \dots + T(0),$$

ze kterého bychom po odstranění stejných prvků na obou stranách rovnice měli dostat následující závislost:

$$T(n) = (n + 1)t_c.$$

Tato funkce uspokojivě a jednoduše znázorňuje, jakým způsobem velikost vstupní hodnoty ovlivňuje počet instrukcí porovnání, které program provádí – čili v důsledku čas provedení algoritmu. Známe-li totiž parametr t_c a hodnotu n , můžeme přesně určit, za kolik sekund (minut, hodin, let...) dokončí algoritmus svou činnost v určitém počítači.

Výsledky těchto přesných výpočtů obvykle nazýváme *praktickou složitostí* algoritmu. Příslušnou funkci obvykle označujeme stejně jako v uvedeném příkladu – písmenem T .

V praxi zpravidla nepotřebujeme úplně přesný výsledek. U dostatečně velkých čísel n totiž téměř nezáleží na tom, když místo $T(n) = (n + 1)t_c$ dostaneme $T(n) = (n + 3)t_c$.

Co tím naznačujeme? V dalších úvahách budeme hledat hlavně odpovědi na otázky typu:

Jaký typ matematické funkce, který se vyskytuje v závislosti vyjadřující praktickou složitost programu, v ní hráje nejdůležitější roli a nejvýrazněji ovlivňuje čas provádění programu?

Tuto hledanou funkci budeme nazývat *teoretickou složitostí* nebo třídou algoritmu. Právě na ni můžeme v katalogových popisech algoritmů narazit nejčastěji. Funkce se nejčastěji označuje symbolem O . Zamysleme se, jakým způsobem ji můžeme najít.

Existují dva klasické přístupy, které zpravidla vedou ke stejnemu výsledku: buď můžeme vycházet z určitých matematických tvrzení a aplikovat je v příslušných situacích, nebo ke správnému výsledku dojdeme intuitivní metodou.

Podle názoru autora je druhý přístup rychlejší a současně i mnohem přístupnější, takže se zaměříme nejdříve na něj. Podívejme se tedy na tabulku 3.2, která obsahuje několik příkladů, jak lze z rovnic vyjadřujících praktickou složitost algoritmu zjistit jeho teoretickou složitost.

Výsledky shrnuté v tabulce můžeme vysvětlit následovně: v rovnici $3n+1$ si dovolíme vynechat konstantu 1 a výsledek se nijak zásadně nezmění. V rovnici n^2-n+1 je mnohem důležitější kvadratický člen než lineární závislost na hodnotě n . Obdobně v poslední rovnici dominuje funkce 2^n .

Tabulka 3.2: Teoretická složitost algoritmů – příklady

T(n)	O
6	O(1)
$3n + 1$	O(n)
$n^2 - n + 1$	O(n^2)
$2^n + n^2 + 4$	O(2^n)

Uvedeme nyní několik typických funkcí O , se kterými se v algoritmice často můžeme setkat:

- Třída $O(1)$ z definice označuje, že počet operací prováděných algoritmem nezávisí na rozsahu problému⁶. Může taková situace v případě netriviálních algoritmů vůbec nastat? Ukazuje se, že s jistým zjednodušením může. Jako příklad lze uvést vyhledávací funkci, která je založena na hešování (touto technikou se budeme zabývat v kapitole 7). Pokud nám jistá vhodná funkce H umožní najít hledaný záznam ve víceméně stejném konečném čase, na velikosti prohledávané množiny teoreticky nezáleží. Toto tvrzení zatím může znít poněkud nevěrohodně, ale po přečtení zmíněné kapitoly se přesvědčíme, že je pravdivé.
- Třída $O(n)$ znamená, že algoritmus funguje v čase úměrném velikosti problému. Příkladem takového algoritmu může být sekvenční zpracování znakového řetězce, obsluha fronty atd. Jedná se o jednoduchou lineární závislost, kde každé z n vstupních hodnot musí algoritmus věnovat jistý čas, aby splnil svůj úkol.
- Složitost $O(\log n)$ je lepší než lineární⁷. Aritmeticky to znamená, že jestliže rozsah problému roste geometricky (například o řád ze 100 na 1 000), bude výpočetní náročnost vzrůstat aritmeticky (zde dvojnásobně). Pokud n neroste příliš rychle, algoritmus se sice zpomaluje, ne však výrazně. Z logaritmickou složitostí se setkáme například v algoritmu prohledávání setříděného pole, kdy algoritmus v každém kroku vynechává část dat (viz kapitolu 7).
- Třída $O(n^2)$ se často objevuje v aritmetických či kombinatorických úlohách, kde se uplatňuje pravidlo „každý s každým“. Jako příklad je možné zmínit sčítání matic rozměru $n \times n$. Analogicky u třídy $O(n^3)$ lze uvést násobení matic velikosti $n \times n$. Nikoho asi nepřekvapí, že algoritmy „kvadratického“ a vyššího typu se dají nasadit spíše pro menší hodnoty n .
- Exponenciální třída $O(2^n)$ se často uvádí jako určitý strašák, ačkoli v praxi se algoritmy této třídy také dají používat – samozřejmě v případě, že vracejí výsledky v uživatelsky snesitelném čase.

Mohli bychom pokračovat ještě dalšími názornými příklady funkcí O , ale princip algoritmické složitosti je natolik klíčový, že přejdeme k formální matematické definici.

Za tímto účelem si zopakujme následující symboly, které známe z příruček matematické analýzy:

6 To vůbec neznamená, že bude malý: číslo „1“ se často mylně zaměňuje s jedinou instrukcí!

7 Připomeňme, že logaritmus čísla $x > 0$ o základu $b \neq 1$, který zapisujeme jako $u = \log_b x$, je takové číslo u , které splňuje závislost $b^u = x$, např. $3 = \log_2 8$.

KAPITOLA 3 Analýza složitosti algoritmů

Legenda:

- t_a – čas provedení instrukce přiřazení,
- t_c – čas provedení instrukce porovnání.

V dalších úvahách musíme rozumět tomu, jak funguje cyklus typu `while`:

```
i=1;
while (i<=n)
{
    instrukce;
    i=i+1;
}
```

Její princip spočívá v tom, že n -krát provede instrukce obsažené mezi složenými závorkami. Podmínka se tedy testuje $n+1$ -krát⁸.

Na základě tohoto vysvětlení a informací uvedených v komentářích kódů můžeme napsat:

$$T(n) = t_c + t_a + \sum_{i=1}^N \left(2t_a + 2t_c + \sum_{j=1}^i (t_c + 2t_a) \right)$$

Po odstranění sumy z vnitřních závorek dostaneme (*):

$$T(n) = t_c + t_a + \sum_{i=1}^N (2t_a + 2t_c + i(t_c + 2t_a))$$

Připomeňme ještě užitečný vzorec na výpočet součtu posloupnosti přirozených čísel od 1 do N :

$$1 + 2 + 3 + \dots + N = \frac{N(N + 1)}{2}$$

Po jeho dosazení do rovnice (*) získáme:

$$T(n) = t_c + t_a + 2N(t_a + t_c) + \frac{N(N + 1)}{2}(t_c + 2t_a)$$

Poslední zjednodušení by nám mělo poskytnout tuto rovnici:

$$T(n) = t_a(1 + 3N + N^2) + t_c\left(1 + 2,5t_c + \frac{N^2}{2}\right)$$

což okamžitě naznačuje, že analyzovaný program má třídu $O(n^2)$.

Ufff!

Nebylo to zrovna příjemné, že? A problém přitom nepatří mezi zvláště složité. Nenechme se však náročným začátkem odradit. Zákratko se ukáže, že jsme se k cíli mohli dostat mnohem snáze. Budeme k tomu však potřebovat trochu teoretických znalostí, abychom dokázali řešit rekurzivní

⁸ Stojí za zmínku, že stávající cykly v jazyce C++ lze snadno převést na uvedenou verzi.

rovnice. Tyto znalosti získáme poté, co projdeme následující příklad. Obsahuje jednu past, kterou bohužel musíme alespoň jednou poznat na vlastní kůži.

Chytáme se do pasti

Oba předchozí příklady se vyznačovaly podstatnou vlastností: čas provádění programu nezávisel na hodnotě vstupních dat, ale pouze na jejich rozsahu. Někdy však tento předpoklad bohužel neplatí. Právě s takovou situací se setkáme v následující početní úloze. Jedná se o část většího programu, jehož funkce pro nás na tomto místě není důležitá. Dejme tomu, že dostaneme tento fragment kódu vytržený z kontextu a máme jej analyzovat:

```
const int N=10;
int t[N];
funkce_ad_hoc()
{
    int k,i;
    int suma=0;          // ta
    while (i<N)          // tc
    {
        while (j<=t[i]) // tc
        {
            suma=suma+2; // ta
            j=j+1;         // ta
        }
        i=i+1;           // ta
    }
}
```

Problém můžeme poněkud zjednodušit na základě těchto předpokladů:

- Časově nejnáročnější jsou instrukce porovnání. Všechny ostatní instrukce tedy budeme ignorovat, protože čas provádění programu významně neovlivňují.
- Místo toho, abychom explicitně zadávali hodnotu t_c , zavedeme princip jednotkového času provedení instrukce, který označíme číslem 1.

Jedna zásadní komplikace však zůstává: neznáme obsah pole, a nevíme tedy, kolikrát se provede vnitřní cyklus `while`. Podívejme se, jak si můžeme v tomto případě poradit:

$$T(n) = t_c + \sum_{i=1}^N \left(t_c + \sum_{j=1}^{t[i]} t_c \right) \quad (*)$$

$$T(n) = t_c + Nt_c \sum_{i=1}^N t[i]t_c \quad (**)$$

$$T(n) = t_c + Nt_c + Nt[i]t_c$$

$$T(n) = t_c(1 + N + Nt[i])$$

$$T(n) \approx \max(N, Nt[i])$$

KAPITOLA 3 Analýza složitosti algoritmů

Začínáme klasicky: vnější sumu od 1 do N z rovnice (*) nahradíme N -násobným součinem jejího argumentu. Podobný trik provedeme i s rovnici (**) a poté již můžeme spokojeně přejít k seskupování členů a zjednodušování. Čas provádění programu je úměrný většimu z čísel: N a $Nt[i]$. Zatím nedokázeme říci nic víc. Na tomto místě svých úvah bychom se však mohli chytit do pasti. Náš problém spočívá v tom, že neznáme obsah pole, a bez těchto dat nedostaneme konečný výsledek! Nemůžeme přece aplikovat matematickou funkci na neurčitou hodnotu.

Ve svých výpočtech jsme se dostali do fáze, kdy si uvědomujeme, že o analyzovaném problému nemáme úplné informace. Kdybychom například věděli, že fragment programu se zkoumanou funkcí bude s velkou pravděpodobností zpracovávat pole, které je vyplňeno převážně nulami, celý problém by najednou zmizel. Nemáme však žádnou jistotu, že tento předpoklad skutečně platí. Můžeme se jen obrátit na nějakého matematika, který by – na základě mnoha předpokladů – provedl statistickou analýzu úlohy a došel k závěru, který by měl pro nás uspokojivou formu.

Různé typy výpočetní složitosti

Vraťme se opět k problému: potřebujeme zkontrolovat, zda se určité číslo x nachází v poli velikosti n . Tento úkol jsme již vyřešili v kapitole 2, ale nyní se pokusíme napsat iterativní verzi stejné procedury. Toto zadání není kdovíjak složité a výsledný program vypadá asi takto:

```
szukaj.cpp
const int n=10;
int tab[n]={1,2,3,2,-7,44,5,1,0,-3};
int hledej(int tab[],int x)
{
    int pos=0;
    while ((pos<n) && (tab[pos]!=x)) pos++;
    if(pos<n)
        return pos; // prvek byl nalezen
    else
        return -1; // prvek nebyl nalezen
}

int main()
{
    cout << hledej(tab,7) << endl; // výsledek = -1
    cout << hledej(tab,5) << endl; // výsledek = 6
}
```

Princip algoritmu spočívá v kontrole toho, zda má levý krajní prvek zkoumané části pole hledanou hodnotu x . Když proceduru vyvoláme příkazem `hledej(tab,x)`, program prohledá celé pole velikosti n . Co můžeme říci o výpočetní složitosti tohoto algoritmu, když jako kritérium zvolíme počet porovnání, která se provedou v cyklu `while`? Na takto formulovanou otázku můžeme jen pokrýt rameny a zamumlal: „Záleží na tom, kde leží prvek x .“ Můžeme se samozřejmě setkat se dvěma krajními případy:

- Prvek je uložen v buňce `tab[0]`, tedy $T(n) = 1$ a narazili jsme na tzv. nejlepší případ.
- Při hledání prvku x prohlédneme celé pole, tedy $T(n) = n$, což znamená, že se jedná o tzv. nejhorší případ.

Jestliže na jednu přesně formulovanou otázku: „Jaká je výpočetní složitost algoritmu lineárního prohledávání pole s n prvky?“ dostaneme dvě reakce uvedené klauzulemi „*pokud*“ nebo „*v případě, že...*“, je jasné, že odpověď na svou otázku stále nemáme!

Chyba samozřejmě spočívá v otázce, která by měla zohlednit konfiguraci dat. Ta má totiž při prohledávání pole klíčový význam. Můžeme tedy navrhnut následující odpovědi: Praktická složitost uvažovaného algoritmu je v nejlepším případě dána vztahem $T(n) = 1$ a v nejhorším případě ji popisuje vztah $T(n) = n$. Život se většinou odvíjí poklidně a vyhýbá se krajnostem (toto tvrzení je sice poněkud sporné, ale na chvíli připustme, že platí). Bude nás proto zajímat i odpověď na otázku: Jaká je průměrná hodnota $T(n)$ tohoto algoritmu? Na otázky postavené tímto způsobem umí odpovědět statistika. Nezbývá nám tedy nic jiného než provést statistickou analýzu diskutovaného algoritmu.

Symbolem p označme pravděpodobnost, že se prvek x nachází v poli tab , a předpokládejme, že pokud se prvek x v poli skutečně vyskytuje, jsou všechny jeho indexy stejně pravděpodobné.

Zápisem $D_{n,i}$ (kde $0 \leq i < n$) dále označme množinu dat, kde se prvek x nachází na i -tém místě pole, a zápisem $D_{n,n}$ množinu dat, která prvek x neobsahuje. Na základě výše uvedené symboliky můžeme prohlásit, že:

$$P(D_{n,i}) = \frac{p}{n} \quad P(D_{n,n}) = 1 - p$$

Náklady algoritmu označíme klasicky písmenem T , takže:

$$T(D_{n,i}) = i \quad T(D_{n,n}) = n$$

Získáme tedy výraz:

$$T_{\text{průměr}} = \sum_{i=0}^N P(D_{n,i}) T(D_{n,i}) = (1-p)n + \sum_{i=0}^{n-1} i \frac{p}{n} = (1-p)n + (n+1) \frac{p}{2}$$

Například: Víme-li, že prvek x v poli určitě existuje ($p = 1$), můžeme okamžitě napsat:

$$T_{\text{průměr}} = (1-1)n + \frac{(n+1)}{2} = \frac{(n+1)}{2}$$

Definovali jsme tedy tři základní typy výpočetní složitosti (pro tyto případy: *nejhorší*, *nejvýhodnější* i *průměrný*). Nyní bychom se měli zamyslet nad jejich praktickým významem. Z matematického hlediska tyto tři definice plně určují chování algoritmu. Skutečně je však potřebujeme?

V katalogových popisech algoritmů se nejčastěji setkáme s rozborem *nejhoršího* případu, abychom znali určitou horní hranici, kterou algoritmus určitě nepřekročí (tato informace je pro programátora nejužitečnější).

Podobný charakter má i *nejvýhodnější* případ, který se však týká dolního časového limitu činnosti programu.

Vidíme, že principy *nejlepší* i *nejhorší* výpočetní složitosti programu nemají jen matematický smysl, ale také informují programátory, v jakých hranicích algoritmus funguje. Můžeme podobně přistupovat i k *průměrnému* případu?

Je zřejmé, že výpočet průměrného případu (jinými slovy *typického*) není snadný a musíme přitom přijmout řadu hypotéz ohledně možných konfigurací dat. Mimo jiné je nutné, abychom se shodli na definici datové množiny, kterou program zpracovává. Bohužel to však zpravidla není možné a nemá to ani žádný smysl. Programátor, který dostane informaci o průměrné výpočetní složitosti programu, by si proto měl uvedená omezení uvědomovat a brát tento parametr s rezervou.

Nový úkol: zjednodušení výpočtu

Nelze popřít, že všechny dosavadní výpočty byly poněkud komplikované, a to líným lidem (čti: programátorem) nevyhovuje. Jak tedy postupovat, abychom si hledání výsledku co nejvíce zjednodušili? Za tímto účelem bychom si měli zapamatovat následující triky, které nám značně usnadní postup a často se díky nim můžeme k požadovanému cíli dostat okamžitě:

- Při analýze programu se zabýváme pouze časově nejnáročnějšími operacemi (např. v předchozích příkladech to byly instrukce porovnání).
- Vybereme jeden řádek programu, který se nachází v nejvíce zanořené iterační instrukci (cykly v cyklech a ty zase v jiných cyklech), a následně počítáme, kolikrát jej program provede. Podle tohoto výsledku posuzujeme teoretickou složitost.

První způsob jsme již vyzkoušeli. Abychom podrobněji pochopili druhou metodu, prostudujme následující část programu:

```
while (i<N)
{
    while (j<=N)
    {
        suma=suma+2;
        j=j+1;
    }
}
```

Vybereme instrukci `suma=suma+2` a jednoduchým způsobem spočítáme, že se vykoná $N(N + 1)/2$ krát. Můžeme prohlásit, že teoretická složitost tohoto fragmentu programu se rovná $O(n^2)$.

Analýza rekurzivních programů

Většinu rekurzivních programů naneštěstí pomocí výše uvedené metody analyzovat nelze. Často uplatňovaná metoda řešení rekurzivní rovnice, která spočívá v rozepsání členů a sečtení příslušných stran soustavy rovnic, pokaždé nevede k cíli. V našem případě byla úspěšná, protože umožnila výpočet zjednodušit. Bohužel se však rovnice upravené tímto způsobem obvykle ještě více komplikují.

V tomto odstavci představíme metodu mnohem obecnějšího typu. Má sice své matematické zdůvodnění, ale to vzhledem k jeho složitosti vynecháme. Čtenáři, kteří se více zajímají o matematickou stránku problematiky, mohou snadno sáhnout po příslušné literatuře (viz poznámky v úvodu kapitoly).

Terminologie a definice

Při čtení následujících odstavců se neobejdeme bez znalosti speciálních termínů, se kterými se vícekrát setkáme. Navzdory „hrozivému“ vzhledu následujících definic by čtenářům nemělo jejich pochopení činit zvláštní problémy.

Lineární rekurzivní posloupnost LRP je posloupnost následujícího typu:

$$X_{n+r, n \geq 0} = \sum_{i=1}^r a_i X_{n+r-i} + u(n, m)$$

Legenda:

$u(n,m)$ – nerekurzivní zbytek rovnice, která je mnohočlenem m -tého stupně proměnné n .

Například:

- pokud $u(n,m) = 3n + 1$, máme mnohočlen *prvního stupně*,
- jestliže $u(n,m) = 2$, jedná se o mnohočlen *nultého stupně*.

Poznámky:

- koeficienty a_i jsou libovolná reálná čísla,
- r je celé číslo.

Komplikovanou podobou tohoto výrazu se nenechejme odradit. Jedná se prostě o formalizovaný zápis obecné rekurzivní rovnice, který uvádíme spíše pro úplnost než kvůli nějakému praktickému využití.

Charakteristická rovnice CHR je mnohočlen uměle odvozený z rekurzivní rovnice, který se tvoří podle schématu:

$$R(x) = X^r - \sum_{i=1}^r a_i x^{r-i}$$

Řešením této rovnice lze získat rozklad ve tvaru:

$$R(x) = \prod_{i=1}^p (x - \lambda_i)^{m_i}$$

Příklad: $LRP = x_n - 3x_{n-1} + 2x_{n-2} = 0$ dává $R(x) = x^2 - 3x + 2 = (x - 1)(x - 2)$.

Výše získané koeficienty λ_i umožňují zkonstruovat tzv. **obecné řešení OR** lineární rekurzivní rovnice:

$$RO = \sum_{i=1}^p P_i \lambda_i$$

Kromě toho budeme potřebovat tzv. *partikulární řešení PŘ* lineární rekurzivní rovnice.

Tvar **PŘ** závisí na formě, kterou nabývá zbytek $u(n,m)$. Níže jsou rozepsány konkrétní případy.

- Pokud $u(n,m) = 0$, pak **PŘ** = 0.
 - Pokud $u(n,m)$ je mnohočlen m -tého stupně proměnné n a zároveň 1 (jedna) není řešením **CHR**, pak **PŘ** = $Q(n,m)$, kde $Q(n,m)$ je určitý mnohočlen m -tého stupně proměnné n s neznámými koeficienty (které je nutné najít).
 - Pokud $u(n,m)$ je mnohočlen m -tého stupně proměnné n a zároveň 1 je řešením **CHR**, pak **PŘ** = $n^p Q(n,m)$, kde p je stupeň kořene.
- Například: Jestliže je číslo 1 jednoduchým kořenem **CHR**, pak $p = 1$, jestliže je dvojnásobným kořenem, pak $p = 2$ atd.
- Pokud $u(n,m) = \alpha^n$ a α není řešením **CHR**, pak:

$$Př = c\alpha^n.$$

KAPITOLA 3 Analýza složitosti algoritmů

- Pokud $u(n,m) = \alpha^n$ a α je kořenem stupně p CHR, pak:
 $P\check{R} = c\alpha^nn^p$.
- Pokud $u(n,m) = \alpha^nW(n,m)$ a α není řešením CHR (kde $W(n,m)$ je už tradičně určitý mnohočlen m -tého stupně proměnné n), dostáváme:
 $P\check{R} = \alpha^nS(n,m)$,
kde $S(n,m)$ je určitý mnohočlen m -tého stupně proměnné n .



Poznámka: Mnohočleny a konstanty na pravé straně mají charakter hledaných proměnných.

Řešením rekurzivní rovnice je součet obou rovnic: *obecné* a *partikulární*.

Bez té spousty vzorců to opravdu nešlo. Metodu představíme při řešení jednoduché úlohy.

Vysvětlení metody na příkladu

Podívejme se znovu na příklad ze strany 55, který se týká funkce na výpočet faktoriálu. Dostali jsme tehdy následující rovnice:

$$\begin{aligned}T(0) &= 1, \\T(n) &= 1 + T(n - 1).\end{aligned}$$

Zkusme je nyní vyřešit nově představenou metodou.

- **1. fáze:** Hledání charakteristické rovnice:
Z obecného tvaru $LRP = T(n) - T(n - 1)$ vyplývá, že $CHR = x - 1$.
- **2. fáze:** Kořen charakteristické rovnice:
Pochopitelně platí, že $r = 1$.
- **3. fáze:** Obecné řešení:
 $O\check{R} = Ar^n$, kde A je nějaká hledaná konstanta. Vzhledem k tomu, že $r = 1$, $O\check{R} = A$ (libovolná mocnina čísla 1 se samozřejmě rovná 1). Konstantu A vypočítáme dále.
- **4. fáze:** Hledání partikulárního řešení:
Víme, že $u(n,m) = 1$ (číslo 1 je mnohočlen nultého stupně). Kromě toho je číslo 1 jednoduchým kořenem charakteristické rovnice. Proto:

$$S = n^p c = n \cdot c.$$

Zbývá nám ještě určit konstantu c . Víme, že $P\check{R}$ musí vyhovovat původní rekurzivní rovnici. Po jeho dosazení za $T(n)$ získáme:

$$\begin{aligned}n \cdot c &= 1 + (n - 1)c, \\n \cdot c &= 1 + n \cdot c - c \\c &= 1.\end{aligned}$$

- **5. fáze:** Hledání konečného řešení:
Víme, že konečné řešení rovnice je součtem $O\check{R}$ a $P\check{R}$: $T(n) = RO + RS = A + n \cdot c = A + n$. Konstantu A můžeme snadno vypočítat dosazením elementárního případu:

$$\begin{aligned}T(0) &= 1, \\1 &= A + 0, \\A &= 1.\end{aligned}$$

Po těchto krkolomných výpočtech dostáváme: $T(n) = n + 1$.

Shoduje se s předchozím řešením⁹.

Je zřejmé, že metoda charakteristických rovnic je velmi flexibilní. Umožňuje nám rychle určit algoritrickou složitost i v případě poměrně rozsáhlých programů. Někdy se samozřejmě neobejdeme bez pomocí matematika, ale takové případy se stávají málokdy. Obvykle se týkají rekurzivních programů, které mají jen minimální praktický význam.

Logaritmický rozklad

Z předchozí kapitoly si jistě pamatujeme úlohu, kterou jsme věnovali binárnímu vyhledávání. Jedna z možných verzí příslušné funkce¹⁰ vypadá takto:

```
int binarni_vyhledavani(int *tab, int x, int left, int right)
{
    if (left==right)
        if (t[left]==x)
            return left;
        else // prvek byl nalezen
            return -1; // prvek nebyl nalezen
    else
    {
        int mid=(left+right)/2;
        if (tab[mid]==x)
            return mid; // prvek byl nalezen!
        else
            if (x<tab[mid])
                return binarni_vyhledavani(tab,x,left,mid);
            else
                return binarni_vyhledavani(tab,x,mid,right);
    }
}
```

Jakou výpočetní složitost má tato funkce? Analýzou počtu instrukcí porovnání získáme následující rovnice:

$$T(1) = 1 + 1 = 2$$

$$T(1) = 1 + 1 + T\left(\frac{n}{2}\right) = 2 + T\left(\frac{n}{2}\right)$$

Je jasné, že tato soustava rovnic nevyhovuje výše popsané metodě. Charakteristickou rovnici nemůžeme vyjádřit kvůli dělení n číslem 2. Tento problém sice můžeme překonat například tak, že dosadíme $n = 2p$, ale další postup výpočtu se tím dosti zkomplikuje. Matematici zde však programátorům připravili milou pozornost: v případě zadání tohoto typu lze určit složitost bez komplikovaných výpočtů, protože můžeme uplatnit několik hotových pravidel. Tato matematická pomoc je tím cennější, že podobné úkoly se v programátorské praxi vyskytují velmi často. Než se pustíme do řešení, musíme se však seznámit s několika dalšími matematickými vzorcí. Pak už budeme mít „vyšší“ matematiku opravdu za sebou.

⁹ Jestliže dvě metody vedou ke stejnemu regulérnímu výsledku, můžeme s vysokou jistotou předpokládat, že obě jsou správné.

¹⁰ Trochu se liší od té, kterou jsme navrhli původně.

KAPITOLA 3 Analýza složitosti algoritmů

Předpokládejme, že získáme soustavu rekurzivních rovnic v následujícím obecném tvaru:

$$T(1) = 1$$

$$T(n) = aT\left(\frac{n}{b}\right) + d(n)$$

(Vycházíme z toho, že $n \geq 2$ a dále a a b jsou určité konstanty.)

V závislosti na hodnotách a , b a $d(n)$ dostaneme různá řešení, která shrnuje tabulka 3.3.

Tabulka 3.3: Analýza binárního vyhledávání

Třída algoritmu		
$a > d(b)$	$T(n) \in O(n^{\log_b a})$	
$a < d(b)$	$T(n) \in O(n^{\log_b d(b)})$	jestliže $d(n) = n^a$, pak $T(n) \in O(n^a) = O(d(n))$
$a = d(b)$	$T(n) \in O(n^{\log_b d(b)} \log_b n)$	jestliže $d(n) = n^a$, pak $T(n) \in O(n^a \log_b n)$

Tyto vzorce jsou odvozeny z poměrně komplikovaných výpočtů, které vycházejí z následujících předpokladů:

- n je mocninou b , takže můžeme dosadit $n = b^k$ a převést nelineární rovnici do tvaru $T(b^k) = aT(b^{k-1}) + d(b^k)$. Díky substituci $t_k = T(b^k)$ dále dostaneme lineární rovnici $t_k = at_{k-1} + d(b^k)$ s počáteční podmínkou $t_0 = 1$. Rozbor řešení této rovnice vede ke konečným závěrům, které jsou uvedeny v tabulce 3.3.
- Funkce $d(n)$ musí splňovat tuto podmínsku: $d(xy) = d(x)d(y)$ (např. $d(n) = n^2$ této podmínce vyhovuje, ale $d(n) = n - 1$ už nikoli).

Přes tato omezení se ukazuje, že díky uvedeným vzorcům lze snadno řešit značně rozsáhlou třídu rovnic. Pokusme se například dokončit úlohu na binární vyhledávání. Pamatujeme si, že jsme tehdyn dostali následující rovnice:

$$T(1) = 2$$

$$T(n) = 2 + T\left(\frac{n}{2}\right)$$

Vidíme, že výše uvedené vzory neodpovídají vzorci z naší úlohy. Nic nám však nebrání, abychom jej jednoduchou substitucí převedli do podoby, která nám bude více vyhovovat:

$$U(n) = T(n) - 1 \Leftrightarrow U(1) = T(1) - 1 = 1$$

$$T(n) - 1 = 1 + T\left(\frac{n}{2}\right) \Leftrightarrow U(n) = U\left(\frac{n}{2}\right) + 1$$

Identifikujeme hodnoty konstant: $a = 1$, $b = 2$ a $d(n) = 1$, takže si můžeme všimnout, že nastává třetí případ: $a = d(b)$. Hledaný výsledek tedy vypadá takto:

$$U(n) \in O(n^{\log_2 1} \log_2 n) = O(n^0 \log_2 n) = O(\log_2 n)$$

Změna definičního oboru rekurzivní rovnice

Existuje skupina rekurzivních rovnic, které se vyznačují složitým vzhledem a nijak neodpovídají výše uvedeným vzorcům ani metodám. Občas však stačí provést obyčejnou změnu definičního oboru a řešení se objeví téměř okamžitě. Rozeberme následující příklad:

$$a_n = 3_{n-1}^2 \text{ pro } n \geq 1$$

$$a_0 = 1$$

Rovnice nevyhovuje žádnému schématu, se kterými jsme se dosud setkali. Dosadíme však $b_n = \log_2 a_n$ a zlogaritmujme obě strany rovnice:

$$\log_2 a_n = \log_2 (3_{n-1}^2)$$

takže dostaneme:

$$\begin{cases} b_n = 2b_{n-1} + 3 \log_2 3 \\ b_0 = 0 \end{cases} \text{lineární rovnici}$$

Úlohu v tomto tvaru již umíme vyřešit. Po jednoduchých úpravách získáme:

$$b_n = (2^n - 1) \log_2 3$$

která vede k výsledku

$$a_n = 2^{(2^n-1)\log_2 3} = 3^{(2^n-1)}$$

Ackermannova funkce aneb něco pro labužníky

Kdyby se malé děti trochu orientovaly v informatice, rodiče by je jistě nestrašili čertem, ale Ackermannovou funkcí. Tento příklad dokonale ilustruje, že rekurzivní funkce, které vypadají zdánlivě nevinně, mohou být ve skutečnosti mimořádně náročné. Podívejme se na výpis:

```
A.cpp
int A(int n,int p)
{
    if (n==0)
        return 1;
    if ((p==0)&&(n>=1))
        if (n==1)
            return 2;
        else
            return n+2;

    if ((p>=1)&&(n>=1))
        return A(A(n-1,p),p-1);
}
int main()
{
    cout << "A(3,4)=" << A(3,4) << endl;
}
```

Otázka týkající se tohoto programu zní: Co je příčinou zprávy *Stack overflow!* (přetečení zásobníku) po spuštění programu? Tato zpráva jednoznačně naznačuje, že během činnosti programu došlo k velkému počtu volání Ackermannovy funkce. Už zakrátko ukážeme, kolik těch volání opravdu bylo.

Zběžnou analýzou funkce A dojdeme k následujícímu poznatku:

$$\forall n \geq 1, A(n, 1) = A(A(n-1, 1), 0) = A(n-1, 1) + 2,$$

což okamžitě dává

KAPITOLA 3 Analýza složitosti algoritmů

$$\forall n \geq 1, A(n, 1) = 2n.$$

Analogicky pro číslo 2 dostáváme:

$$\forall n \geq 1, A(n, 2) = A(A(n - 1, 2), 1) = 2A(n - 1, 2),$$

což nám následně umožňuje zapsat, že:

$$\forall n \geq 1, A(n, 2) = 2^n.$$

Ze samotné definice Ackermannovy funkce můžeme usoudit, že:

$$\forall n \geq 1, A(n, 3) = A(A(n - 1, 3), 2) = 2^{A(n - 1, 3)} \text{ a } A(0, 3) = 1.$$

Na základě těchto rovnic můžeme rekurzivně dokázat, že:

$$\forall n \geq 1, A(n, 3) = 2^{2^{\cdot^{\cdot^2}}\!\!\!\}^n}$$

Poněkud horší situace nastává v případě $A(n, 4)$, kde lze jen těžko uvést obecný vzor. Podívejme se na několik číselných příkladů:

$$A(1, 4) = 2$$

$$A(2, 4) = 2^2 = 4$$

$$A(3, 4) = 2^{2^{2^2}} = 65536$$

$$A(4, 4) = 2^{2^{\cdot^{\cdot^2}}\!\!\!\}^{65536}}$$

Výraz v číselné formě $A(4, 4)$ není – diplomaticky řečeno – na první pohled srozumitelný. V případě Ackermannovy funkce je těžké vůbec definovat její třídu – tvrzení, že se chová exponenciálně, může vyznít poněkud ironicky!

Výpočetní náročnost není modla

Možná nás překvapí, že při psaní programů nebo výběru algoritmu „z knihovny“ někdy výpočetní složitost nepředstavuje zásadní rozhodovací kritérium.

Uvedeme několik situací, kdy ji můžeme brát poněkud s rezervou:

- Někdy píšeme algoritmus, který bude počítat výsledek jen několikrát, a nemá proto smysl jej optimalizovat. Uděláme lépe, když se pokusíme program spustit nebo extrapolujeme, jak dlouho bude pracovat, a zhodnotíme, zda je taková verze přijatelná. Tento případ se samozřejmě týká funkcí (procedur), jejichž čas provedení i tak splňuje výkonnostní požadavky aplikace.
- Jednoduchost: optimální algoritmy občas bývají na první pohled zcela nesrozumitelné a kvůli jejich vysoké složitosti se programátoři často dopouštějí logických chyb, které mohou být způsobeny i triviální záměnou při definování mezních či logických podmínek v cyklech.
- Zejména v numerických algoritmech může být důležitější přesnost než třída algoritmu či jeho části.
- Notace velkého O se projevuje hlavně u takových vstupních dat, jejichž zpracování může trvat značně dlouho (nemluvě už o neomezeně dlouhém čase, protože nekonečno je OPRAVDU velké číslo...). V praxi to může znamenat, že „kvadratický“ algoritmus bude v určitých konfiguracích lepší než „logaritmický“!

Obecně bychom se tedy měli řídit zdravým rozumem, který nás varuje před příliš akademickým rozhodováním. Programy totiž nefungují v teoretických prostředích či modelech, ale ve skutečných

počítačích. Nemá smysl stonásobně urychlit čas finančních výpočtů (např. ze 100 ms na 1 ms), když si následný zápis výsledku do databáze i tak vyžádá 2–3 s.

Techniky optimalizace programů

Řekněme, že jsme napsali program a zdá se nám, že běží příliš pomalu¹¹. Jak můžeme zjistit, v čem spočívá výkonnostní problém? Při optimalizaci je klíčové, abychom určili vhodné místo. Pokud je program plně funkční a máme k dispozici reprezentativní data (např. databázi zaplněnou tišíci záznamy, a nikoli jen několika fiktivními položkami), můžeme začít měřit, jak dlouho trvá provedení jednotlivých modulů nebo procedur. Místo, kde budeme měřit, záleží na architektuře. Musíme si přitom uvědomit, kolik dat se přenáší a kde jsou úzká místa.

Mezi jednodušší způsoby patří měření časů provádění procedur pomocí dodatečných čítačů, které do těchto procedur vložíme. Přitom je ale důležité, aby tyto čítače nezatěžovaly samotné algoritmy. Měly by proto používat systémové mechanizmy (odečítat čas ze systému a nezjišťovat jej programově).

Po získání výsledků měření postupujeme podle Paretova pravidla: zaměříme svou pozornost na nejvytíženější nebo nejčastěji volané úseky kódu (např. v cyklech či procedurách).

Poté lze zvolit jeden ze dvou přístupů:

- Analyzujeme algoritmy podle pravidel, s nimiž jsme se seznámili v této kapitole, a pátráme po příslovečných „mouchách“.
- Optimalizujeme kód a hledáme v něm fragmenty, které by bylo možné zjednodušit, například pomocí jedné z těchto metod:
 - Využití mezipaměti (ang. *cache*): výsledek počítáme jen jednou a zapisujeme jej do paměti, která se využívá po celou dobu práce programu. Přitom například pomocí statických proměnných zajišťujeme viditelnost výsledků v rámci celého programu nebo jednotlivých modulů.
 - Eliminujeme časově náročné operace (např. přístupy k databázi) a provádíme je jen tehdy, je-li to opravdu nezbytné.
 - Časově nákladné instrukce násobení nebo umocňování nahrazujeme rychlými operacemi bitového posunu doleva či doprava (např. $n << 3$ znamená totéž jako $n \cdot 2^3 = n \cdot 8$). Dobré komplilátory to samozřejmě dělají automaticky, pokud si ovšem „domyslí“, kde to provést, a mají nastaveny příslušné optimalizační parametry.
 - Část aplikační logiky na zpracování dat přesunujeme na databázový server (odkazujeme na jeho standardní procedury).
 - Nasazujeme multithreading.
 - Využíváme nám známé vlastnosti architektury počítače. Nutíme například komplilátor, aby pracoval s konkrétními optimalizovanými instrukcemi příslušného procesoru.
 - Uplatňujeme vlastnosti programovacího jazyka (např. instrukce *break* nebo *goto* v C++, které sice ztěžují analýzu, ale na druhou stranu jsou efektivní).

Technik optimalizace kódu je skutečně mnoho a na tomto místě je zmiňujeme jen proto, abychom naznačili rozsah problematiky, která nás může při profesionálním programování zajímat.

¹¹ Na tomto místě se nezajímáme o aspekt *velikosti* kódu, protože v současnosti toto kritérium již nemá valný význam.

Úlohy

Úloha 1

Rozhodněte, zda jsou následující rovnice pravdivé či nepravdivé:

- $T(n^3) \in O(n^3)$;
- $T(n^2) \in O(n^3)$;
- $T(2^{n+1}) \in O(2^n)$;
- $T((n+1)!) \in O(n!)$;
- $T(n) \in O(n) \Rightarrow \{T(n)\}^2 \in O(n^2)$;
- Uveďte vlastní příklad.

Úloha 2

V předchozím textu jsme již analyzovali příklad tzv. Fibonacciho posloupnosti. Funkce, která počítá prvky této posloupnosti, není složitá:

```
unsigned long int fib(int x)
{
    if (x<2)
        return x;
    else
        return fib(x-1)+fib(x-2);
}
```

Určete, do které třídy funkce patří.

Úloha 3

Analýzujte některý vlastní program, který obsahuje hodně vnořených cyklů a komplikovaných konstrukcí podobného typu. Nebylo by možné jej nějakým způsobem optimalizovat?

Často se například stává, že v cyklech se inicializují určité proměnné při každém průchodu, ačkoli v praxi by stačilo inicializovat je jen jednou. V tomto případě lze instrukci přiřazení přenést před cyklus, čímž se čas provádění cyklu zkracuje. Pokud vhodně uspořádáme pořadí určitých výpočtů, můžeme obdobně využít mezikvůli určitých bloků instrukcí v dalších blocích – samozřejmě za podmínky, že je jiné fragmenty programu mezičím nepřepíší. Našim úkolem je vypočítat praktickou složitost vlastního programu *před i po* optimalizaci a přesvědčit se, zda se případně podařilo program zrychlit.

Úloha 4

Vyřešte následující rekurzivní rovnici:

$$u_n = u_{n-1} - u_n \cdot u_{n-1} \quad (\text{pro } n \geq 1), \\ u_0 = 1.$$

Řešení a poznámky k úlohám

Úloha 2

Rekurzivní rovnice má tento tvar:

$$\begin{aligned} T(0) &= 0, \\ T(1) &= 1, \\ T(n) &= T(n-1) + T(n-2). \end{aligned}$$

Navzdory poměrně komplikované podobě se v tomto zadání neskrývá žádný chyták a lze jej vyřešit celkem pohodlně. Podívejme se na princip řešení:

1. fáze: Hledání charakteristické rovnice:

Z obecného tvaru LRP: $T(n) = T(n-1) - T(n-2)$ vyplývá, že $CHR = x^2 - x - 1$.

2. fáze: Kořeny charakteristické rovnice:

Po jednoduchých výpočtech dostaváme dva kořeny této kvadratické rovnice:

$$CHR = x^2 - x - 1 = (x - r_1)(x - r_2), \text{ kde:}$$

$$r_1 = \frac{1 + \sqrt{5}}{2} \quad a \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

3. fáze: Obecné řešení:

Z výše vysvětlené teorie lze odvodit, že obecné řešení má tvar $OŘ = Ar_1^n + Br_2^n$ – momentálně je ponechejme v této podobě.

4. fáze: Hledání partikulárního řešení:

Víme, že $u(n,m) = 0$, a tedy $PŘ = 0$.

5. fáze: Hledání konečného řešení:

Hledané řešení je součtem OŘ a PŘ:

$$T(n) = OŘ + PŘ = Ar_1^n + Br_2^n.$$

Zbývá určit záhadné konstanty A a B . Za tímto účelem si pomůžeme počátečními podmínkami (tzn. elementárními případy, abychom vyhověli terminologii z kapitoly 2) soustavy rekurzivních rovnic ($T(0) = 0$ a $T(1) = 1$). Po substituci dostaneme:

$$\begin{aligned} 0 &= A + B, \\ 1 &= Ar_1 + Br_2. \end{aligned}$$

Jedná se o jednoduchou soustavu dvou rovnic se dvěma neznámými (A a B). Vyřešením této soustavy dostaneme hledaný výsledek. Tuto závěrečnou část úlohy můžete dokončit samostatně.

Úloha 4

Uveďme náčrt řešení:

Za předpokladu, že $u_n \neq 0$, můžeme rovnice vydělit výrazem $u_n u_{n-1}$:

$$\frac{1}{u_{n-1}} = \frac{1}{u_n} - 1$$

Dosadíme nyní $v_n = 1/u_n - 1$. Získáme tím velmi jednoduchou rovnici, se kterou jsme se již setkali:

$$\begin{aligned} v_n &= v_{n-1} + 1, \\ v_0 &= 1. \end{aligned}$$

Jejím řešením je samozřejmě $v_n = n + 1$. Po návratu k původnímu definičnímu oboru dostaneme docela překvapivý výsledek: $u_n = 1/(n + 1)$.

KAPITOLA 4

Třídící algoritmy

V této kapitole popíšeme několik známějších metod třídění dat. O významu této problematiky asi nemusíme nikoho přesvědčovat – s úkolem seřadit data se dříve či později setká každý programátor. Třídící algoritmy se ideálně hodí k poznávání algoritmiky a otázek, které souvisejí s výpočetní složitostí programů.

Popisy algoritmů, které představíme v této kapitole, se budou týkat hlavně tzv. *vnitřního třídění*, kdy jsou data uložena výhradně v operační paměti počítače. V tomto vydání knihy jsem se rozhodl probrat také problematiku tzv. *vnějšího třídění*. Vnější třídění se uplatňuje v situacích, do kterých se většina čtenářů ve své programátorské praxi možná nikdy nedostane: tříděných dat je tak mnoho, že je nelze umístit do operační paměti a zpracovat je pomocí jedné z metod vnitřního třídění. Otázky související s problematikou vnějšího třídění jsou však z praktického hlediska natolik zajímavé, že už jejich zběžná analýza může značně rozšířit naše programátorské obzory.

V praxi však operační paměť (označovaná zkratkou RAM) i pevné disky neustále zlevňují a ve většině případů vystačíme s vnitřním tříděním¹. V poslední době se stále častěji mluví o databázích, které jsou kompletně umístěny v paměti (kvůli zvýšení výkonu), což by bylo ještě před několika lety prakticky nepředstavitelné.

Nutnost třídění dat nepřímo souvisí s typickou lidskou touhou hromadit (chceme toho mít hodně) a organizovat (když už toho máme hodně, objevuje se další problém: jak zjistit, co a kde máme). Tento poslední problém lze do značné míry vyřešit právě tříděním.

Při třídění dat se musíme v prvé řadě vyrovnat se značnou rozmanitostí algoritmů tohoto typu. Začínající programátor často nedokáže samostatně zvolit takový třídící algoritmus, který by se pro konkrétní zadání nejlépe hodil. K tématu třídících algoritmů bychom mohli přistupovat tak, že bychom každý algoritmus stručně popsali, uvedli jeho výhody i nevýhody a nabídli subjektivní hodnocení jeho kvality. Podle názoru autora by však taková prezentace neplnila svůj informační účel a pouze by čtenářům zanechala značný zmatek v hlavě. V praxi se totiž ukazuje, že programátoři raději volí vyzkoušená klasická řešení, jako je například třídění vkládáním, bublinkové třídění či metodu Quicksort, než stejně kvalitní (ne-li ještě lepší) řešení, která ovšem slouží spíše jako téma odborných článků nebo příspěvky k výzkumu v oblasti efektivity algoritmů.

V této kapitole:

- Třídění přímým vkládáním, algoritmus třídy $O(N^2)$
- Bublinkové třídění, algoritmus třídy $O(N^2)$
- Quicksort, algoritmus třídy $O(N \log N)$
- HeapSort – třídění haldou
- Slučování setříděných množin
- Třídění slučováním
- Vnější třídění
- Praktické poznámky

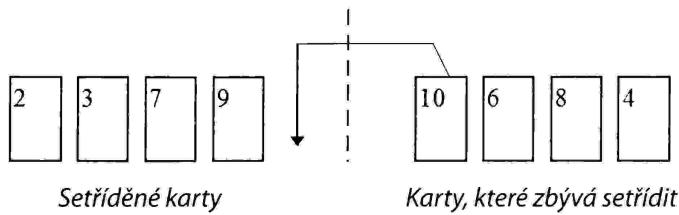
¹ Můj první osobní počítač typu IBM PC XT měl 1 MB paměti RAM a pevný disk s kapacitou 20 MB!

KAPITOLA 4 Třídící algoritmy

Z didaktických důvodů se proto soustředíme na podrobný popis pouze několika dobře známých a přímo ukázkových metod. Tyto algoritmy jsou různě komplikované (z hlediska úsilí na pochopení jejich principu) a vyznačují se různými časovými parametry. Výběr metod je dosti subjektivní a nezbývá než doufat, že vyhoví co největší části čtenářů.

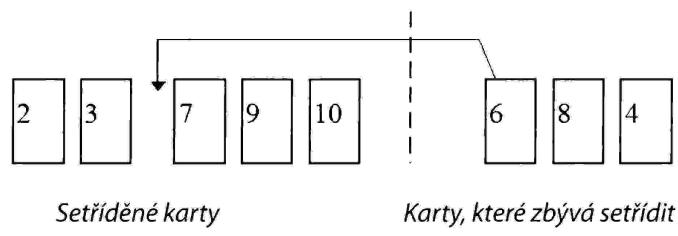
Třídění přímým vkládáním, algoritmus třídy $O(N^2)$

Metodu třídění přímým vkládáním nevědomky používá většina hráčů, když v ruce řadí rozdané karty. Na obrázku 4.1 je znázorněna situace z pohledu hráče. Právě třídí své karty, které dostal značně přeházené:



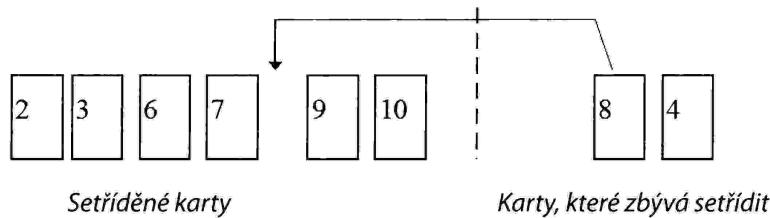
Obrázek 4.1: Třídění přímým vkládáním (1)

Algoritmus je založen na následujícím principu: v danou chvíli držíme v ruce setříděné karty² spolu s kartami, které zbývá setřídit. Při třídění postupujeme tak, že z hromádky nesetříděných karet vezmeme první kartu a vložíme ji na správné místo balíčku již dříve setříděných karet. Podívejme se na dvě následné fáze třídění. Obrázek 4.2 představuje situaci po vložení karty 10 na správné místo. Poté je na řadě karta 6.



Obrázek 4.2: Třídění přímým vkládáním (2)

Po umístění šestky na správné místo dostáváme situaci, kterou vidíme na obrázku 4.3.



Obrázek 4.3: Třídění přímým vkládáním (3)

Na první pohled je jasné, že algoritmus je docela jednotvárný a kromě toho také značně pomalý.

² Na začátku algoritmu sice můžeme mít prázdné ruce, ale v tom případě formálně tvrdíme, že držíme nulový počet karet.

Zajímavá varianta tohoto algoritmu realizuje *vkládání* posunem obsahu pole o jedno místo doprava. Tím vytváří potřebné místo, kam následně vkládá příslušný prvek. Jak máme vědět, zda při hledání volného místa pokračovat v přesunování obsahu pole? Při rozhodování testujeme třídicí podmínku (závisí na tom, zda se jedná o třídění sestupné, vzestupné nebo podle jiných kritérií). Podívejme se na kód programu³:

```
insert.cpp
void InsertSort(int *tab)
{
    for(int i=1; i<n; i++)
    {
        int j=i; // část [0..., i-1] je už setříděná
        int temp=tab[j];
        while ((j>0) && (tab[j-1]>temp))
        {
            tab[j]=tab[j-1];
            j--;
        }
        tab[j]=temp;
    }
}
```

Algoritmus třídění přímým vkládáním se vyznačuje dosti vysokými náklady: patří totiž do třídy $O(N^2)$, takže se v praxi nedá použít k třídění větších polí. Jestliže nám však nezáleží na rychlosti třídění a dáváme přednost krátkému algoritmu, ve kterém určitě neuděláme chybu, pak je tento algoritmus díky své mimořádné jednoduchosti ideální.

Poznámka: Pro zjednodušení příkladů budeme analyzovat jen třídění polí s celými čísly. V praxi se nejčastěji třídí pole nebo seznamy záznamů. Třídicí kritérium se v tomto případě vztahuje na jednu z položek záznamu. (Další informace naleznete také v kapitole 5 a v tématu třídění seznamů.)

Bublinkové třídění, algoritmus třídy O(N²)

Podobně jako algoritmus třídění přímým vkládáním lze také algoritmus bublinkového třídění (třídění přímou výměnou) zapsat velmi snadno. Svůj zajímavý název získal díky přirovnání se vzduchovými bublinkami, které v trubici plné vody putují vzhůru k hladině. Svisle orientované pole hodnot si přitom můžeme představit jako nádobu s vodou a čísla jako pohyblivé bublinky. Nejrychleji se nahoru dostávají ty nejlehčí „bublinky“ – čísla s nejnižší hodnotou (samozřejmě za předpokladu vzestupného třídění). Uvedme kompletní kód programu:

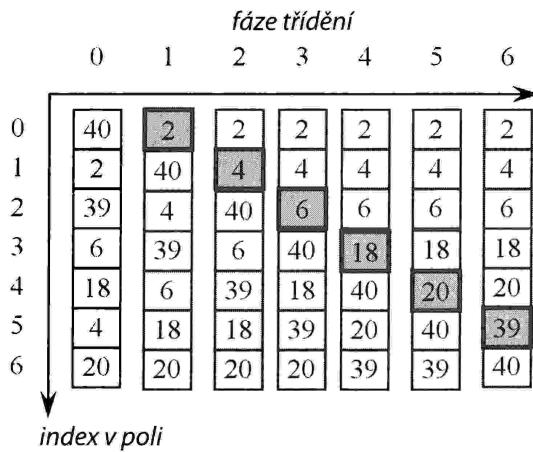
```
bubble.cpp
void bubble(int *tab)
{
    for (int i=1;i<n;i++)
        for (int j=n-1;j>=i;j--)
            if (tab[j]<tab[j-1])
```

³ V tomto a následujících příkladech budeme předpokládat, že kód obsahuje instrukci `const int n=n` jaká hodnota, která definuje rozsah tříděného pole. V přetištěných výpisech ji nebude uvádět.

KAPITOLA 4 Třídící algoritmy

```
{
    // záměna
    int tmp=tab[j-1];
    tab[j-1]=tab[j];
    tab[j]=tmp;
}
}
```

Nyní podrobně rozeberme bublinkové třídění konkrétního pole se sedmi prvky. Prvek, který v příslušném průběhu hlavního cyklu programu „vybublal“ nahoru jako nejlehčí, je na obrázku 4.4 zvýrazněn šedě. Program postupně prochází pole zdola nahoru (cyklus proměnné *i*). Vždy se analyzují dva sousedící prvky (cyklus proměnné *j*): pokud nejsou ve správném pořadí (nahore je „těžší“ prvek), algoritmus je zamění. Při prvním průběhu se na první pozici pole (index 0) dostává „nejlehčí“ prvek, při druhém průběhu se na druhou pozici pole přesune druhý nejmenší prvek (index 1) a stejným způsobem algoritmus pokračuje až do výsledného setřídění pole. Oblast, kterou algoritmus zpracovává, se v každém následném průchodu hlavním cyklem zmenšuje o jeden prvek – samozřejmě by bylo naprosto neefektivní testovat pokaždé celé pole.



Obrázek 4.4: Bublinkové třídění

Již zběžná analýza nám ukáže několik několik nevýhod tohoto algoritmu:

- Poměrně často dochází k „prázdným průchodům“ (nedochází k žádné záměně, protože prvky jsou již setříděné).
- Algoritmus je značně citlivý na konfiguraci dat. Uvedeme příklad dvou velmi podobných polí, z nichž první vyžaduje jen jednu záměnu sousedních prvků, ale v případě druhého pole je potřeba provést až šest záměn:
 - **verze 1:** 4 2 6 18 20 39 40
 - **verze 2:** 4 6 18 20 39 40 2

Nabízí se několik možností, jak tento algoritmus vylepšit. Tyto úpravy sice nezmění jeho třídu (i nadále se jedná o algoritmus třídy $O(N^2)$), ale alespoň jej značně urychlují. Vylepšení spočívají:

- v zapamatování indexu poslední záměny (řešení „prázdných průchodů“),
- v přepínání směrů procházení pole (řešení nevýhodných konfigurací dat).

Takto upravený algoritmus bublinkového třídění se označuje jako *třídění přetřásáním* (ang. *shaker-sort*). Dále uvádíme úplný výpis kódu, tentokrát však již bez podrobnějšího vysvětlení:

```

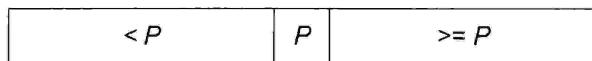
shaker.cpp
void ShakerSort(int *tab)
{
    int left=1, right=n-1, k=n-1;
    do
    {
        for(int j=right; j>=left; j--)
            if(tab[j-1]>tab[j])
            {
                swap(tab[j-1],tab[j]);
                k=j;
            }
        left=k+1;
        for(int j=left; j<=right; j++)
            if(tab[j-1]>tab[j])
            {
                swap(tab[j-1],tab[j]);
                k=j;
            }
        right=k-1;
    }
    while (left<=right);
}

```

Quicksort, algoritmus třídy $O(N \log N)$

Tento slavný algoritmus⁴ se také označuje jako rychlé třídění. Patří mezi řešení, která díky vhodné dekompozici problému dosahují značného zvýšení rychlosti třídění. Procedura třídění se dělí na dvě hlavní části: část určená k samotnému třídění, která prakticky jen vyvolává sama sebe, a procedury dělení prvků pole podle hodnoty určité buňky pole, která slouží jako dělící osa (označuje se anglickým slovem *pivot*). Proces třídění zajišťuje druhá uvedená procedura, zatímco díky rekurzivnímu volání první procedury dochází ke složení částečných výsledků a v důsledku i setřídění celého pole.

Jak přesně funguje procedura dělení? V první fázi zjišťuje hodnotu pivotu P , což obvykle bývá první prvek analyzované oblasti pole. Stejná oblast pole je následně rozdělena tak, aby se prvky menší než prvek „ P “ nacházely po jeho levé straně a větší vpravo od něj⁵. Nové uspořádání je symbolicky znázorněno na obrázku 4.5.



Obrázek 4.5: Rozdělení pole v metodě Quicksort

V další fázi se procedura Quicksort aplikuje na levou i pravou část pole, což zajistí jeho setřídění. To je vše!

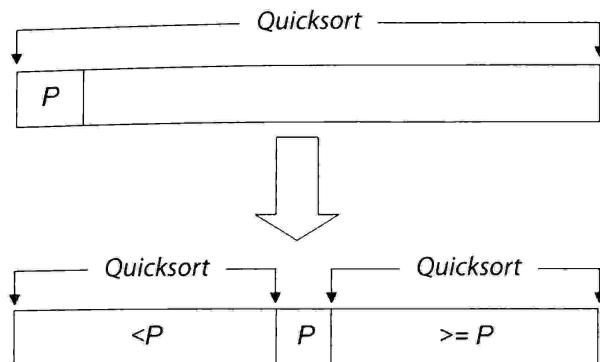
Obrázek 4.6 symbolicky představuje dvě hlavní etapy třídění metodou Quicksort (symbol P tradičně označuje buňku pole, která slouží jako pivot).

⁴ Viz C.A.R. Hoare – „Quicksort“ v časopise Computer Journal, 5, 1(1962).

⁵ V případě potřeby se prvky pole fyzicky přemisťují.

KAPITOLA 4 Třídicí algoritmy

Na první pohled vidíme, že rekurzivní volání se zastavuje v okamžiku, kdy má oblast pole jednotkovou velikost – tehdy už není co třídit.



Obrázek 4.6: Princip fungování procedury Quicksort

Metoda třídění, kterou jsme právě vysvětlili, je založena na velice prostém principu. Tato vlastnost se dokonale projevuje na kompaktním zápisu samotné procedury (skutečný kód je uveden dále):

```

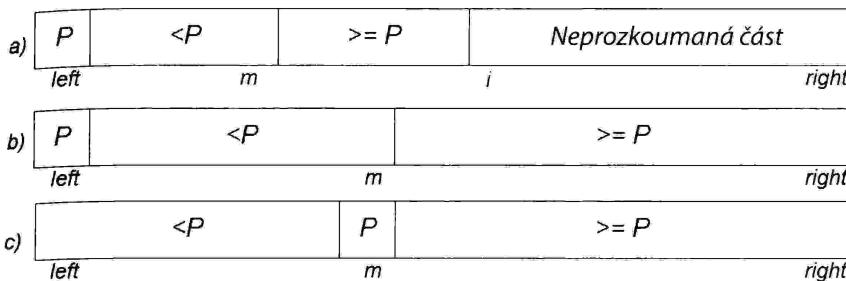
void Quicksort(int *tab, int left, int right)
{
    if (left < right)
    {
        int m;
        // Dělení pole podle pivotu "P"
        // ("P" může být např. první prvek pole, čili tab[left])
        // Po rozdělení je prvek "P" přenesen do buňky
        // s indexem "m"
        Quicksort(tab, left, m-1);
        Quicksort(tab, m+1, right);
    }
}
    
```

Jak nejrychleji realizovat část procedury, která se skromně ukrývá za komentářem? Má přece v algoritmu klíčovou roli, ale zatím jsme ji popisovali značně obecně. Tento přístup lze snadno zdůvodnit: algoritmus *Quicksort* má mnoho různých implementací, které se liší právě tím, jakým způsobem realizují proceduru dělení pole podle hodnoty pivotu.

Abychom se nemuseli zamýšlet nad tím, která verze je správná, uvedeme nyní – v souladu se zásadami současné demokracie – tu nejhodnější.

Při výběru se řídíme těmito kritérii: elegance, rychlosť a jednoduchost – těmito vlastnostmi se nepochyběně vyznačuje řešení, které nabízí [Ben92].

Princip je založen na chování poměrně jednoduchého invariantu v aktuálně dělené části pole (viz obrázek 4.7).

**Obrázek 4.7:** Určení invariantu pro algoritmus Quicksort

Legenda:

- $left$ – levý krajní index aktuální části pole
- $right$ – pravý krajní index aktuální části pole
- P – hodnota pivotu (obvykle je to $tab[left]$)
- i – index, který se „přemisťuje“ po indexech pole od $left$ do $right$
- m – hledaný index buňky pole, kam umístíme pivot

Přemisťování umožňuje uspořádat prvky pole takovým způsobem, aby se po levé straně prvku m nacházely hodnoty menší než pivot a po pravé straně hodnoty stejné nebo větší (obrázek 4.7a). Kvůli tomu během přemisťování indexu i kontrolujeme pravdivost podmínky $tab[i] > P$. Pokud podmínka neplatí, obnovujeme správné pořadí pomocí inkrementace a výměny hodnot $tab[m]$ a $tab[i]$. Po dokončení posledního průchodu polem (obrázek 4.7b), kdy hledáme buňky, které dosud nevyhovovaly invariantu, vede záměna hodnot $tab[left]$ a $tab[m]$ k žádanému stavu (obrázek 4.7c).

Nyní nám už nic nebrání, abychom představili hotovou verzi procedury Quicksort. Všechny fáze činnosti algoritmu, které jsme dosud popsali, jsou zde zkombinovány do jediné procedury:

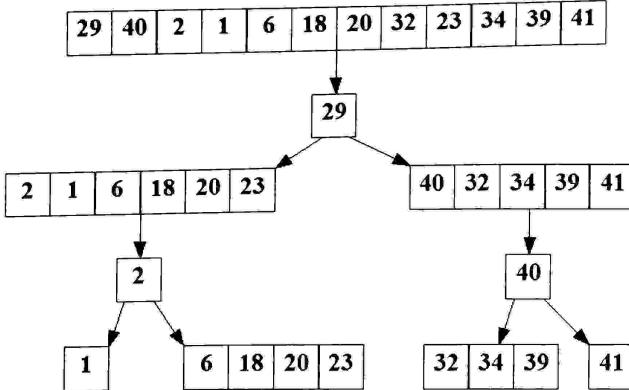
```
qsrt.cpp
void qsrt(int *tab, int left, int right)
{
    if (left < right)
    {
        int m=left;
        for(int i=left+1;i<=right;i++)
            if (tab[i]<tab[left])
                swap(tab[++m],tab[i]);
        swap(tab[left],tab[m]);
        qsrt(tab,left,m-1);
        qsrt(tab,m+1,right);
    }
}
```

Abychom dobře pochopili, jak algoritmus funguje, zkusme pomocí něj ručně setřídit malé pole, například s těmito čísly:

29, 40, 2, 1, 6, 18, 20, 32, 23, 34, 39, 41

Výsledek činnosti těch exemplářů procedury Quicksort, v nichž skutečně dochází k nějakým změnám, shrnuje obrázek 4.8.

KAPITOLA 4 Třídící algoritmy



Obrázek 4.8: Příklad třídění metodou Quicksort

Jasně vidíme, že když procházíme od levé krajní větve stromu k pravé krajní větvi a přitom nejdříve navštěvujeme jeho levé dílčí větve, postupujeme vlastně po setříděném poli! V programu je tento postup založen na rekurzivním volání procedury `qsort`. Algoritmus *Quicksort* představuje dobrý příklad programovací techniky zvané „rozděl a panuj“, kterou podrobně vysvětlíme v kapitole 9.

Prozatím prozradíme, že se jedná o takovou dekompozici problému, která urychlí činnost programu (a při té příležitosti zjednoduší řešenou úlohu). Algoritmus *Quicksort* oba tyto požadavky splňuje dokonale.

HeapSort – třídění haldou

Algoritmus třídění haldou (tento podivný výraz ihned vysvětlíme) patří do třídy $O(N \log N)$. Využívá datovou strukturu s názvem halda (ang. heap), která se vyznačuje zajímavými vlastnostmi. Výhodou tohoto algoritmu je právě uvedená datová struktura, která zároveň funguje jako prioritní fronta (viz kapitolu 5).

Nyní se tedy s touto datovou strukturou seznámíme, i když tím poněkud předběhneme téma následující kapitoly, která je věnována právě datovým strukturám. V případě terminologických nejasností lze zběžně prolistovat následující kapitolu, kde je tato struktura popsána podrobněji. V uvedené kapitole také představíme jinou verzi stejného algoritmu v objektové notaci. V této kapitole ukážeme stručnější verzi stejného algoritmu, která se místo datové struktury zaměřuje spíše na samotný princip třídění.

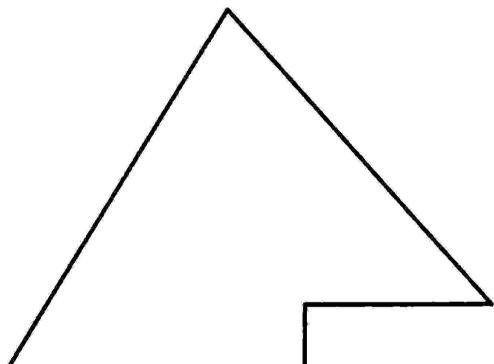
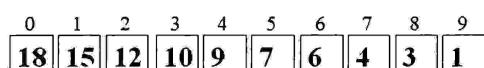
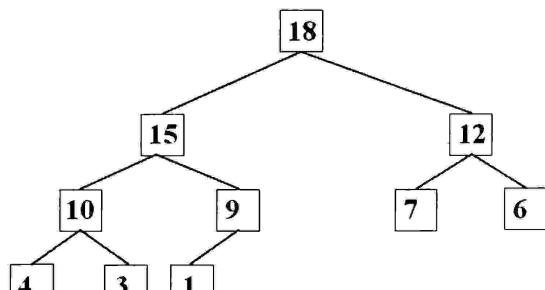
Halda je binární strom obsahující čísla nebo libovolné jiné prvky z množiny, kterou lze uspořádat. Halda nabývá tvaru, který je znázorněn na obrázku 4.9 – jakési pyramidy, jejíž podstava je na pravé straně „nakousnutá“.

Vedle tvaru je klíčovou vlastností haldy její uspořádanost – žádný podřízený uzel nemá vyšší hodnotu než jemu nadřazený uzel. Tuto podmínu lze v syntaxi pole zapsat následovně: $T[\text{predek}(i)] \geq T[i]$.

Obsah haldy můžeme snadno reprezentovat pomocí běžného pole T . Uplatňujeme přitom tato pravidla:

- Vrcholek haldy uložíme na pozici $T[0]$.
- Pro libovolný uzel (existuje-li) v $T[i]$ platí, že jeho levý potomek se nachází na pozici $T[2i+1]$ a pravý na pozici $T[2i+2]$.

Příklad takové struktury najdeme na obrázku 4.10.

**Obrázek 4.9:** Halda a její vlastnosti (1)**Obrázek 4.10:** Halda a její vlastnosti (2)

Algoritmus třídění haldou lze popsat takto:

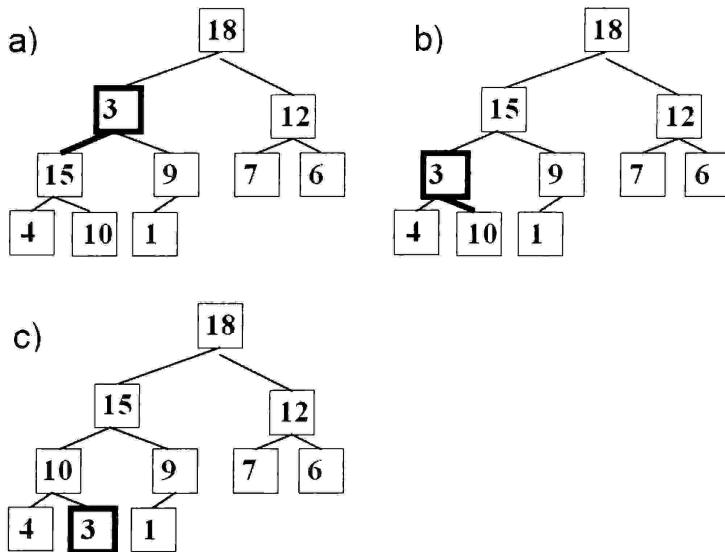
1. Vložíme data do haldy, která bude mít formu pole délky `délka`.
2. Z haldy odstraníme vrcholek (je to největší prvek) tak, že jej zaměníme s posledním listem stromu (`délka--`).
3. Obnovíme vlastnost haldy pro zbývající části haldy (bez odstraněného prvku).
4. Přejdeme k bodu 2.

Proceduru z bodu 3 můžeme implementovat následujícím způsobem:

1. Pokud je vrcholek větší než oba potomci, procedura končí.
2. Zaměníme vrchol s větším potomkem.
3. Obnovíme vlastnost haldy v té její části, kde došlo k záměně.

Na obrázku 4.11 je znázorněn výsledek činnosti procedury z bodu 3 aplikované na zvýrazněný uzel, který porušuje podmínu $T[\text{předek}(i)] \geq T[i]$. V dalších krocích a), b) a c) pomocí záměny vrcholku s největším potomkem obnovíme uspořádaný stav struktury.

KAPITOLA 4 Třídící algoritmy



Obrázek 4.11: Příklad obnovení vlastnosti haldy

Dále je uvedena jednoduchá realizace algoritmu třídění haldou.

```
heap.cpp
void obnov(int T[], int k, int n)
{
    int i,j;

    i = T[k-1];
    while (k <= n/2)
    {
        j=2*k;
        if((j<n) && ( T[j-1]<T[j] ) ) j++;
        if (i >= T[j-1])
            break;
        else
        {
            T[k-1] = T[j-1];
            k=j;
        }
    }
    T[k-1]=i;
}

// třídění pole T
void heapsort(int T[], int n)
{
    int k, swap;
    for(k=n/2; k>0; k--) obnov(T, k, n);
    do
    {
```

```

swap=T[0];
T[0]=T[n-1];
T[n-1]=swap;
n--;
obnov(T, 1, n);
} while (n > 1);
}

int main()
{
    int i, T[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
    for (i=0;i<14;i++)
        cout << " " << T[i];

    cout << endl;

    heapsort(T,14);

    for (i=0;i<14;i++)
        cout << " " << T[i];
}

```

Slučování setříděných množin

Jednoduchý algoritmus, který popíšeme v dalším textu, slouží k třídění v poměrně specifickém případě: vstupní pole $T_1[N]$ a $T_2[M]$ jsou uspořádaná a máme za úkol vytvořit pole $T_3[N + M]$, které bude rovněž setříděné.

Samotný algoritmus vypadá prostě: musí z obou polí načítat data tak dlouho, dokud je „nevycerpaný“, a přitom do výsledného pole vždy přenášet nejmenší prvek (třídíme-li vzestupně). Algoritmus „neztrácí čas“ hledáním prvků, které by bylo potřeba porovnat. Předpokládáme totiž, že vstupní data jsou v praxi již seřazena. Tato podmínka je velmi důležitá, protože se na žádném místě následující procedury již neověruje.

Podívejme se na vlastní program:

```

scalaj.cpp
void slucuj(int T1[], int T2[], int T3[])
{ // T1, T2 - vstupní pole velikosti M a M, T3 - výsledné pole
    for (int i=0, j=0, k=0; k < N+M; k++)
    {
        if (i==N) // dosažen konec množiny T1, kopírování zbytku
        {
            T3[k]=T2[j++]; continue;
        }

        if (j==M) // dosažen konec množiny T2, kopírování zbytku
        {
            T3[k]=T1[i++]; continue;
        }
        if (T1[i]<T2[j])
            T3[k]=T1[i++];
        else
            T3[k]=T2[j++];
    }
}

```

```
    else
        T3[k]=T2[j++];
    }
}
```

Všimněme si, že program k datům přistupuje sekvenčním způsobem. To připomíná práci se soubory, které se obvykle načítají právě takto.

Jako malé cvičení zkuste kód procedury, která zpracovává pole, přepracovat na odpovídající verzi, která bude obsluhovat soubory. Není to tak triviální, protože nelze postupovat úplně analogicky jako v případě polí (musíte počítat s tím, že předem neznáte velikost vstupních souborů). Máte-li potíže s použitím funkcí, které operují na souborech, můžete nahlédnout do přílohy A, kde najdete program na čtení dat ze souboru.

Třídění slučováním

Nyní popíšeme metodu třídění, která – podobně jako algoritmus *Quicksort* – patří k rekurzivním metodám typu „rozděl a panuj“ (podrobněji je rozebereme v kapitole 9). Algoritmus není složitý:

- Rozdělíme n -prvkovou posloupnost na dvě dílčí posloupnosti po $n/2$ prvcích.
- Každou dílčí posloupnost třídíme metodou slučování.
- Setříděné dílčí posloupnosti spojíme do jediné setříděné posloupnosti.

Celá metoda se samozřejmě opírá o vhodně napsanou slučovací proceduru. Uvedme jednu z možných implementací takové procedury:

```
merge.cpp
const int N = 10;
void slucuj(int T[], int p, int mid, int k)
// p - počátek, k - konec, mid - střed
// procedura spojuje 2 setříděná pole T[p...mid] a T[mid+1...k]
{
    int T2[N]; // pomocné pole
    int p1 = p, k1 = mid; // dílčí pole 1
    int p2 = mid+1, k2 = k; // dílčí pole 2
    // až do vyčerpání obsahu polí pokračuje slučování
    // s využitím pomocného pole
    int i = p1;
    while((p1 <= k1) && (p2 <= k2))
    {
        if(T[p1] < T[p2])
        {
            T2[i] = T[p1];
            p1++;
        }
        else
        {
            T2[i] = T[p2];
            p2++;
        }
        i++;
    }
}
```

```

while(p1 <= k1)
{
    T2[i] = T[p1];
    p1++;
    i++;
}
while(p2 <= k2)
{
    T2[i] = T[p2];
    p2++;
    i++;
}
// kopírování z dočasného do původního pole
for(i = p; i <= k; i++)
    T[i] = T2[i];
}

```

Samotná třídicí procedura pochopitelně dodržuje výše uvedené rekurzivní schéma:

```

void MergeSort(int T[], int p, int k)
{
    if(p < k)
    {
        int mid = (p + k) / 2; // střed
        MergeSort(T, p, mid); // třídění levé poloviny
        MergeSort(T, mid+1, k); // třídění pravé poloviny
        slucuj(T, p, mid, k); // slučování
    }
}
int main()
{
    int T[N] = {4, 6, 4, 12, -3, 6, -6, 1, 8, '2'};
    cout << "Před setříděním:\n";
    for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
    MergeSort(T, 0, N-1);
    cout << "Po setřídění:\n";
    for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
}

```

Výsledky:

```

Před setříděním:
4 6 4 12 -3 6 -6 1 8 50
Po setřídění:
-6 -3 1 4 4 6 6 8 12 50

```

Vnější třídění

Vnější třídění se uplatňuje tehdy, když rozměr dat značně přesahuje kapacitu paměti. Může taková situace v praxi vůbec nastat? Samozřejmě, ale ihned je potřeba upozornit, že informační systémy, kde se s těmito případy můžeme setkat, jsou dosti specifické: jedná se například o velké účetní či bankovní systémy. Využije však některá banka či telekomunikační firma informace z této části kapitoly? Asi nikoho nepřekvapí, že nikoli:

KAPITOLA 4 Třídicí algoritmy

- Velké informační systémy totiž odhadnou využívají n -vrstevnou architekturu, která zahrnuje databázový server (nebo jejich farmu). V tomto případě lze požadavky na třídění dat přenést na databázový modul. Ten může v datových tabulkách vytvářet tzv. indexy, které dovolují prohlížet data v setříděném tvaru.
- Některé systémy (například bankovní systémy založené na počítačích mainframe) využívají programovací jazyk COBOL, který vznikl již v 60. letech. Slouží k programování velkých podnikových aplikací a poskytuje efektivní funkce pro práci se soubory včetně jejich třídění.

Navzdory omezenému praktickému významu je teoretický model vnějšího třídění dosud zajímavý a stojí za zábavnou prohlídku, třeba jen proto, abychom se seznámili s problémy, jaké musí někteří programátoři řešit.

Jak si můžeme všimnout, samotné vnější třídění lze považovat za jistou programátorskou *techniku*, protože k samotné operaci třídění (která probíhá v operační paměti) se využívají klasické rychlé algoritmy vnitřního třídění. Hlavní nároky jsou však kladený na algoritmy, které efektivně manipulují s pracovními soubory.

Model vnějšího třídění musí vyhovovat následujícím podmínkám a omezením:

- Existuje značný nepoměr mezi velikostí operační paměti počítače (která je *omezená*) a množinou tříděných dat (teoreticky *neomezená* nebo obecně značně rozsáhlá).
- Máme ztížený přístup k prvkům, které potřebujeme třídit, protože se nacházejí např. v souborech, které musíme načítat pomocí mechanizmů operačního systému – obvykle sekvenčně nebo v nejlepším případě blokově.
- Čas činnosti algoritmu, který data třídí v paměti, je zanedbatelný v porovnání s tím, jak dlouho trvá čtení dat z vnějších souborů a zápis do nich.
- Kvůli fyzickým omezením struktury disků je výhodnější data ze vstupního souboru načítat ve větších blocích (stránkách) paměti. Obsah diskového souboru lze proto považovat za určitý druh seznamu, který se skládá z jednotlivých bloků.
- K třídění fragmentů souboru, které se vejdu do paměti, slouží řada vstupních vyrovnávacích pamětí. Jedná se o rezervovanou část operační paměti, kam operační systém umisťuje načtené bloky dat nebo odkud načítá blok dat, který má zapsat na disk.

Máme-li k dispozici takové obslužné funkce, dokážeme načítat a zapisovat soubory a navíc disponujeme rychlým klasickým algoritmem vnitřního třídění, můžeme navrhnut například metodu třídění, která spočívá v *rozdělení a sloučení v kombinaci s tříděním*:

- Rozdělíme velký vstupní soubor na menší dílčí soubory, které se snadno vejdu do operační paměti.
- Setřídíme dílčí soubory známou efektivní metodou, jako je např. Quicksort, a zapíšeme je již v setříděném podobě. V tomto bodu je v praxi nutné sekvenčně zapsat část vstupního souboru do paměti. Po překročení jisté předem určené velikosti setřídíme načtený obsah v operační paměti a výsledek zapíšeme do dalšího dílčího souboru. Takto postupujeme, dokud není vstupní soubor kompletně rozdělen na menší, již setříděné soubory.
- V dalších fázích (průbězích) slučujeme tyto menší a setříděné soubory na větší, dokud není setříděn celý vstupní soubor.

Jak může fungovat operace slučování setříděných souborů? Není tak náročná jako vlastní třídění, protože vstupní data jsou již uspořádaná. Jednoduchý algoritmus na slučování setříděných dat v rostoucím pořadí by mohl vypadat takto:

- Otevřeme *vstupní soubory*.
- Sekvenčně načítáme data z obou souborů a přenášíme menší hodnotu do *výstupního souboru* až do okamžiku, kdy se dostaneme na konec některého ze souborů.
- Na konec výstupního souboru zapíšeme zbytek druhého souboru.

Stručný popis algoritmu samozřejmě vůbec nezaručuje krátký program v jazyce C++. Korektní implementace, která bude kontrolovat konce souborů a zajistí řadu dalších porovnání, si může vyžádat i několik desítek řádků kódu. Určitou alternativu, která zdaleka není tak nesmyslná, představuje spojení souborů dohromady a setřídění jejich obsahu pomocí příkazů operačního systému, k čemuž slouží například následující výpis.

```
systemsrt.cpp
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream soubor_vst1 ("input1.txt"); // vstupní soubor 1
    ifstream soubor_vst2 ("input2.txt"); // vstupní soubor 2
    ofstream soubor_VYS ("output.txt"); // výsledný soubor
    string s; // spojení souborů pomocí jazyka C++
    while (getline(soubor_vst1,s))
        soubor_VYS << s << endl;
    while (getline(soubor_vst2,s))
        soubor_VYS << s << endl;
    soubor_VYS.close(); // uvolnění výsledného souboru
    // Soubory lze samozřejmě spojit i systémovým
    // příkazem: system("copy input1.txt+input2.txt output.txt");
    system("sort output.txt /0 output.txt"); // třídění výsledného souboru
    // pomocí příkazu operačního systému
}
```

Je zřejmé, jaké jsou nevýhody systémového třídění:

- Ztrácíme kontrolu nad efektivitou (proces třídění může mít tak nízkou prioritu, že proběhne velmi pomalu).
- Program v jazyce C++ už není přenositelný (má-li např. předchozí kód fungovat v systému Linux, musíme upravit syntaxi příkazu sort a místo příkazu copy uvést cp).
- Systémové třídění standardně nerozlišuje čísla a znaky. Pokud by náš soubor obsahoval čísla, mohlo by se ukázat, že hodnota „111“ je menší než „22“.

Jak tedy setřídit soubory s řadami čísel, které jsou však zapsány jako znaky (v textovém editoru)? V systému Unix (Linux) můžeme příkaz sort přinutit, aby interpretoval číselné hodnoty (parametrem -n). Jak ale postupovat v systému DOS (Windows)?

Můžeme využít následující řešení: Zkopírujeme obsah diskových souborů například do jednoho pole celých čísel, setřídíme je některou z metod, které již známe, a setříděné pole znova zapíšeme do souboru.

KAPITOLA 4 Třídící algoritmy

Jediný problém se může objevit při konverzi z textového formátu na celá čísla. V jazyce C++ k tomu lze využít jednoduchou instrukci:

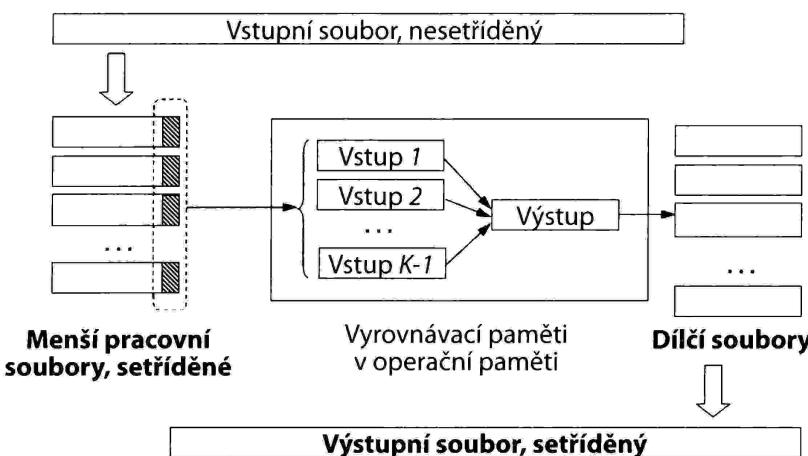
```
int nějaká_proměnná_int = atoi(nějaká_proměnná_string.c_str());
```

Místo toho, abychom v programu porovnávali načtené znaky (typ `string`), převedeme údaje na celočíselné proměnné, které již můžeme dosadit do instrukcí porovnání.

Je zřejmé, že vnější třídění souvisí více s konkrétními aspekty systému (soubory, konverze dat) než s čistě algoritmickými otázkami. Vráťme se však na chvíli k teoretickému modelu.

Na obrázku 4.12 je znázorněno jisté zobecněné schéma systému vnějšího třídění. Předpokládáme, že vstupní soubor má N stránek a k jeho třídění můžeme kromě výkonu procesoru využít vyhrazenou oblast paměti:

- $K-1$ vstupních vyrovnávacích pamětí, které umožňují zpracovat data ze vstupních souborů.
- 1 výstupní vyrovnávací paměť, která je určena k zápisu do vnějších souborů (pracovních či výsledných).



Obrázek 4.12: Obecný model vnějšího třídění

Celková velikost těchto K vyrovnávacích pamětí pochopitelně nemůže překročit dostupnou kapacitu operační paměti.

Proces třídění bude samozřejmě vyžadovat více průběhů, protože v jediném průběhu lze zpracovat jen část velkého vstupního souboru. V prvním „nultém“ průběhu je potřeba rozdělit vstupní soubor na menší pracovní soubory a setřídit je. Velikost tohoto „menšího“ souboru závisí na dostupné operační paměti – čím větší soubor, tím lépe.

Následně můžeme do vyhrazených vyrovnávacích pamětí v operační paměti načítat počáteční fragmenty setříděných souborů (jsou setříděny po „nultém“ průběhu) – viz levou část obrázku, třídit je v paměti počítače slučováním (přitom využijeme vyrovnávacích pamětí znázorněných v prostřední části obrázku) a zapisovat do částečných výsledných souborů, které vidíme na pravé straně obrázku. (Reálný operační systém může vyhrazené soubory na pravé straně schématu tvořit dynamicky nebo může nové částečné výsledky znova zapisovat do předchozích souborů.)

V procesu třídění slučováním s použitím vyrovnávacích pamětí a zápisu do jednoho či více výsledných souboru pokračujeme až do chvíle, kdy soubory na levé straně schématu ani samotné vyrovnávací paměti neobsahují žádná data. Jak jsme již zmínili, náklady na třídění tímto algoritmem

souvisejí s počtem provedených průběhů, protože operace se soubory jsou časově nejnáročnější. Vzhledem k tomu, že úzké hrdlo představuje samotné dělení souboru na N bloků a počet použitých vyrovnávacích pamětí K v operační paměti, můžeme náklady vnějšího třídění vyčíslit takto:

$$2N \left\lceil 1 + \log_{K-1} \left\lceil \frac{N}{K} \right\rceil \right\rceil$$

Pravý člen tohoto součinu v praxi představuje počet průběhů. Hodnoty se přitom zaokrouhlují na nejbližší vyšší celé číslo (symbol $\lceil \rceil$).

Praktické poznámky

Kritéria výběru třídicích algoritmů lze shrnout do několika snadno zapamatovatelných pravidel:

- K třídění malého počtu prvků nemá smysl používat nejfektivnější algoritmy (např. *Quicksort*), protože časová úspora bude zanedbatelná.
- Část třídicích algoritmů, která je známa z informatických publikací, se v praxi vůbec nepoužívá (nebo jen výjimečně). Tyto algoritmy vznikly v rámci akademického výzkumu a při programování se nikdy neuplatnily. Budeme-li se držet dobré známých metod, máme také větší jistotu, že se nedopustíme nějaké programátorské chyby.

Při psaní kódu je také vhodné důsledně kontrolovat, zda standardní knihovny používaného komplátora již neobsahují implementovanou třídicí funkci. Například v mnoha komplátorech⁶ existuje hotová funkce s názvem `qsort`, která má tuto hlavičku:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *))
```

Lze třídit libovolné pole (typ `void`), ale musíme přesně uvést jeho rozměr: počet prvků `nmemb` velelkosti `size`. Funkce `qsort` navíc jako parametr vyžaduje *ukazatel na porovnávací funkci*. Princip ukazatelů na funkce podrobně rozebereme v části o jednosměrných seznamech. Na tomto místě pro ilustraci uvedeme jen hotový kód na třídění pole celých čísel (chceme-li třídit pole jiného typu než `int`, stačí pouze upravit porovnávací funkci `comp` a způsob volání funkce `qsort`):

```
qsort2.cpp
int comp(const void *x, const void *y)
{
    int xx=*(int*)x; // explicitní konverze z typu
    int yy=*(int*)y; // 'void*' na 'int*'
    if( xx < yy)
        return -1;
    if( xx == yy)
        return 0;
    else
        return 1;
}
const int n=12;
int tab[n]={40,29,2,1,6,18,20,32,34,39,23,41};

int main()
```

⁶ Například: Borland C++ Builder, GNU C++, Visual C++.

KAPITOLA 4 Třídící algoritmy

```
{  
    for (int i=0; i<n; i++)  
        cout << tab[i] << " ";  
    cout << endl;  
    qsort(tab, n, sizeof(int), comp);  
    for (int i=0; i<n; i++)  
        cout << tab[i] << " ";  
    cout << endl;  
}
```

Porovnávací funkce `comp` závisí na typu dat tříděného pole. Například u pole ukazatelů na znakové řetězce bychom ji použili takto:

```
int comp(const void* a, const void* b)  
{  
    return(strcmp((char*)a, (char*)b));  
}  
int main()  
{  
    char s[5][4]={"aaa", "ccc", "ddd", "zzz", "fff"};  
    qsort((void*)s, 5, sizeof(s[0]), comp);  
    for (int i=0; i<5; i++)  
        cout << s[i] << endl;  
}
```

Nevýhoda hotové knihovní funkce spočívá v tom, že nemáme přístup k jejímu zdrojovému kódu: dostáváme zajíce v pytli a musíme si na něj zvyknout.

Napíšeme-li místo toho vlastní třídící proceduru, můžeme ji optimalizovat z hlediska své vlastní aplikace. K určitému zlepšení časových parametrů stačí, když funkci `comp` zahrneme do třídící procedury, i když třída algoritmu se tím nezmění.

KAPITOLA 5

Datové typy a struktury

V této kapitole:

- Základní a složené typy
- Znakové řetězce a texty v jazyce C++
- Abstraktní datové struktury
- Úlohy
- Řešení úloh

O významu tématu, kterým se budeme zabývat v této kapitole, asi není potřeba nikoho přesvědčovat. Na výběru momentálně nevhodnější datové struktury může záviset rychlosť programu, rozsah obsluhovaných základních případů, snadnost úprav kódu, čitelnost zápisu algoritmů, přenositelnost kódu a druhotně i spokojenost programátora.

Základní a složené typy

Každý, kdo se učil libovolný programovací jazyk, musel zvládnout principy používání tzv. základních (integrovaných) typů.

Například v jazyce C++ jsou k dispozici mj. následující typy:

- `int` a `long` (celá čísla – např. deklarace `int proměnná_typu_int=5;` umožňuje ukládat do proměnné `proměnná_typu_int` celá čísla, v tomto případě hodnotu 5),
- `float` a `double` (čísla s plovoucí desetinnou čárkou),
- `char` (znaky, např. `char znak='a'`),
- `bool` (umožňuje deklarovat logické proměnné, které nabývají hodnot `true` a `false`),
- typy ukazatelové¹ (s „hvězdičkou“), které umožňují uchovávat adresy – např. deklarace `int *pr;` znamená, že proměnná `pr` bude odkazovat na proměnnou celočíselného typu. „Ukazatel“ uchovává adresu. Chceme-li z něj dostat odkazovanou hodnotu, musíme jej uvést symbolem hvězdičky (např. `pr` obsahuje adresu a `*pr` hodnotu – samozřejmě za předpokladu, že jsme proměnnou `pr` dříve inicializovali, třeba pomocí instrukce `pr=&proměnná_typu_int`). Operátor `&` zjišťuje adresu proměnné. Je zřejmé, že se jedná o opak operátoru `*`).

Základní typy představují stavební kameny, z nichž můžeme budovat složitější datové struktury. Měli bychom je dobré zvládnout, abychom věděli, že symbol `*` neoznačuje jen násobení. Každý základní typ navíc jednoznačně určuje množinu povolených hodnot (obor) a operací (např. sčítání, násobení atd.).

1 V příloze A jsou také vysvětleny tzv. odkazy, které se používají poněkud snáze než ukazatele, ačkoli v praxi je nemohou zastoupit vždy.

KAPITOLA 5 Datové typy a struktury

Některé datové typy jazyka C++ mají objektový charakter (mj. `string` či `vector`). Díky tomu můžeme snadno manipulovat jejich obsahem (např. sčítat objekty) a nemusíme řešit technické aspekty, jako je přidělování a uvolňování paměti.



Upozornění: Seznam základních typů spolu s jejich přesností je součástí standardu jazyka C++, ale konkrétní *implementace* těchto typů (např. počet bitů obsazených proměnnou určitého typu) závisí na použité technologii. Jestliže tedy ve svých programech pracujete s nízkoúrovňovými vlastnostmi hardwaru, jako je rozměr paměťové buňky, musíte počítat s tím, že ztrácejí přenositelnost. Minimální a maximální hodnoty konkrétních datových typů jsou uvedeny v souborech `limits.h` a `float.h`.

Pro matematické algoritmy má mimořádný význam rozsah přípustných hodnot. Například v souboru `limits.h` jazyka Visual C++ najdeme tyto informace:

```
#define INT_MIN (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX 2147483647 /* maximum (signed) int value */
```

Pokud tedy proměnnou `int` používáme v určitém algoritmu, jehož výsledky se do povoleného rozsahu teoreticky nemusí vejít, musíme buď změnit typ proměnné, nebo si pomocí nějakým programátorským trikem, jaké uvádíme třeba v kapitole 13 v části „Kódování dat a aritmetika velkých čísel“.

Ani začínajícím programátörům v C++ nedělá žádné problémy používat poněkud složitější datové struktury, jako jsou pole a záznamy. Pole a záznamy jsou natolik jednoduché, že se jimi nebudeme podrobněji zabývat. (Pokud však syntaxi jazyka C++ dostatečně nerozumíte, naleznete konkrétní příklady v příloze A.) Měli bychom však uvést obecné vlastnosti těchto „poněkud složitějších“ datových struktur, abychom si uvědomili, jaké mají výhody a nevýhody:

- Pole (např. `int t[N]`) umožňuje v souvislé struktuře uchovávat množinu N proměnných stejného typu, které lze adresovat pomocí indexu od 0 do N-1 (`t[0], t[1], ... t[N-1]`). K datům lze tedy přistupovat přirozeně a neomezeně: známe-li index konkrétní hodnoty, můžeme ji přímo adresovat. Nevhoda klasických polí samozřejmě spočívá v tom, že je pro ně nutné rezervovat určitou pevnou část paměti. Pole v jazyce C++ mohou mít více rozměrů a kromě proměnných základního typu obsahovat i datové záznamy.
- Záznamy (v jazyce C++ definované klíčovým slovem `struct`) představují nový základní typ, který umožňuje v jakémusi „balíčku“ seskupit několik proměnných jiného typu. Díky tomu je možné snadno sdružit například informace o adrese, platu a rodném čísle (jsou to tzv. položky) na jednom místě. K položkám záznamů můžeme opět přistupovat velice přirozeně (pomocí tečkové notace – rozšířené příklady kódu jazyka C++ jsou uvedeny v příloze A). Do položek lze umístit i složené typy (např. pole) či ukazatele.



Poznámka: Do jazyka C++ patří rovněž klíčové slovo `union`. Tento pozůstatek z jazyka C umožňuje v určitých specifických situacích ušetřit paměť definováním tzv. sjednocení, ale za cenu nižší čitelnosti výsledného kódu. Definice sjednocení na první pohled připomíná definici záznamu, ale to je jen zdání: položky sjednocení následují v paměti bezprostředně za sebou a programátor musí důsledně kontrolovat, co z paměti skutečně načítá. S ohledem na čitelnost kódu uděláme lépe, když se sjednocení vyhneme, protože tato konstrukce jazyka je značně matoucí a ve většině případů se bez ní můžeme obejít.

Znakové řetězce a texty v jazyce C++

Zařazení tohoto tématu může někoho překvapit, protože úzce souvisí s jazykem C++ a v příručkách algoritmiky je zmiňováno málokdy (výjimkou je [Sed92]). Poučení o textech a znakových řetězcích má rozhodně smysl. Umíte-li totiž manipulovat s texty v programech, dokážete snáze vyřešit mnohé úlohy. Bez správného zpracování textů nelze napsat praktickou komerční aplikaci. V tomto vydání knihy jsem se proto rozhodl představit několik příkladů, které by měly vysvětlit tajemné konstrukce, s nimiž se v kódu jazyka C++ můžeme setkat.

Nejdříve si musíme uvědomit, že znakový řetězec v jazyce C++ je pole, které má proměnnou délku. Tento zdánlivý detail má ze systémového hlediska značný význam: běžné pole je v programu od začátku definováno staticky, má známou délku a víme, jak číst jednotlivé jeho prvky (např. znaky, čísla, záznamy). Znakový řetězec se teoreticky chová dosti podobně, ale vzhledem k tomu, že se jeho délka může během činnosti programu měnit, potřebuje komplikátor dodatečnou informaci – značku konce řetězce, což je speciální nulový kód (zapisuje se jako '\0').

V jazyce C++ můžeme často narazit na deklarace tohoto tvaru:

```
char *s1;
char s2[100];
char *s3="Kód chyby 100";
```

V prvním případě je deklarován ukazatel, který lze případně připsat nějaké textové proměnné (např. v argumentech volání funkce main, jiné textové proměnné), ale sám o sobě *nevyhrazuje paměť* na uchovávání znaků. Teprve druhá forma umožňuje bezpečně ukládat znaky a volně přistupovat ke každému z nich pomocí indexování, aniž bychom porušovali pravidla bezpečného čtení z paměti. Třetí forma má stejný význam, jako bychom deklarovali ukazatel s3, přidělili místo v paměti na 14 znaků (text + značka konce, tedy 0) a iniciovali obsah textem „Kód chyby 100“.

Podívejme se na jednoduchý ukázkový program, který přijímá znakový řetězec, zapisuje jej do pole a poté načítá jeho obsah po jednotlivých znacích spolu s automaticky vloženou značkou konce.

```
znaki.cpp
#include <string.h>
#include <iostream>
using namespace std;

int main()
{
    char s[100]; // zatím tady nic není!
    cout << "Zadejte slovo:" ;
    cin >> s;
    for (int i=0; i<=strlen(s); i++) // strlen = délka řetězce
        cout << "Znak [" << i << "]=" << s[i] << ",kód: "
            <<(int) s[i] << endl;
}
```

Příklad spuštění programu:

```
Zadejte slovo:pes
Znak [0]=p,kód: 112
Znak [1]=e,kód: 101
Znak [2]=s,kód: 115
Znak [3]= ,kód: 0
```

KAPITOLA 5 Datové typy a struktury

Kdybychom v tomto kódu nahradili výraz `char s[100]` výrazem `char *s`, program by oznámil chybu přístupu k paměti.

Nutnost testování délky znakových řetězců, kontrolování eventuálního přeplnění „vyrovnávacích pamětí“ a rezervování paměti dlouho představovaly noční můru mnoha programátorů v C++ a vedly k mnoha omylům, které se projevovaly teprve po spuštění programu. V aktualizované verzi standardu jazyka C++ se proto objevil výhodný objektový datový typ `s` názvem `string`, který umožňuje snadno zpracovávat znakové řetězce bez únavného používání polí a ukazatelů a celé té otravné kontroly paměti.

Podívejme se na jednoduchou ukázkou použití objektů třídy `string`.

```
string.cpp
int main()
{
    string s1, s2="ma maso";           // deklarace + inicializace
    s1 = "Ema ";                      // přiřazení hodnoty
    string s3 = s1 + s2 + "\n";        // spojování řetězců
    cout << "s3=" << s3;
    s3.erase();                        // nulování znakového řetězce
    cout << "s3=" << s3;
}
```

Program po spuštění zobrazí:

```
s3=Ema ma maso
-----
s3=
s2: znak [0]=m,kód: 109
s2: znak [1]=a,kód: 97
s2: znak [2]= ,kód: 32
s2: znak [3]=m,kód: 109
s2: znak [4]=a,kód: 97
s2: znak [5]=s,kód: 115
s2: znak [6]=o,kód: 111
```

Tento jednoduchý výpis dokládá, jak snadno a přirozeně lze znakové řetězce vytvářet a upravovat. Třída `string` samozřejmě kromě metody `erase` nabízí mnoho jiných zajímavých metod a díky pře definovaným (tzn. přetíženým) standardním operátorům je možné znakové řetězce přiřazovat a přirozeně spojovat. Kromě toho nám pochopitelně zůstává plný přístup k obsahu řetězců, jako by se jednalo o klasické pole znaků.

Poznámka: V této knize budeme podle momentálních potřeb uplatňovat klasický přístup (založený na polích) i objektový přístup s využitím třídy `string`. Chceme-li v jazyce C++ programovat efektivně, musíme bohužel znát obě tyto metody. Dosud totiž vzniklo mnoho vynikajících programů (často přenesených z jazyka C), které vycházejí hlavně z klasického přístupu.

Abstraktní datové struktury

Z programátorského hlediska jsou zajímavější například seznamy, binární stromy, grafy a podobné datové struktury. Tyto struktury značně rozšiřují možnosti programového řešení mnoha

zajímavých úloh a obohacují škálu potenciálních aplikací informatiky. Vzhledem ke specifickým pravidlům používání se uvedené struktury často označují jako *abstraktní*. V jazyce C++ to vede k objektovému programování, které se dokonale hodí k implementaci takových útvarů, které přirozeně (čti: v komplátorech) neexistují.

Datové struktury programátorem mimořádně usnadňují práci, protože dovolují uspořádat informace uložené v počítači způsobem, který je srozumitelnější pro člověka. Jak ukážeme v další části knihy, kromě formy uspořádání dat jsou datové struktury také zajímavým nástrojem k řešení složitých algoritmických problémů, např.:

- *Seznamy* usnadňují tvorbu flexibilních databází.
- *Binární stromy* mohou sloužit k symbolické analýze aritmetických výrazů.
- *Grafy* zjednodušují řešení mnoha úloh z oblasti tzv. umělé inteligence.

Seznamy a binárními stromy se budeme zabývat v této kapitole. Materiál týkající se grafů pak vzhledem ke svému významu a rozsahu tvoří samostatnou kapitolu 10.

V následujících podkapitolách představíme nejdůležitější datové struktury a metody jejich zpracování. Doplňkové příklady na ilustraci použití těchto struktur jsou zvoleny tak, aby naznačily, ve kterých oblastech je lze prakticky aplikovat.

Tato kapitola obsahuje hodně ukázkového kódu C++, který může být pro začínající adepty tohoto jazyka poměrně náročný (zahrnuje např. šablony tříd, ukazatele na funkce, spřátelené funkce, přetížení operátorů...). Součástí moderních implementací jazyka C++ jsou bohaté kolekce hotových tříd, které realizují dokonce i složité datové struktury. Proto není nutné, abychom uměli od nuly naprogramovat vše, co je již v použitelné formě dostupné. Měli bychom však alespoň porozumět tomu, jakým způsobem se poněkud komplikovanější třídy navrhují. Tím získáme zběhlost při používání konkrétních programovacích technik, kterými se jazyk C++ vyznačuje. Tato kapitola tedy poslouží jak čtenářům, které zajímají samotné datové struktury, tak i programátorům, kteří chtějí pouze „opsat“ nějaké hotové řešení.

Jednosměrné seznamy

Jednosměrný seznam je paměťově úsporná datová struktura, která umožňuje seskupit libovolný počet prvků (čísel, znaků, záznamů atd.). Jediné omezení představuje jen dostupná kapacita paměti. To je velká výhoda oproti polím, jejichž rozměr lze sice určovat dynamicky, ale pokus o přidělení velkého „lineárního“ bloku paměti při činnosti programu se pokaždé nemusí podařit. Snadno si dokážeme představit, že operační systém mnohem pravděpodobněji bez problémů 50 000 krát přidělí paměť na jednu datovou položku (velikosti řekněme 1 kB), než aby v jednom kroku rezervoval místo na pole velikosti 50 000 kB. Navíc pole (alespoň zpočátku) téměř nebude využité.

Při práci se seznamem je nutné rezervovat v paměti místo pro jisté dodatečné informace ukazatelů. Tato režie však není velká a většinou ji lze zanedbat, zejména v programech na zpracování rozsáhlých obchodních dat.

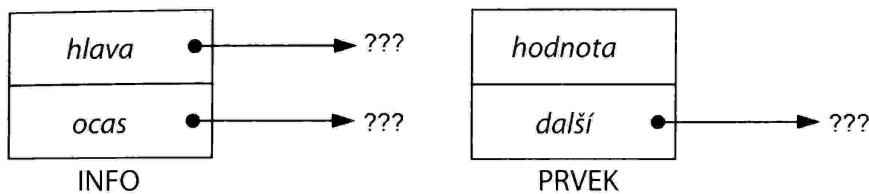
Při tvorbě jednosměrného seznamu se používají dva typy paměťových buněk (obrázek 5.1)²:

- Záznam informační povahy (označení `INFO`), který obsahuje dva ukazatele: *na začátek* seznamu a *na konec* seznamu. Ukazatel obsahuje adresu buňky v paměti počítače a k jeho uložení slouží proměnná ukazatelového typu.

² Výrazná tečka na obrázku označuje hodnotu typu „ukazatel“.

KAPITOLA 5 Datové typy a struktury

- Záznam pracovní povahy (označení PRVEK). Obsahuje položku *hodnota* (zde je uvedena informační hodnota, např. číslo, znakový řetězec, množina atributů atd.) a ukazatel na následující prvek seznamu.

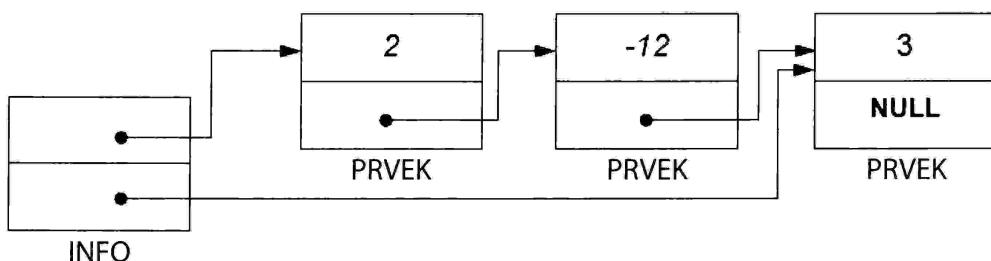


Obrázek 5.1: Typy záznamů používané při programování seznamů

V popisech datových struktur typu seznamu se obvykle nezmíňuje informační záznam (asi proto, že nepatří mezi prvky samotné datové struktury). To je ovšem zásadní chyba. Za cenu několika bajtů paměti³ totiž získáme stálý přístup k mnoha důležitým operacím a mimořádně si usnadníme základní operaci: připojení nového prvku na konec seznamu (pokud prvek nechceme vložit na konec, můžeme jej vždy připojit na začátek seznamu, ale tehdy ztrácíme přehled o pořadí připojování prvků).

Položky *hlava*, *ocas* a *dalsi* obsahují ukazatele⁴, zatímco v položce *hodnota* se mohou nacházet libovolná data – čísla, znaky, záznamy atd. V příkladech na stránkách této knihy se pro jednoduchost používají hlavně hodnoty celočíselného typu. Obecnost výkladu se tím však nesnižuje. Případné úpravy takto zjednodušených algoritmů mají spíše kosmetický než zásadní charakter.

Princip lze popsát takto: Pokud je seznam prázdný, informační struktura obsahuje dva ukazatele NULL. Měli bychom si pamatovat, že hodnota NULL se v obecném případě nerovná nule – jedná se o jistou adresu, na kterou určitě neodkazuje žádná proměnná (taková je hlavní idea ukazatele NULL, ale hodně programátorů si to bohužel neuvědomuje). První prvek seznamu zahrnuje svou vlastní hodnotu (informaci k uložení) spolu s ukazatelem na druhý prvek seznamu. Druhý prvek je tvořen vlastní informační položkou a samozřejmě také ukazatelem na třetí prvek seznamu atd. Místo ukončení seznamu označujeme speciální hodnotou – ukazatelem NULL. Podívejme se nyní na obrázek 5.2, který znázorňuje seznam sestávající ze tří prvků: 2, -12, 3.



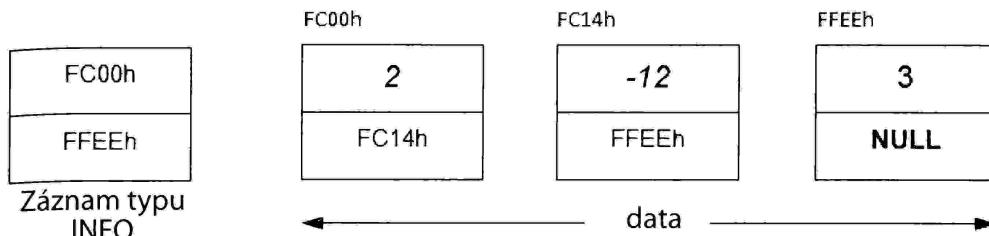
Obrázek 5.2: Příklad jednosměrného seznamu (1)

Obrázek 5.3 přesně odpovídá předchozímu obrázku, s tou výjimkou, že místo šipek symbolizujících „odkazování“ jsou uvedeny konkrétní číselné hodnoty adres paměťových buněk. Šestnáct-

³ Velikost ukazatelové proměnné závisí na potřebách adresování používaného modelu paměti a zásadně závisí na operačním systému.

⁴ Princip odkazování dále symbolizujeme pomocí šipek.

kové číslo umístěné nad záznamem představuje adresu v operační paměti počítače, kde záznamu přidělila místo standardní procedura new⁵.



Obrázek 5.3: Příklad jednosměrného seznamu (2)

Vraťme se ještě k analýze záznamů, které tvoří seznam. Položka `hľava` informační struktury odkazuje na buňku s hodnotou 2 – jedná se o první prvek seznamu. Srozumitelněji řečeno: položka obsahuje adresu, na které je v paměti počítače uložen záznam.

Položka `ocas` informační struktury odkazuje na buňku, která uchovává hodnotu 3 (poslední prvek seznamu). Tyto položky umožňují prohlížet prvky seznamu a připojovat nové. Ukažme si, jak může vypadat procedura na prohlížení prvků seznamu, například při hledání hodnoty `x` (informační buňka se nazývá `info`):

```
adresa_tmp=info.hľava
dokud (adresa_tmp != NULL) proved{
{
    jestliže(adresa_tmp.hodnota == x) pak
    {
        Vypiš "Hledaný prvek byl nalezen"
        ukonči proceduru
    }
    v opačném případě
        adresa_tmp=adresa_tmp.dalsi
}
vypiš "Hledaný prvek nebyl nalezen"
```

V další části kapitoly budeme kombinovat popisy algoritmů v programovacím pseudojazyce (výše uvedeného typu) s hotovým kódem jazyka C++. Při výběru jedné z možností se budeme řídit kritériem čitelnosti procedur. Samozřejmě i v případech, kdy algoritmus představíme na pseudo-kódu, najdeme v archivu ke stažení plné verze jazyka C++, které lze zkompilovat a spustit.

Realizace datových struktur jednosměrného seznamu

Následující implementace struktur, které jsou potřebné k programové obsluze jednosměrného seznamu, přesně odráží obrázek 5.2, takže nás nečekají větší překvapení. Čtenáři, kteří zatím dostačně neovládají syntaxi jazyka C++, by si měli dobré zapamatovat, jakým způsobem se deklarují rekurzivní datové typy (tzn. typy, které obsahují ukazatele na prvky svého typu). Zápis se poněkud liší od toho, který se používá například v jazyce Pascal (viz také přílohu A).

⁵ Z historického hlediska bychom mohli připomenout, že v klasickém jazyce C bylo nutné přidělovat paměť pomocí knihovních funkcí `calloc` a `malloc`. V jazyce C++ úplně totéž zajišťuje instrukce `new`, která je však mnohem čitelnější a patří již přímo do programovacího jazyka.

KAPITOLA 5 Datové typy a struktury

```
lista.h
// definice datových typů
enum hledani {NEUSPECH=0, USPECH=1};
typedef struct rob
{
    int hodnota;
    struct rob *další; // ukazatel na následující prvek
}PRVEK;

// začátek deklarace třídy SEZNAM

class SEZNAM
{
public:
    // hlavičky:
    friend SEZNAM& operator +(SEZNAM&,SEZNAM&); // sčítá dva seznamy
    friend void slouceni(SEZNAM &x,SEZNAM &y);
    void vypis(); // vypíše obsah seznamu
    int hledej(int x); // hledá v seznamu prvek x
    void vloz1(int x); // přidá prvek bez třídění
    void vloz2(int x); // přidá prvek s tříděním
    SEZNAM& operator --(int); // odstraní poslední prvek seznamu

    // několik jednoduchých metod:
    bool prazdne(); // je seznam prázdný?
    {
        return (inf.hlava==NULL);
    }
    void nuluj(); // nuluje seznam bez odstranění
    {
        inf.hlava=inf.ocas=NULL;
    }

    SEZNAM() // konstruktor
    {
        inf.hlava=inf.ocas=NULL;
    }
    ~SEZNAM() // destruktor, který používá pře definovaný operátor --
    {
        while (!prazdne()) (*this)--;
    }

private:
    // informační struktura zajistí přístup k seznamu
    typedef struct
    {
        PRVEK *hlava;
        PRVEK *ocas;
    }INFO;
    INFO inf;
}; // konec deklarace třídy SEZNAM
```

Pole hodnota v našem příkladu má typ `int`, ale v praxi to může být značně komplikovaný informační záznam (který např. obsahuje jméno, příjmení, věk atd.).

Několik jednoduchých metod třídy `SEZNAM` jsme definovali již v hlavičkovém souboru. Pokud je metoda triviální a lze ji zapsat stručně, mnohdy ji definujeme přímo v těle třídy. Jako příklad lze uvést miniaturní obslužnou funkci `prazdny`. Ačkoli je velmi jednoduchá, je poměrně pravděpodobné, že se bude v praxi používat často.

Třída `SEZNAM` sice není příliš propracovaná, ale zahrnuje několik řešení, která vyžadují podrobnější komentář. Čtenářům, kteří jazyk C++ tak dobře neznají, možná není úplně jasná deklarace:

```
SEZNAM& operator --(int);
```

Je to samozřejmě hlavička metody, která předefinuje operátor `--`. K čemu však slouží ten dodatečný parametr typu `int`? Vyžadují jej pravidla jazyka C++. Tento parametr je umělý a pouze informuje kompilátor o tom, že v této metodě předefinujeme „přírůstkový“ operátor.

Máme ještě na výběr ohledně případného skrytí datových typů (viz deklarace `public` a `private`). Programátor se přitom musí vhodně rozhodnout na základě různých aspektů: *smyslu* zveřejnění či skrytí atributů, „výkonnostních“ parametrů metod, pozdějšího dědění atd. Kódy uvedené v této kapitole v žádném případě nemají sloužit jako vzorová řešení. Vzhledem k prakticky nekonečnému počtu nových situací a problémů, s nimiž se může programátor v praxi setkat, taková řešení v zásadě ani nelze najít. Snažíme se spíše ukázat různé programátorské možnosti a nevnucovat jediné řešení při současném zavrhování jiných.

V následujících odstavcích představíme všechny metody, u nichž jsme zatím viděli jen hlavičky.

Vytvoření jednosměrného seznamu

Rozhodně je nejvyšší čas, abychom ukázali, jakým způsobem lze k seznamu připojit další prvky. Pomůžeme si přitom několika více či méně složitými funkcemi. S nutností kontroly, zda jsou jisté prvky již v seznamu přítomny, se setkáme například ve funkci `vloz1`, která k seznamu připojuje nový prvek.

Během přidávání nového prvku můžeme zvolit jeden ze dvou přístupů: buď budeme seznam počítat za běžnou strukturu k ukládání neuspořádaných dat (tak budeme vědecky zkoumat, jak se zvětšuje neporádek), nebo se rozhodneme, že budeme nové prvky do seznamu ukládat v určeném pořadí – například je ihned setřídíme do neklesající posloupnosti.

Následuje procedura `vloz1`, která odpovídá triviálnímu prvnímu případu:

```
lista.cpp
// Část souboru lista.cpp, zatím bez funkce main:
#include <iostream>
using namespace std;
#include "LISTA.H"           // bez této direktivy by kompilátor nerozuměl
                             // novému datovému typu SEZNAM
void SEZNAM::vloz1(int x)   // připojí záznam na konec seznamu
{
    PRVEK *q=new PRVEK;      // bez třídění ; operator :: je
    q->hodnota=x;           // nezbytný, protože definujeme kód
    q->dalsi=NULL;          // metody mimo "tělo" třídy
    if (prazdne())           // je seznam prázdný?
        inf.hlava=inf.ocas=q;
```

KAPITOLA 5 Datové typy a struktury

```

else
    // seznam není prázdný
{
    (inf.ocas)->dalsi=q;      // položka dalsi je
    inf.ocas=q;                // ukazatelem: proto -> a nikoli . (tečka)
}
}

```

Funkce *vloz1* se chová takto: V případě prázdného seznamu jsou obě pole informační struktury iniciována ukazatelem na nově vytvořený prvek. Jinak je nový prvek připojen na konec, a stává se tedy ocasem seznamu.

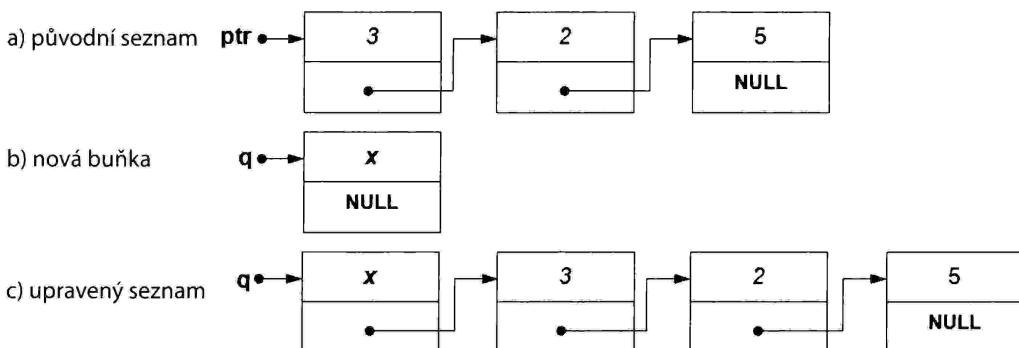
Nový prvek můžeme pochopitelně vložit i před *první* prvek seznamu (na který vždy směřuje jistý snadno dostupný ukazatel, řekněme *ptr*). Nový prvek by pak automaticky přebíral roli hlavy seznamu a program by si jej musel pamatovat, aby neztratil přístup k datům:

```

PRVEK *q=new PRVEK;      // a)
q->hodnota=x;          // b)
q->dalsi=ptr;          // c)

```

Tento kód lze ilustrovat schématem na obrázku 5.4.



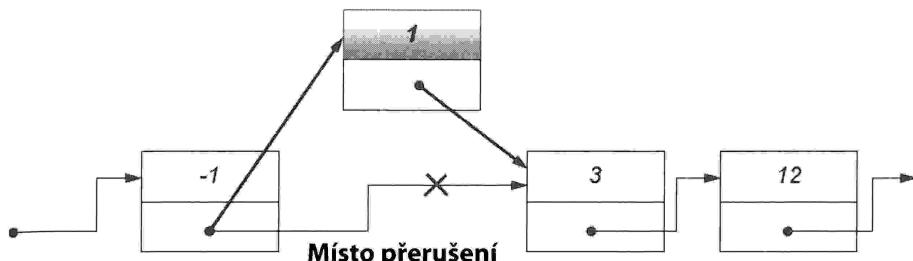
Obrázek 5.4: Připojení nového prvku seznamu na jeho začátek



Upozornění: V tomto a dalších příkladech předpokládáme, že žádost o přidělení paměti je VŽDY úspěšná. V reálných programech je takový předpoklad dosti nebezpečný a doporučuje se kontrolovat, zda po použití instrukce s klíčovým slovem *new*, např. *PRVEK *q=new PRVEK*, nebyla proměnné *q* přiřazena hodnota *NULL*. S ohledem na lepší srozumitelnost prezentovaných algoritmů budeme kontrolu tohoto typu z uváděných kódů vynechávat. Při praktickém programování se však takové opomenutí může značně vymstít.

Výše uvedený způsob je sice správný, ale musíme si uvědomit, že vkládáme-li nové prvky vždy na začátek seznamu, ztrácíme tím informaci o *následnosti* prvků, která občas bývá důležitá.

Značně komplikovanější je taková funkce, která nový prvek vkládá na takové místo, aby celý seznam zůstal setříděný (zde jako neklesající posloupnost). Koncepci znázorňuje obrázek 5.5, kde můžeme sledovat vkládání čísla 1 do existujícího seznamu, který obsahuje čísla -1, 3 a 12.



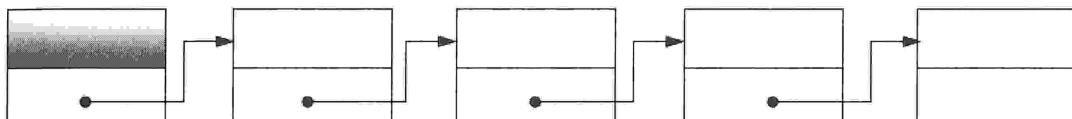
Obrázek 5.5: Vložení prvku do seznamu a setřídění

Nový prvek (zvýrazněný tučným písmem a stínováním) lze umístit na začátek (a) či konec (b) seznamu nebo také někam doprostřed seznamu (c). V každém z těchto případů musíme v existujícím seznamu najít místo, kam nový prvek vložit. Přitom je nutné zapsat dva ukazatele: na prvek, *před který* vkládáme novou buňku, a prvek, *za kterým* ji vkládáme. K zapamatování těchto důležitých informací slouží proměnné *pred* a *po*.

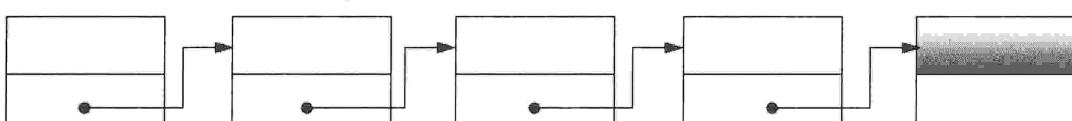
Když poté zjistíme vhodnou polohu, můžeme vložení nového prvku do seznamu dokončit. Přitom postupujeme v závislosti na místě vkládání a na tom, zda seznam náhodou ještě není prázdný. Stručně řečeno: realizace je bohužel dosti komplikovaná. Větší složitost funkce *vloz2* je zčásti dána tím, že funkce sdružuje hledání místa pro vložení prvku se samotným připojením prvku do seznamu. Stejně dobře bychom obě činnosti mohli rozdělit mezi samostatné funkce, avšak v publikované verzi jsme zvolili kombinovaný přístup.

Existují tři varianty místa vložení nového prvku do seznamu, které symbolicky představuje obrázek 5.6 (předpokládáme, že seznam už obsahuje nějaké prvky).

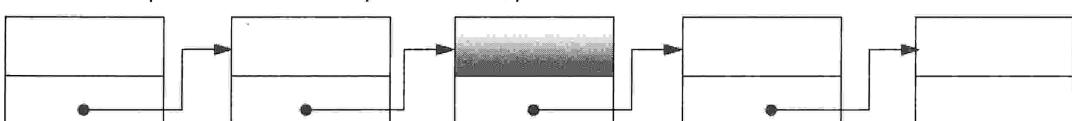
a) Vložení na začátek seznamu (*pred* = NULL)



b) Vložení na konec seznamu (*po* = NULL)



c) Vložení doprostřed seznamu (*pred* ≠ NULL, *po* ≠ NULL)



Obrázek 5.6: Vkládání nového prvku do seznamu – analýza případů

Při vkládání nového prvku do seznamu postupujeme podle toho, který případ platí. Uvedeme nyní úplný kód funkce *vloz2*, která je založena právě na principu znázorněném na obrázku 5.6:

```
void SEZNAM::vloz2(int x)      // připojení záznamu na správné místo
{
    PRVEK *q=new PRVEK;        // s tříděním
                                // vytvoření nového prvku seznamu
```

KAPITOLA 5 Datové typy a struktury

```
q->hodnota=x;
// Hledání vhodné pozice k vložení prvku
if (prazdne())
{
    inf.hlava=inf.ocas=q;
    q->dalsi=NULL;
}
else // hledání místa k vložení
{
    PRVEK *pred=NULL,*po=inf.hlava;
    // výčtová proměnná
    enum {HLEDEJ,KONEC} stav=HLEDEJ;
    while ((stav==HLEDEJ) && (po!=NULL))
        if (po->hodnota>=x)
            stav=KONEC;           // nalezeno vhodné místo
        else                  // pokračuje hledání
        {
            // vhodného místa
            pred=po;             // ukazatele "pred" a "po"
            po=po->dalsi;        // uchovávají místo vložení
        }
    if (pred==NULL)           // vložení na začátek seznamu
    {
        inf.hlava=q;
        q->dalsi=po;
    }
    else
        if (po==NULL)         // vložení na konec seznamu
        {
            inf.ocas->dalsi=q;
            q->dalsi=NULL;
            inf.ocas=q;
        }
        else                  // vložení někam doprostřed
        {
            pred->dalsi=q;
            q->dalsi=po;
        }
    }
}
```

Následující důležité (i když velice prosté) metody vycházejí prakticky ze stejného principu. Při hledání určitého prvku x procházíme seznam pomocí běžného cyklu while:

```
int SEZNAM::hledej(int x)
{
    PRVEK *q=inf.hlava;
    while (q != NULL)
    {
        if (q->hodnota==x)
            return USPECH;
        q=q->dalsi;
    }
    return NEUSPECH;
}
```

Stejnou strukturu má i metoda `vypis()`, která vypisuje obsah seznamu:

```
void SEZNAM::vypis()
{
    PRVEK *q=inf.hlava;
    if (prazdne()) cout << "(seznam je prázdný)";
    else
        while (q != NULL)
        {
            cout << q->hodnota << " ";
            q=q->dalsi;
        }
    cout << "\n";
}
```

Je načase přejít k poněkud těžším částem kódu.

Začneme od operace odstraňování *posledního* prvku seznamu. Využíváme přitom předefinovaný operátor dekrementace `--`. Jak jsme již dříve uvedli, dodatečný parametr typu `int` vyžadují pravidla jazyka C++; je umělý a pouze informuje komplilátor o tom, že v této metodě předefinujeme „*přírůstkový*“ operátor.

Funkce, která se za ním skrývá, je poměrně jednoduchá: pokud seznam obsahuje pouze jeden prvek, funkce upraví položky `hlava` i `ocas` informační struktury. Po odstranění jediného prvku seznamu jsou obě tyto položky inicializovány hodnotou `NULL`.

Poněkud složitější případ nastává tehdy, když seznam zahrnuje více prvků. Tehdy musíme vyhledat jeho *předposlední* prvek, abychom mohli vhodně upravit ukazatel `ocas` informační struktury. Známe-li předposlední prvek seznamu, dokážeme snáze odstranit jeho poslední prvek. Následuje úplný kód funkce, která plní tento úkol.

```
SEZNAM& SEZNAM::operator -(int)           // parametr int je umělý
{
    if (inf.hlava==inf.ocas)             // (operátor -- bude přírůstkový)
    {
        delete inf.hlava;
        inf.hlava=inf.ocas=NULL;
    }
    else
    {
        PRVEK *temp=inf.hlava;
        while ((temp->dalsi) != inf.ocas) // hledání předposledního
            temp=temp->dalsi;           // prvku seznamu...
        inf.ocas=temp;
        delete temp->dalsi;           // ...a jeho odstranění
        temp->dalsi=NULL;
    }
    return (*this);                  // vrácení upraveného objektu
}
```

Objekt je vrácen svou adresou, takže může sloužit jako argument libovolné operace, kterou s ním lze provést. Můžeme utvořit třeba výraz `(12--)--.vypis()`. Tato instrukce sice vypadá hrozivě, ale přitom je úplně triviální: první dekrementace vrací skutečný, fyzicky existující objekt, který okamžitě prochází druhou dekrementací. Výsledek druhé dekrementace – jako plnohodnotný

KAPITOLA 5 Datové typy a struktury

objekt – může aktivovat libovolnou metodu své třídy, tedy například zkontrolovat svůj obsah pomocí funkce `vypis`.

Když se zabýváme operátorem dekrementace, podívejme se ještě na jeho další možnosti. Definice třídy obsahuje i příslušný destruktor. Připomeňme, že destruktor je speciální funkce, která se automaticky volá při likvidaci objektu. Tato likvidace může probíhat přímo, například pomocí operátoru `delete`:

```
SEZNAM *p=new SEZNAM; // vytvoření nového objektu...
...
delete p; // ...a jeho likvidace
```

nebo také zprostředkováně ve chvíli, kdy ztratíme přístup k objektu. Jako příklad druhé situace uvedme následující fragment programu:

```
if (podminka)
{
    SEZNAM p; // vytvoření lokálního objektu, který
    ...         // je viditelný pouze v této instrukci if
}
```

Objekt `p` deklarovaný v těle instrukce `if` má výhradně lokální platnost. Nelze k němu přistupovat z žádného jiného úseku programu. Takový dočasný objekt často vyžaduje hodně paměti, která je vyhrazena pouze pro něj. Nebýt destruktoru, programátor by s jistotou nevěděl, zda byla tato paměť kompletně vrácena operačnímu systému. Zdůrazněme slovo „kompletně“ – automaticky lze totiž paměť uvolňovat jen v případě těch proměnných, které jsou ze své povahy umístěny v zásobníku. Týká se to například běžných datových položek objektu, ale stejný postup není možný v případě dynamických struktur, které jsou často rozptýleny v poměrně rozsáhlé části operační paměti. Mezi takové struktury patří seznamy, stromy, dynamická pole atd. V tomto případě musí programátor sám napsat destruktor s veřejným přístupem, který bude dokonale vědět⁶, jakým způsobem jsme objektu přidělili paměť, a dokáže tuto paměť správně vrátit.

Postupujeme tak i v tomto příkladu. Zápis destruktoru je překvapivě stručný:

```
~SEZNAM()
// destruktor, který používá pře definovaný operátor --
{
    while (!prazdne()) (*this)--;
}
```

Je to jednoduchý cyklus `while`, který ze seznamu odstraňuje prvky tak dlouho, dokud není prázdný. I když tento způsob uvolnění paměti není optimální, volíme jej proto, abychom ukázali možné využití ukazatele `this`, který – jak víme – směruje na vlastní objekt. Řádek `(*this)--` znamená, že na příslušném objektu proběhne operace dekrementace. Objekt, který je z nějakého důvodu určen k destrukci (typické případy jsme již uvedli výše), vyvolá svůj destruktor. Tento destruktor pak na objekt aplikuje funkci dekrementace tolíkrát, aby kompletně uvolnil paměť, která byla seznamu dříve přidělena.

Nyní vysvětlíme další část kódu, která souvisí s pře definováním operátoru `+` (plus). Chceme vytvořit takovou funkci, která dokáže *scítat* seznamy v co nejdoslovnějším významu tohoto slova. Chceme, aby po provedení následujících instrukcí:

6 Ve skutečnosti to víme *my* a tuto cennou informaci poskytujeme destruktoru. Občasným personifikacím se v popisech tohoto typu bohužel nedá vyhnout.

```
SEZNAM x,y,z; // vytvoření 3 prázdných seznamů
x.vloz2(3); x.vloz2(2); x.vloz2(1);
y.vloz2(6, y.vloz2(5); y.vloz2(4);
z=x+y;
```

výsledný seznam z obsahoval všechny prvky seznamů x i y, tzn.: 1, 2, 3, 4, 5 a 6 (setříděn!). Nejjednodušší metodou je zkopírovat všechny prvky ze seznamů x a y do seznamu z a zároveň pro naposledy uvedený seznam aktivovat metodu vloz2. Tím zajistíme, že vzniklý seznam bude již setříděný:

```
SEZNAM& operator +(SEZNAM &x, SEZNAM &y)
{
    SEZNAM *temp=new SEZNAM;
    PRVEK *q1=(x.inf).hlava; // pracovní ukazatele
    PRVEK *q2=(y.inf).hlava;
    while (q1 != NULL)          // zkopírování seznamu x do temp
    {
        temp->vloz2(q1->hodnota);
        q1=q1->dalsi;
    }
    while (q2 != NULL)          // zkopírování seznamu y do temp
    {
        temp->vloz2(q2->hodnota);
        q2=q2->dalsi;
    }
    return (*temp);
}
```

Je tato metoda optimální? Asi ne, už kvůli zbytečnému duplikování dat. Ideální metoda by vycházela z toho, že seznamy jsou již seřazené⁷, a složila by je (neboli *fúzovala*) s použitím výhradně existujících paměťových buněk a bez tvoření nových. Jinak řečeno: nezbývá nám než manipulovat výhradně ukazateli. Jiné nástroje bohužel k dispozici nemáme.

Například na obrázku 5.7 můžeme sledovat, jak by měla vypadat fúze dvou konkrétních seznamů $x=(1, 3, 9)$ a $y=(2, 3, 14)$, aby jeden z těchto seznamů následně obsahoval všechny prvky původních seznamů x i y – samozřejmě setříděné (v našem případě neklesajícím způsobem).

Menší ze dvou prvních prvků seznamu je prvek 1, který bude také představovat začátek nového seznamu. Následníkem tohoto prvku bude fúze dvou seznamů: $x' = (3, 9)$ a $y = (2, 3, 14)$.

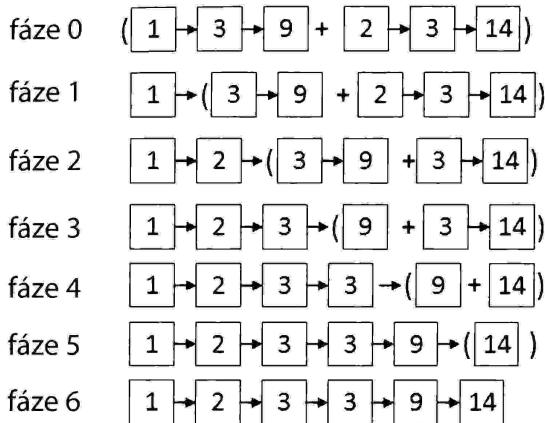
Jak sloučit seznamy x' a y? Úplně stejně: Bereme prvek 2, který je menším ze dvou prvních prvků seznamů x' a y... Tak můžeme rekurzivně pokračovat až do okamžiku, kdy narazíme na elementární případy: je-li některý ze seznamů prázdný, výsledkem sloučení obou seznamů bude samozřejmě druhý seznam. Na tomto principu je postavena procedura fuze(ob1, ob2). Když ji vyvoláme se dvěma parametry ob1 a ob2, vrátí v seznamu ob1 sumu prvků seznamů ob1 a ob2. Seznam ob2 se po této operaci vynuluje, avšak úplné odstranění musí programátor provést sám (tak jsme se rozhodli – stejně dobře bychom mohli seznam odstraňovat automaticky).

Úkol splníme způsobem, který sice není pro jazyk C++ úplně typický, ale ukáže nám šíři možných uplatnění tzv. *funkcí spřátených* se třídou. Připomeňme, že se jedná o funkce (nebo procedury), které sice nejsou metodami dané třídy, ale mají přístup k chráněným položkám *private* a *protected* objektu, jehož adresa jim byla předána v rámci parametrů volání. Vzhledem k tomu,

⁷ Současně předpokládáme, že seznam budeme tvořit metodou vloz2.

KAPITOLA 5 Datové typy a struktury

že nepatří mezi metody třídy, nelze je volat pomocí tečkové notace. Objekt, který mají zpracovávat, jim navíc musíme předat explicitním způsobem – například pomocí jeho adresy.



Obrázek 5.7: Příklad fúze seznamů

Sloučení seznamů probíhá ve dvou fázích. Nejdříve připravíme jednoduchou funkci, která bude přijímat dva setříděné seznamy a a b a jako výsledek bude vracet sloučený seznam. Rekurzivní zápis tohoto procesu je velmi jednoduchý a jeho styl připomíná řešení práce se seznamy v jazyčích typu LISP či PROLOG:

```

PRVEK *setrid(PRVEK *a, PRVEK *b)
{
    if (a==NULL)
        return b;
    if (b==NULL)
        return a;
    if (a->hodnota<=b->hodnota)
    {
        a->dalsi=setrid(a->dalsi,b);
        return a;
    }
    else
    {
        b->dalsi=setrid(b->dalsi,a);
        return b;
    }
}

```

Když máme k dispozici funkci `setrid`, můžeme ji využít v proceduře `słoučení`. Díky tomu, že se jedná o funkci sprátelenou se třídou `SEZNAM`, může libovolně manipulovat se soukromými komponentami seznamů x a y, které jsme jí předali při volání⁸.

```

void fuze(SEZNAM &x, SEZNAM &y)
{ // seznamy a i b musí být setříděné
    PRVEK *a=x.inf.hlava,*b=y.inf.hlava;
    PRVEK *vysledek=setrid(a,b);
    x.inf.hlava=vysledek;
}

```

⁸ Další informace nalezneme také v příloze A, kde je princip spřátelené funkce vysvětlen na jiném příkladu.

```

    if(x.inf.ocas->hodnota <= y.inf.ocas->hodnota)
        x.inf.ocas=y.inf.ocas;
    else x.inf.ocas=x.inf.ocas;
    y.nuluje();
}

```

Záměrně značně propracovaná funkce main ukazuje, jakým způsobem lze s výše popsanými funkcemi pracovat. Do obou seznamů jsou vloženy položky pole. Následně dochází k testování některých metod a třídění dvou seznamů jejich sloučením.

```

int main()
{
    SEZNAM l1,l2;
    const int n=6;
    int tab1[n]={2,5,-11,4,14,12};
    // všechny prvky pole jsou vloženy do seznamu
    cout << "\nL1 = ";
    for (int i=0; i<n; l1.vloz2(tab1[i++]));
    l1.vypis();
    int tab2[n]={9,6,77,1,7,4};
    cout << "L2 = ";
    for (int i=0; i<n; l2.vloz2(tab2[i++]));
    l2.vypis();
    cout << "Výsledek vyhledávání čísla 14 v seznamu l1: "
        << l1.hledej(14) << endl;
    cout << "Výsledek vyhledávání čísla 0 v seznamu l1: "
        << l1.hledej(0) << endl;
    cout << "Tento seznam vznikl sloučením dvou předchozích\nL3 = ";
    SEZNAM l3=l1+l2;
    l3.vypis();
    cout << "Seznamy L1 a L2 se nezměnily:\nL1 = ";
    l1.vypis();
    cout << "L2 = ";
    l2.vypis();
    cout << "Seznam L1 bez dvou posledních prvků:\nL1 = ";
    (l1--)--.vypis();
    cout << "Výsledek sloučení seznamů L1 a L2:\n";
    fuze(l1,l2);
    cout << "L1 = ";
    l1.vypis();
    cout << "L2 = ";
    l2.vypis();
    l1.vloz2(80);l1.vloz2(8);
    cout << "Do seznamu L1 přidáváme čísla 80 a 8\nL1 = ";
    l1.vypis();
}

```

Program po spuštění poskytne tyto výsledky:

```

L1 = -11 2 4 5 12 14
L2 = 1 4 6 7 9 77
Výsledek vyhledávání čísla 14 v seznamu l1: 1
Výsledek vyhledávání čísla 0 v seznamu l1: 0
Tento seznam vznikl sloučením dvou předchozích

```

KAPITOLA 5 Datové typy a struktury

```
L3 = -11 1 2 4 4 5 6 7 9 12 14 77
Seznamy L1 a L2 se nezměnily:
L1 = -11 2 4 5 12 14
L2 = 1 4 6 7 9 77
Seznam L1 bez dvou posledních prvků:
L1 = -11 2 4 5
Výsledek sloučení seznamů L1 a L2:
L1 = -11 1 2 4 4 5 6 7 9 77
L2 = (seznam je prázdný)
Do seznamu L1 přidáváme čísla 80 a 8
L1 = -11 1 2 4 4 5 6 7 8 9 77 80
```

Jednosměrné seznamy – teorie a praxe

Kromě elegantních teoretických vývodů existuje ještě tvrdá realita, ve které musí naše pečlivě se-stavené programy fungovat⁹.

Zkusme objektivně zhodnotit výhody i nevýhody jednosměrných seznamů:

- *Nedostatky*: nepřirozený přístup k prvkům, složité třídění, ztížená analýza obsahu a hodno-cení velikosti seznamu
- *Přednosti*: efektivní využití paměti, flexibilita

Nyní podrobně analyzujme aspekt třídění prvků seznamu. Jak si snadno dokážeme představit, třídění seznamů je dosti komplikované, protože tato datová struktura není v paměti uložena sou-visle (na rozdíl od polí).

Seznam, do kterého vkládáme nové prvky, proto již od začátku kromě své základní role shroma-žďování dat slouží zároveň i k jejich uspořádání. Jedná se o výhodnou vlastnost – o třídění dat se stará samotná datová struktura. V situaci, kdy existuje jen jedno třídicí kritérium (např. ve směru neklesajících hodnot určité položky x), můžeme to považovat za ideální. Co si však počneme, když seznam sestává ze záznamů, které mají mnohem složitější strukturu, např.:

```
struct
{
    char jmeno[20];
    char prijmeni[30];
    int vek;
    int kod_pracovnika;
}
```

V jednom případě můžeme chtít, aby byl takový seznam seřazen abecedně podle příjmení, a jin- dy nás zase bude zajímat věk pracovníků. Budeme v takovém případě pracovat se dvěma verze-mi tohoto seznamu (na úkor cenné paměti počítače), nebo se rozhodneme podle požadovaného kritéria třídit jeden seznam? Musíme si však uvědomit, že druhé řešení zase zvyšuje nároky na procesorový čas.

Výše uvedené dilema se vyskytovalo v tolika praktických programech, že se nakonec objevilo jeho důmyslné řešení, které nyní podrobně vysvětlíme.

Princip tohoto řešení na jednu stranu zjednodušuje, ale z jiného hlediska zase komplikuje to, co jsme se dosud naučili. Zjednodušení spočívá v tom, že *záznamy ukládané do seznamu nejsou žád-ným způsobem tříděny*. Jinak řečeno: k jejich zapamatování můžeme využít ekvivalent jednoduché

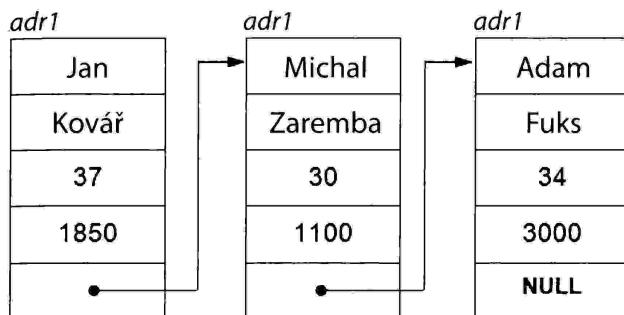
⁹ Na vysvětlenou uvedeme, že nemáme na mysli prostředí operačního systému.

funkce `vloz1` (viz stranu 101). Slovo „ekvivalent“ je zde nevhodnější. Jak se totiž ukáže, funkce bude potřebovat několik kosmetických úprav, které souvisejí s obecnou změnou koncepce.

Vedle seznamu dat budeme navíc disponovat několika seznamy ukazatelů na data. Těchto seznamů bude tolik, kolik třídících kritérií potřebujeme.

Je zřejmé, že nemáme-li v úmyslu třídit seznam dat (a zároveň chceme mít přístup k setříděným datům), musíme třídit seznamy adres pokaždé, když do nich vkládáme novou adresu. Podobnou roli plnila funkce `vloz2`, ale na rozdíl od ní data nyní nejsou přístupná bezprostředně.

Během třídění seznamů ukazatelů se poloha dat vůbec nemění – v příslušných seznamech se přemisťují pouze samotné ukazatele. Z dosavadního popisu nemusí být úplně jasné, jak to vlastně funguje. Je proto načase, abychom uvedli nějaký konkrétní příklad. Podívejme se tedy na obrázek 5.8.

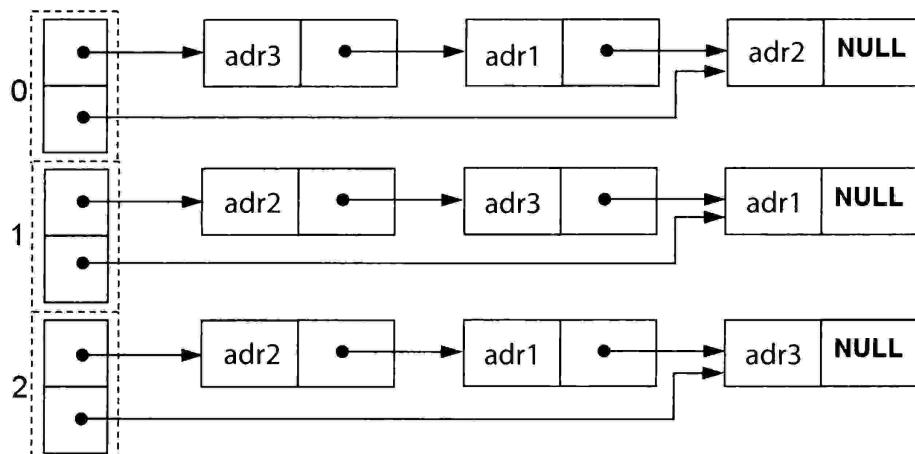


Seznam DATA

Obrázek 5.8: Třídění seznamu bez přemisťování jeho prvků (1)

Znázorňuje seznam s názvem DATA s několika záznamy, které tvoří začátek malé databáze pracovníků nějaké firmy. Pro jednoduchost předpokládejme, že potřebujeme ukládat pouze následující informace: jméno, příjmení, určitý kód a samozřejmě výši platu. Na obrázku jsou označeny symbolické adresy záznamů `adr1`, `adr2` a `adr3`, které jim přidělila funkce `vloz1`.

Na obrázku 5.9 si můžeme všimnout několika změn oproti tomu, s čím jsme se dosud setkali.



Seznam TAB_PTR

Obrázek 5.9: Třídění seznamu bez přemisťování jeho prvků (2)

KAPITOLA 5 Datové typy a struktury

Pole TAB_PTR obsahuje informační záznamy (tzn. ukazatele hlava a ocas) seznamů, které se skládají z adres záznamů v seznamu DATA. V našem případě vytváříme tři seznamy ukazatelů, které budou samozřejmě obsahovat adresy adr1, adr2 a adr3 (seznam bude dočasně zahrnovat tři prvky, ale jak budou přibývat prvky v seznamu DATA, poroste rovněž počet ukazatelů na ně).

Velikost pole TAB_PTR je dána počtem třídicích kritérií: při pohledu shora si můžeme všimnout, že seznamy jsou postupně seřazeny podle příjmení, kódu a výdělku.

Shrňme, co lze vyčíst z obrázků 5.8 a 5.9:

- Neseříděná databáze, jejíž data jsou uložena v seznamu s názvem DATA, aktuálně obsahuje 3 záznamy.
- Pole ukazatelů TAB_PTR obsahuje 3 informační záznamy (popsané výše), jejichž položky hlava a ocas poskytují přístup ke třem seznamům ukazatelů. Každý z těchto seznamů je setříděn podle jiného kritéria:
 - Seznam, na který odkazuje záznam TAB_PTR[0], je setříděn abecedně podle příjmení pracovníků (Fuks, Kovář a Zaremba).
 - Analogicky seznam TAB_PTR[1] klasifikuje pracovníky podle jistého firemního kódu (Zaremba, Fuks a Kovář).
 - Poslední seznam (TAB_PTR[2]) řadí pracovníky podle jejich platů.

Následuje nová verze třídy SEZNAM, která již zohledňuje koncepční změny představené na obrázku 5.8. Kvůli zjednodušení výstupů ukázkového programu jsme museli poněkud zjednodušit datovou strukturu, která obsahuje data o zaměstnancích: omezíme se pouze na příjmení a mzdu. (Kdybychom tyto datové struktury rozšířili, jejich princip by nám to nijak nejasnilo, ale navíc bychom tím zhoršili přehlednost výpisu, který je beztak dosti dlouhý.)

Datové struktury v nové verzi vypadají takto:

```
typedef struct rob
{
    char prijmeni[100];
    long vydelen;
    struct rob *dalsi;           // ukazatel na
                                // následující prvek
}PRVEK;

typedef struct rob_ptr      // pracovní struktura seznamu
{                           // ukazatelů
    PRVEK *adresa;
    struct rob_ptr *dalsi;
}LPTR;
```

Žádné zásadní změny zatím nejsou vidět. Čtenář se proto může oprávněně zeptat, proč neuplatníme mechanizmy dědění, abychom co nejvíce využili již napsaný kód. Důvod je jednoduchý: Předchozí verze třídy SEZNAM v zásadě jen umožňovala předvést principy a základní algoritmy související s jednosměrnými seznamy. Nemá proto valný praktický význam a její použití jako bázové třídy by bylo poněkud umělé. Bázové třídy jazyka C++ by měly být co nejuniverzálnější, aby jejich použití při dědění působilo přirozeně.

Aktuálně prezentovaná verze struktury jednosměrného seznamu se vyznačuje vysokou flexibilitou používání. Právě tato třída by mohla nakonec posloužit jako bázová třída v další hierarchii dědění (budeme-li mechanizmus dědění vůbec potřebovat – tento mechanizmus bychom neměli aplikovat jen pro efekt).

Následuje nová verze třídy SEZNAM:

```
lista2.h
const int n=5;
const int kriteria_trideni=2; // počet kritérií třídění

typedef struct rob
{
    char prijmeni[100];
    long vydelek;
    struct rob *dalsi;           // ukazatel na
}PRVEK;                         // následující prvek

typedef struct rob_ptr           // pracovní struktura seznamu
{                                // ukazatelů
    PRVEK *adresa;
    struct rob_ptr *dalsi;
}LPTR;

class SEZNAM
{
public:
    SEZNAM();                      // konstruktor
    ~SEZNAM();                     // destruktor
    void vloz(PRVEK *);           // připojí nový prvek q
    void vypis(char);              // vypíše obsah seznamu
    int odstran(PRVEK*, int(*rozhodnuti)(PRVEK *,PRVEK*));
    // odstraní prvek, který se shoduje se vzorovou buňkou
    // uvedenou jako parametr
private:
    // soukromé struktury:
    typedef struct                  // informační struktura seznamu dat
    {
        PRVEK *hlava;
        PRVEK *ocas;
    }
    INFO;
    typedef struct                  // informační struktura seznamu ukazatelů
    {
        LPTR *hlava;
        LPTR *ocas;
    }LPTR_INFO;

    LPTR_INFO inf_ptr[kriteria_trideni];
    INFO info_data;
    // "vnitřní" metody, veřejně nedostupné:
    LPTR_INFO *vyhledej_ukz(LPTR_INFO*,PRVEK*,
                           int(*)(PRVEK*,PRVEK*));
    PRVEK *odstran_ukz(LPTR_INFO*, PRVEK*, int()(PRVEK*,PRVEK*));
    int odstran_data(PRVEK*);
    void vloz2(int, PRVEK*, int(*rozhodnuti)(PRVEK*,PRVEK*));
    void vypis1(LPTR_INFO*);
};
```

KAPITOLA 5 Datové typy a struktury

Tajemné soukromé metody, které jsme výše uvedli bez jakéhokoli popisu, podrobně vysvětlíme v následujících odstavcích.

Když analyzujeme procedury a funkce na obsluhu seznamů, můžeme si všimnout, že se od sebe příliš neliší operace vyhledávání určitého prvku podle zadaného vzorce (např. „vyhledej pracovníka, který vydělává 12 000 Kč“) a operace vyhledávání místa k vložení nového prvku. Od tohoto postřehu k praktické programové realizaci vede jen jeden krok. Předtím ale musíme dobře rozumět pravidlům práce s ukazateli na funkce¹⁰ v jazyce C++, protože díky těmto ukazatelům dokážeme elegantně vyřešit některé problémy. Kvůli tomu, že ukazatele na funkce se v programech objevují poměrně zřídka, je potřeba ukázat, jak se s nimi v jazyce C++ nakládá. Tyto informace pomohou zejména programátorům v jazyce Pascal, protože jejich oblíbený jazyk tímto mechanizmem vůbec nedisponuje.

Dále uvedený příklad ukazuje, jakým způsobem lze používat ukazatele na funkce v jazyce C++.

```
wsk_fun.cpp
int do_2(int a)
{
    return a*a;
}
int do_4(int a)
{
    return a*a*a*a;
}
int vzor(int x,int(*fun)(int))
{
    return fun(x);
}
int main()
{
    cout << "10 umocněno na 2:" << vzor(10,do_2) << endl;
    cout << "10 umocněno na 4:" << vzor(10,do_4) << endl;
}
```

Funkce `vzor` vrací – podle toho, zda ji vyvoláme jako `vzor(10,do_2)` nebo `vzor(10,do_4)` – buď 100, nebo 10000. Setkáváme se zde s podobným jevem jako v případě polí, kde název (`pole`, `funkce`) je zároveň i ukazatelem. Z toho vzhledem vyplývá, že s těmito názvy můžeme pracovat velmi přirozeně a dokážeme se přitom vyhnout typickým operátorům jazyka C++: `*` (operátor zjištění hodnoty) a `&` (operátor zjištění adresy).

Jiný příklad: Určitou proceduru `f`, která jako parametr přijímá číslo `x` (typu `int`) a ukazatel na funkci s názvem `g` (ta vrací typ `double` a přijímá tři parametry: `int`, `double` a `char*`), lze deklarovat následujícím způsobem:

```
void f(int x, double(*g)(int, double, char *))
{
    k=g(12,5.345,"1984");
    cout << k << endl;
}
```

Ukazatele na funkce se uplatní v různých situacích a dovolují zobecnit mnoho procedur i funkcí.

10 Znalci jazyka LISP mohou tento odstavec přeskočit.

Vraťme se nyní k tématu seznamů a zkusme vyřešit problém vložení nového prvku do dříve seřazeného seznamu. Chceme najít dvě adresy: pred a po (viz obrázek 5.6 na straně 103), které nám umožní upravit ukazatele takovým způsobem, aby celý seznam vypadal jako setříděný. Přitom budeme potřebovat cyklus while, který jsme viděli na straně 104:

```
while((stav==HLEDEJ) && (po!=NULL))
    if (po->vydelek >= x)
        stav=KONEC;
    else
    {
        pred=po;
        po=po->dalsi;
    }
```

Kdybychom chtěli odstranit určitý prvek seznamu, který například splňuje podmínu, že hodnota položky vydelek se rovná 12 000 Kč, budeme ukazatele pred a po potřebovat také. Najdeme je tímto způsobem:

```
while((stav==HLEDEJ) && (po!=NULL))
    if (po->vydelek == 12000)
        stav=KONEC;
    else
    {
        pred=po;
        po=po->dalsi;
    }
```

Oba cykly while se liší pouze podmínkou instrukce if-else. Naše řešení je založeno na následujícím principu: Napišeme univerzální funkci, která umožní vyhledat ukazatele pred a po, abychom pomocí nich mohli později vkládat prvky do seznamu nebo prvky ze seznamu odstraňovat. Funkce by měla vrátit oba ukazatele. Přitom využijeme strukturu LPTR_INFO (viz výpis LISTA2.H) a budeme vycházet z toho, že položka h̄lava odpovídá ukazateli pred a položka ocas ukazateli po.

Je zřejmé, že operace vyhledávání, vkládání atd. začínáme od seznamu ukazatelů, kde získáme adresu datového záznamu (tuto adresu uložíme do adresní položky struktury LPTR, která slouží k uchovávání seznamu ukazatelů – viz obrázek 5.9). Teprve poté, co upravíme všechny seznamy ukazatelů (kterých může být tolik, kolik je třídicích kritérií), můžeme modifikovat i datový seznam. Na první pohled vidíme, že budeme mít hodně práce. Tato investice se nám však vrátí, protože si značně usnadníme pozdější používání seznamu. Může nás přitom těšit, že když jednou napišeme příslušnou sadu bázových funkcí, budeme tyto funkce později moci používat opakováně, aniž bychom museli znova ověřovat, jak vlastně fungují. Přejděme nyní k popisu realizace funkce vyhledej_ukz, jejímž úkolem je vyhledat ukazatele pred a po a vrátit je ve struktuře LPTR_INFO:

- Ukazatel inf na informační strukturu seznamu ukazatelů – počáteční adresa se nachází v položce h̄lava a koncová adresa v položce ocas.
- Ukazatel q na určitý fyzicky existující datový záznam. Jedná se buď o nový záznam, který chceme připojit k seznamu, nebo prostě o nějakou šablonu vyhledávání.
- Ukazatel rozhodnutí na porovnávací funkci, která bude vložena do instrukce if v cyklu while.

KAPITOLA 5 Datové typy a struktury

```
1ista2.cpp
SEZNAM::LPTR_INFO* SEZNAM::vyhledej_ukz(SEZNAM::LPTR_INFO *inf, PRVEK *q,
    int(*rozhodnuti)(PRVEK *q1, PRVEK *q2))
{
    LPTR_INFO *res=new LPTR_INFO;
    res->hlava=res->ocas=NULL;
    if (inf->hlava==NULL)
        return (res);           // seznam je prázdný!
    else
    {
        LPTR *pred,*pos;
        pred=NULL;
        pos=inf->hlava;
        enum {HLEDEJ,KONEC} stav=HLEDEJ;
        while ((stav==HLEDEJ) && (pos!=NULL))
            if (rozhodnuti(pos->adresa,q))
                stav=KONEC;      // nalezeno místo, kde se prvek
            else                 // nachází (nebo kam má být vložen)
            {
                pred=pos;          // pokračuje hledání
                pos=pos->dalsi;
            }
        res->hlava=pred;
        res->ocas=pos;
        return (res);
    }
}
```

Pokud například chceme vyhledat a odstranit první záznam, který v položce příjmeni obsahuje řetězec „Kovář“, musíme vytvořit dočasný záznam, který bude mít v příslušné položce uvedeno stejné příjmení (zbývající položky nemají na vyhledávání žádný vliv):

```
PRVEK *f=new PRVEK;
strcpy(f->prijmeni,"Kovář");
```

Totéž platí i pro vyhledávání podle jiných kritérií – výdělku, jména atd. Jestliže je vyhledávání úspěšné, bude v položce ocas vrácena adresa fyzicky existujícího záznamu, který odpovídá vyhledávanému vzoru. Pokud by takový prvek neexistoval, funkce by měla vrátit hodnoty NULL. Známe-li ukazatele pred a po, můžeme uvolnit paměťové buňky, které datový záznam dosud zabíral, a také příslušným způsobem upravit celý seznam, aby všechny jeho prvky byly na správných místech.

Jako jiný příklad využití funkce lze uvést vložení nového prvku do seznamu. Tehdy je potřeba vytvořit nový záznam, vyplnit jeho položky a umístit jej na konec seznamu. Poté musíme adresu tohoto prvku vložit do seznamů ukazatelů, které jsou seřazeny podle výdělků, příjmení či libovolných jiných kritérií. Do každého z těchto seznamů budeme prvek vkládat najinou pozici. Hodnoty ukazatelů pred a po, které vrátí funkce vyhledej_ukz, se proto mohou v jednotlivých případech lišit.

Je zřejmé, že funkci vyhledej_ukz lze využít nejrůznějšími způsoby. Této flexibility jsme dosáhli výhradně díky ukazatelům na funkci, které jsme ve správném čase umístili na vhodné místo.

Uvedeme několik rozhodovacích funkcí, které mohou sloužit jako parametr:

```
lista2.h
int abecedne(PRVEK *q1,PRVEK *q2)
{ // jsou záznamy q1 a q2 uspořádány abecedně?
  return (strcmp(q1->prijmeni,q2->prijmeni)>=0);
}
int podle_vydeleku(PRVEK *q1,PRVEK *q2)
{ // jsou záznamy q1 a q2 uspořádány podle výdělků?
  return (q1->vydelek>=q2->vydelek);
}
int ident_prijmeni (PRVEK *q1,PRVEK *q2)
{ // mají záznamy q1 a q2 identická příjmení?
  return (strcmp(q1->prijmeni,q2->prijmeni)==0);
}
int ident_vydelky (PRVEK *q1,PRVEK *q2)
{ // mají záznamy q1 a q2 identické výdělky?
  return (q1->vydelek==q2->vydelek);
}
```

První dvě funkce umožňují uspořádat seznam a zbývající funkce usnadňují proces vyhledávání prvků. V reálné aplikaci databáze pracovníků by analogické funkce samozřejmě byly poněkud složitější. Vše závisí na tom, jaká kritéria vyhledávání nebo řazení chceme naprogramovat a nakolik složité datové struktury potřebujme zpracovávat.

Po obsáhlém popisu by již mělo být každému jasné, jak pracuje funkce `vyhledej_ukz`.

Na straně 100 jsme se mohli setkat s funkcí `prazdne`, která informuje o tom, zda datový seznam něco obsahuje. Nic nám nebrání rozšířit sadu funkcí o její další verzi, která bude analogickým způsobem zkoumat seznam ukazatelů¹¹:

```
inline int prazdne(LPTR_INFO *inf)
{
  return (inf->h lava==NULL);
}
```

Vzhledem k tomu, že jsme dvakrát použili stejný název funkce, došlo by v té chvíli k jejímu *přetížení*. Program při své činnosti zvolí odpovídající verzi funkce podle typu parametru, se kterým je volána (ukazatel na strukturu `INFO` nebo ukazatel na strukturu `LPTR_INFO`).

Když nyní máme k dispozici funkci `prazdne`, sadu rozhodovacích funkcí i univerzální funkci `vyhledej_ukz`, pokusme se napsat chybějící proceduru `vloz1`, která umožní připojovat k seznamu dat nové záznamy a zároveň třídit seznamy ukazatelů. Předpokládejme, že se budou používat pouze dvě kritéria třídění dat. Pole, které obsahuje „ukazatele na seznamy ukazatelů“, bude mít v tom případě jen dvě pozice (viz obrázek 5.9).

Adresu tohoto pole spolu s ukazateli na datový seznam i nově vytvořený prvek je nutné funkci předat jako parametry:

```
void SEZNAM::vloz(PRVEK *q)
{
  if (info_data.h lava==NULL) // připojení záznamu bez třídění
    // seznam je prázdný
```

¹¹ Typ inline označuje, že funkce je volána jako makro. To znamená, že na místo jejího výskytu je vložen celý její kód, což prodlužuje výsledný kód programu. Funkce inline je výhodná díky tomu, že značně zvyšuje čitelnost programu, ve kterém lze mnoho primitivních funkcí (například vícenásobných přiřazení) nahradit jedním „aliasem“. Kromě toho nejde o funkční volání, takže se vyhneme celé režii spojené s obsluhou zásobníku.

KAPITOLA 5 Datové typy a struktury

```
    info_data.hlava=info_data.ocas=q;
else                                // seznam není prázdný
{
    (info_data.ocas)->další=q;
    info_data.ocas=q;
}
// připojení ukazatele na záznam v abecedně seřazeném seznamu
vloz2(0,q,abecedne);
// připojení ukazatele na záznam v seznamu seřazeném podle výdělků
vloz2(1,q,podle_vydelku);
}
```

Funkce je velice jednoduchá, hlavně díky tajemné proceduře s názvem `vloz2`. Samozřejmě nejde o proceduru ze strany 103, i když se od ní příliš nelíší:

```
void SEZNAM::vloz2(int poc, PRVEK *q, int(*rozhodnuti)(PRVEK *q1, PRVEK *q2))
{
    LPTR *ukz=new LPTR;
    ukz->adresa=q; // zadání adresy záznamu q
    // Hledání vhodné pozice k vložení prvku
    if (inf_ptr[poc].hlava==NULL) // prázdný
    {
        inf_ptr[poc].hlava=inf_ptr[poc].ocas=ukz;
        ukz->další=NULL;
    }
    else //hledání místa k vložení
    {
        LPTR *pred,*po;
        LPTR_INFO *kde=vyhledej_ukz(&inf_ptr[poc],q,rozhodnuti);
        pred=kde->hlava;
        po=kde->ocas;
        if (pred==NULL)      // vložení na začátek seznamu
        {
            inf_ptr[poc].hlava=ukz;
            ukz->další=po;
        }
        else
            if (po==NULL) // vložení na konec seznamu
            {
                inf_ptr[poc].ocas->další=ukz;
                ukz->další=NULL;
                inf_ptr[poc].ocas=ukz;
            }
            else          // vložení někam "doprostřed"
            {
                pred->další=ukz;
                ukz->další=po;
            }
    }
}
```

Chceme-li pochopit provedené úpravy, mohli bychom porovnat obě verze funkce `vloz2` a vyhledat jejich rozdíly. Principiálně jich není mnoho – místo třídění dat prostě třídíme ukazatele na data.

Funkci, která odstraňuje záznamy, je nutné předat mj. fyzickou adresu prvku, který má odstranit. Na základě této informace je potřeba vyčistit jak seznam dat, tak i seznamy ukazatelů:

```
int SEZNAM::odstran(PRVEK *q, int(*rozhodnuti)(PRVEK *q1, PRVEK *q2))
{
    // kompletní odstranění informací z obou seznamů (ukazatelů i dat)
    PRVEK *ptr_data;
    for (int i=0; i<kriteria_trideni; i++)
        ptr_data=odstran_ukz(&inf_ptr[i], q, rozhodnuti);
    if (ptr_data==NULL)
        return(0);
    else
        return odstran_data(ptr_data);
}
```

Funkce `odstran_ukz` zajišťuje odstranění ukazatelů na daný prvek ze seznamů ukazatelů (bez ohledu na počet těchto seznamů). Struktura tohoto algoritmu na první pohled značně připomíná dříve uvedené algoritmy.

Dá se tedy říci, že výpis kódu je uveden jen pro úplnost. Základní kontrolu chyb zajišťuje hodnota, kterou funkce vrací: adresa fyzického záznamu, který chceme odstranit, by obvykle neměla mít hodnotu NULL.

```
PRVEK* SEZNAM::odstran_ukz(LPTR_INFO *inf, PRVEK *q,
    int(*rozhodnuti)(PRVEK *q1, PRVEK *q2))
{
    if (inf->hlava==NULL) // seznam je prázdný, takže nelze nic odstranit
        return NULL;
    else // hledání prvku k odstranění
    {
        LPTR *pred, *pos;
        LPTR_INFO *kde=vyhledej_ukz(inf, q, rozhodnuti);
        pred=kde->hlava;
        pos=kde->ocas;
        if (pos==NULL) return NULL; // prvek nebyl nalezen

        if (pos==inf->hlava) // odstranění ze začátku seznamu
            inf->hlava=pos->dalsi;
        else
            if (pos->dalsi==NULL) // odstranění z konce seznamu
            {
                inf->ocas=pred;
                pred->dalsi=NULL;
            }
            else // odstranění odněkud "zprostředka"
                pred->dalsi=pos->dalsi;
        PRVEK *ret=pos->adresa;
        delete pos;
        return ret;
    }
}
```

Funkce `odstran_data` má podobné schéma jako funkce `odstran_ukz`. Předpokládáme, že *prvek, který chceme odstranit, existuje*. Musíme proto důsledně kontrolovat, zda jsou prováděné operace

KAPITOLA 5 Datové typy a struktury

správné. Náš kód tento požadavek splňuje – případné nesrovnalosti lze zjistit již při pokusu o odstranění ukazatele a v takovém případě nebude odstraněn žádný záznam.

```
int SEZNAM::odstran_data(PRVEK *q)
{ // předpoklad: q existuje!
    PRVEK *pred, *pos;
    pred=NULL;
    pos=info_data.hlava;
    while ((pos!=q) && (pos!=NULL)) // hledání prvku "pred"
    {
        pred=pos;
        pos=pos->dalsi;
    }
    if (pos!=q)
        return(0); // prvek nebyl nalezen?!
    if (pos==info_data.hlava)           // odstranění ze začátku seznamu
    {
        info_data.hlava=pos->dalsi;
        delete pos;
    }
    else
        if (pos->dalsi==NULL)          // odstranění z konce seznamu
        {
            info_data.ocas=pred;
            pred->dalsi=NULL;
            delete pos;
        }
        else                           // odstranění odněkud "zprostředka"
        {
            pred->dalsi=pos->dalsi;
            delete pos;
        }
    return(1);
}
```

I když jsme usilovali o to, aby byly výše uvedené funkce bezpečné, jejich kontrolní mechanizmy zůstávají značně zjednodušené. Pokud bychom psali větší program v jazyce C++, museli bychom správnost operací s ukazateli kontrolovat mnohem pečlivěji. Zhoršuje se tím sice čitelnost programů, ale na druhou stranu získáváme větší jistotu, že budou správně fungovat.

Následující příklad podrobně ukazuje, jak je možné používat novou verzi jednosměrného seznamu. Tento fragment kódu je sice poměrně dlouhý, ale přetiskujeme jej proto, že vysvětlovaná tematika je dosti složitá – programátoři, kteří zatím s ukazateli nemají dostatečné zkušenosti, by se jinak mohli ve výkladu ztratit. Vycházíme z toho, že díky podrobnému příkladu použití lze snáze pochopit teoretické principy.

Dvě jednoduché funkce `vypis1` a `vypis` mají za úkol úhledně vypisovat obsah databáze setříděný podle jednotlivých seznamů ukazatelů:

```
void SEZNAM::vypis1(LPTR_INFO *inf)
{ // výpis obsahu setříděného seznamu ukazatelů (samořejmě
    LPTR *q=inf->glowa; // nechceme vypsat ukazatele, protože se jedná o adresy,
    while (q != NULL) // ale údaje, na které odkazují)
    {
```

```

        cout << setw(9)<<q->adresa->prijmeni<< " vydělává "<<setw(4)
            << q->adresa->vydelek<<" Kč\n";
        q=q->dalsi;
    }
    cout << "\n";
}

void SEZNAM::vypis(char c)
{
    if (c=='a') // abecedně
        vypis1(&inf_ptr[0]);
    else
        vypis1(&inf_ptr[1]);
}

```

Funkce main testuje všechny nové mechanizmy:

```

int main()
{
    SEZNAM l1;
    char *tab1[n]={"Bednář","Berka","Frynta","Pražák","Černý"};
    int tab2[n]={13000,10000,12000,20000,30000};
    for (int i=0; i<n; i++)
    {
        PRVEK *novy=new PRVEK; // fyzické vytvoření nového záznamu...
        strcpy(novy->prijmeni,tab1[i]);
        novy->vydelek= tab2[i];
        novy->dalsi=NULL;
        l1.vloz(novy); // ...a jeho vložení do seznamu
    }
    cout << "\n*** Databáze setříděná abecedně ***\n";
    l1.vypis('a');
    cout << "*** Databáze setříděná podle výdělků ***\n";
    l1.vypis('z');
    PRVEK *f=new PRVEK;
    f->vydelek=20000;

    cout << "Výsledek odstranění záznamu pracovníka, který vydělává 20 000 Kč="
        << l1.odstran(f, ident_vydelek) <<endl;
    delete f;
    cout << "*** Databáze setříděná abecedně ***\n";
    l1.vypis('a');
    cout << "*** Databáze setříděná podle výdělků ***\n";
    l1.vypis('z');
}

```

Po spuštění programu bychom měli dostat tyto výsledky:

```

*** Databáze setříděná abecedně ***
Bednář vydělává 13000 Kč
Berka vydělává 10000 Kč
Černý vydělává 30000 Kč
Frynta vydělává 12000 Kč
Pražák vydělává 20000 Kč

```

KAPITOLA 5 Datové typy a struktury

```
*** Databáze setříděná podle výdělků ***
Berka vydělává 10000 Kč
Frynta vydělává 12000 Kč
Bednář vydělává 13000 Kč
Pražák vydělává 20000 Kč
Černý vydělává 30000 Kč
Výsledek odstranění záznamu pracovníka, který vydělává 20 000 Kč=1
*** Databáze setříděná abecedně ***
Bednář vydělává 13000 Kč
Berka vydělává 10000 Kč
Černý vydělává 30000 Kč
Frynta vydělává 12000 Kč
*** Databáze setříděná podle výdělků ***
Berka vydělává 10000 Kč
Frynta vydělává 12000 Kč
Bednář vydělává 13000 Kč
Černý vydělává 30000 Kč
```

Implementace seznamů pomocí polí

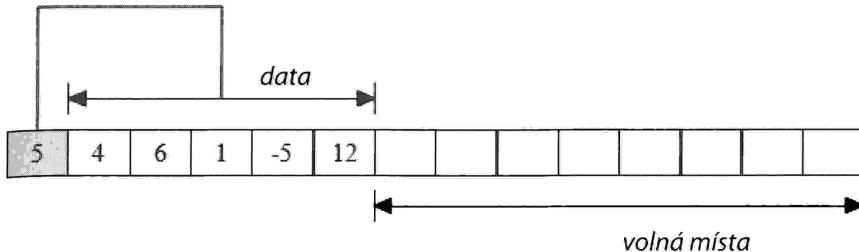
Jazyk C++ předpokládá, že programátoři dobře zvládají operace s dynamickými datovými strukturami, dokáží správně manipulovat s ukazateli apod. Není žádným tajemstvím, že někteří programátoři ukazatele nemají rádi. Ukazatelům se obvykle začnou vyhýbat poté, kdy se pokusí na programovat například strukturu seznamů, aniž by dokonale rozuměli tomu, co to obnáší. Jejich výsledky bývají zpravidla žalostné, ale místo aby hledali chybu u sebe, obviňují ze svých neúspěchů programovací jazyk. Stejně jako v jiných oborech však také platí, že když dva dělají totéž, není to totéž.

Toto pořekadlo se vztahuje i na seznamy. Ukazuje se, že seznamy lze pomocí polí implementovat několika způsoby. Některé z nich přitom mají dost podstatné výhody, které nelze získat při klasické realizaci (s níž jsme se seznámili v předchozí části kapitoly). Zásadní vadou struktur seznamů založených na polích je plýtvání pamětí. Oblast paměti (řekněme na 1 000 prvků) totiž přidělujeme trvale, i když ji maximálně využijeme jen v nejhorším případě. Pokud náš program bude pracovat se seznamem délky 200 prvků, reálně bude i tak obsazena paměť na uložení 1 000 prvků! Jedná se však o cenu, kterou platíme za jednoduchou realizaci.

Klasická reprezentace pomocí pole

Jedna z nejjednodušších metod, která mění pole na seznam, je založena na vhodné interpretaci obsahu pole. Pokud se rozhodneme (a později na to nezapomeneme), že i -tému indexu pole bude odpovídat i -tý prvek seznamu, svůj problém jsme téměř vyřešili. Slovo „téměř“ jsme uvedli proto, že ještě musíme určit, kolik prvků budeme do seznamu maximálně ukládat. Kromě toho je nutné zvolit nějakou proměnnou, která bude uchovávat aktuální počet prvků uložených do seznamu.

Princip je znázorněn na obrázku 5.10, kde vidíme implementaci seznamu založeného na poli, který obsahuje těchto pět prvků: 4, 6, 1, -5 a 12:



Obrázek 5.10: Implementace seznamu pomocí pole

Programová realizace je velmi jednoduchá – deklarace ukázkové třídy by nás neměla nicím překvapit.

```
list_tab.cpp
const int MaxPole=200;

class SeznamPole
{
public:
    SeznamPole() { tab[0]=0; } // konstruktor třídy
                           // metody jsou definovány dále:
    void OdstranPrvek(int k);
    void VlozPrvek(int x);
    void VlozPrvek(int x,int k);
    void VypisSeznam();
private:
    int tab[MaxPole];        // tab[0] rezervováno!
};
```

Nyní stručně popišme všechny obslužné funkce této třídy. Předpokládejme, že chceme mít možnost odstranit ze seznamu jeho k -tý prvek. Když zkontrolujeme, zda má operace smysl (prvek musí existovat), stačí přesunout obsah pole o jedno místo vlevo od k -té pozice. Během přesunování je prvek číslo k nenávratně přepsán svým sousedem:

```
void SeznamPole::OdstranPrvek(int k)
{ // odstranění k-tého prvku seznamu, k >= 1
    if((k>=1) && (k<=tab[0]))
    {
        for(int i=k;i<tab[0];i++)
            tab[i]=tab[i+1];
        tab[0]--;
    }
}
```

Výše uvedená funkce může mít hodně variant. Funkci lze vhodným způsobem upravit a přizpůsobit ji konkrétním požadavkům a potřebám.

Jak ale postupovat při *vkládání* prvků do seznamu? Dále popíšeme dvě verze příslušné funkce: první vkládá prvek na konec seznamu, druhá na jeho k -tou pozici. V případě druhé funkce je samozřejmě potřeba obsah pole vhodně přesunout, podobně jako u metody *OdstranPrvek*:

```
void SeznamPole::VlozPrvek(int x)
{ // vložení na konec seznamu
```

KAPITOLA 5 Datové typy a struktury

```
    if(tab[0]<MaxPole-1)
        tab[++tab[0]]=x;
    }
    //-----
void SeznamPole::VlozPrvek(int x,int k)
{ // vložení na k-tou pozici seznamu
    if((k>=1) && (k<=tab[0]+1) && (tab[0]<MaxPole-1))
    {
        for(int i=tab[0];i>=k;i--)
            tab[i+1]=tab[i]; // uvolnění místa
        tab[k]=x;
        tab[0]++;
    }
}
```

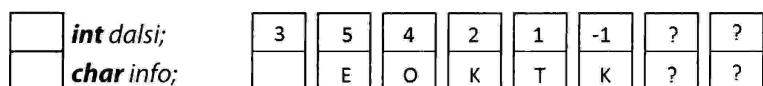
Po vytvoření všech metod lze s takovým pseudoseznamem pracovat shodným způsobem jako se skutečným jednosměrným seznamem, takže funkci `main` tentokrát nemusíme uvádět.

Třídu `SeznamPole` je možné samozřejmě definovat tak, aby byly prvky do seznamu vkládány již v klesajícím či rostoucím pořadí nebo podle nějakého jiného klíče. Chcete-li se trochu procvičit, můžete funkci a metody příslušným způsobem rozšířit.

Metoda souběžných polí

V předchozí implementaci seznamů založené na běžných polích jsme *i*-tému prvku pole pevně přiřadili *i*-tý prvek seznamu. U jednoduchých případů to může zcela stačit, ale takové řešení se koncepcně blíží spíše poli než seznamu. Opravdový seznam by měl umožňovat libovolné vkládání prvků a jejich třídění výhradně pomocí ukazatelů. Od ukazatelů, přidělování paměti a procedur `new` a `delete` jsme však chtěli utéci co nejdále! Že bychom se bez nich přece jen nedokázali obejít?

Naštěstí to zvládneme i bez nich. Simulovat se dá všechno, takže proč by ukazatele měly být výjimkou? Oblíbená metoda spočívá v tom, že deklarujeme pole záznamů, které se skládá z informační položky `info` a celočíselné položky `dalsi` – ta umožňuje vyhledat „následující“ prvek seznamu. Jedná se o dobře známé a vlastně již klasické řešení. Myšlenku představuje obrázek 5.11, který znázorňuje ukázkovou implementaci seznamu na ukládání znaků, která v daný okamžik obsahuje pět písmen tvořících slovo „KOTEK“.



Bázový záznam Ukázkové pole datových záznamů

Obrázek 5.11: Metoda „souběžných polí“ (1)

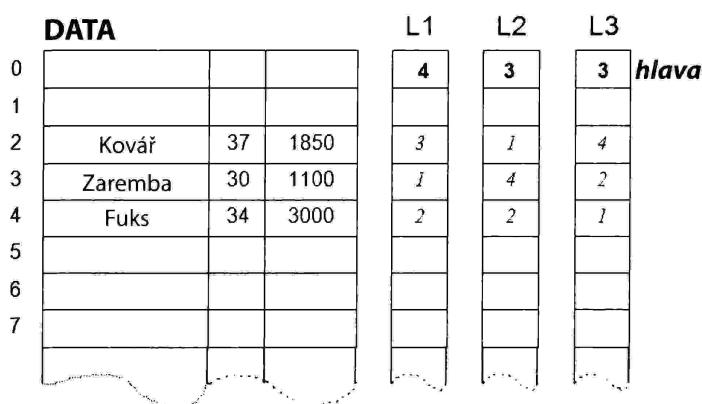
První prvek pole (tzn. prvek na pozici 0) slouží jako ukazatel na počátek seznamu. Je to tedy proměnná typu `hřáva`. Označíme-li pole symbolem `t`, pak položka `t[0].dalsi` obsahuje index prvního skutečného prvku seznamu. V našem příkladu se jedná o číslo 3, takže v položce `t[3].info` se nachází první prvek seznamu – jedná se o znak K. Abychom zjistili, co následuje po písmeni K, musíme načíst položku `t[3].dalsi`. Má hodnotu 2 a na tomto indexu je umístěno další písmeno slova „KOTEK“ – atd. Konec seznamu označujeme dohodnutou hodnotou -1 v položce `dalsi`.

Toto řešení vypadá elegantně i flexibilně. Obslužnou funkci takové datové struktury je možné napsat poměrně snadno. Lze přitom využít úplnou analogii s funkcemi, s nimiž jsme se již se-

tkali (např. funkcemi na obsluhu jednosměrných seznamů). Funkce tedy můžete naprogramovat v rámci samostatného cvičení.

Při té příležitosti bychom měli upozornit na jednu nevýhodu: samotná „syrová“ data jsou velmi úzce vázána na buňky, které simulují ukazatele. V případě seznamů to sice jinak nešlo, ale když využíváme pole, můžeme oba aspekty snadno oddělit. Jinými slovy: hodilo by se mít k dispozici samostatné pole na data i samostatné pole na ukazatele. Proč ale neudělat další krok a nepoužít pro ukazatele několik polí? Tím bychom se přiblížili k verzi představené na obrázku 5.8, avšak značně bychom si usnadnili realizaci.

Obrázek 5.12 znázorňuje miniaturní databázi, která je sdružena ve speciálním datovém poli.



Obrázek 5.12: Metoda „souběžných polí“ (2)

Vedle datového pole vidíme tři samostatná pole ukazatelů. Tyto ukazatele poskytují přístup k datům, která mají podobu seznamů setříděných podle různých kritérií. Pole DATA obsahuje záznamy s daty, přičemž první část informací je uložena v buňce data[2] a další záznamy se nacházejí v následujících buňkách. Proč se používá taková zvláštní struktura? Zajišťuje, že všechny buňky pole dat a pole ukazatelů (L1, L2 a L3, což jsou ve skutečnosti pole celých čísel) mají příslušný protějšek.

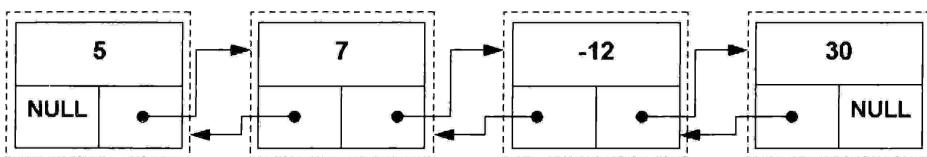
Buňky číslo 0 a 1 jsou v těchto tabulkách vyhrazeny pro ukazatel začátku seznamu a symbol konce seznamu. To znamená, že pokud položka L1[0] obsahuje číslo 4, víme, že prvním prvkem seznamu je položka data[4]. A který záznam následuje po ni? Samozřejmě L1[4]=2, z čehož vyplývá, že na druhém místě seznamu dat je prvek data[2]. Postupujeme-li stejným způsobem, můžeme sestavit celý seznam: data[4], data[2], data[3] – je zřejmé, že se jedná o seznam seřazený abecedně podle příjmení. Jak ale víme, že data[3] je poslední prvek seznamu? Prvek L1[3] totiž obsahuje hodnotu 1, která podle konvence slouží jako symbol konce seznamu.

Analogicky přijdeme na to, že pole L2 odpovídá seznamu setříděnému podle dvoučíselných kódů a L3 podle výdělků. Reprezentace seznamů pomocí polí, kde jsou data oddělena od ukazatelů, umožňuje uložit do stejné oblasti paměti několik seznamů současně – samozřejmě za předpokladu, že se zčásti skládají ze stejných prvků. V případech, kdy tento předpoklad platí, jedná se o cennou vlastnost, která pomáhá omezit spotřebu paměti. Kromě toho vadu polí, o které jsme se zmínili na začátku této části kapitoly (obsazení stálé oblasti paměti), lze snadno obejít šíkovnou dynamickou správou pole v definici třídy. Můžeme přitom využít tzv. vektory, což jsou objekty podobné polím, které se však mohou podle potřeby dynamicky zvětšovat. Srozumitelný popis vektorů naleznete například v publikaci [Eck00].

Seznamy jiných typů

Jednosměrné seznamy se velmi snadno používají a nezabírají příliš místa v paměti. Operace s nimi však občas trvají poněkud dlouho. Všimlo si toho hodně programátorů, a tak se objevily jiné typy seznamů, např.:

obousměrný seznam – pracovní buňka obsahuje ukazatele na předechozí i další prvek (viz obrázek 5.13):



Obrázek 5.13: Obousměrný seznam

- První buňka v seznamu nemá svého předchůdce, což označíme zapsáním hodnoty NULL do položky *predchozi*.
- Poslední buňka seznamu zase nemá svého následníka. Tento fakt symbolizujeme zapsáním hodnoty NULL do položky *dalsi*. Obousměrný seznam má poměrně vysoké paměťové nároky, ale důležitější než úspora paměti občas bývá rychlejší zpracování, které právě patří k výhodám obousměrného seznamu.

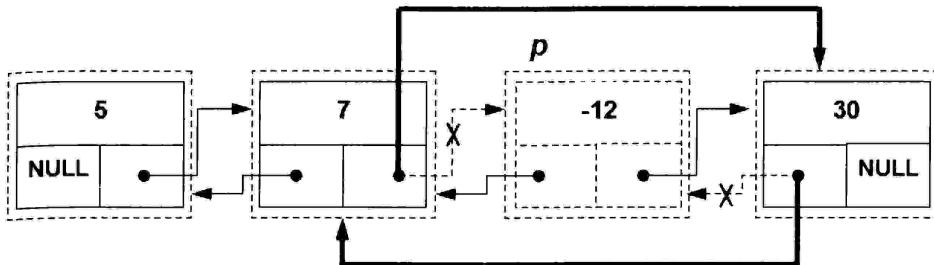
Obousměrný seznam se vyznačuje srozumitelnou vnitřní strukturou:

```
typedef struct rob
{
    int hodnota;
    struct rob *dalsi;
    struct rob *predchozi;
} PRVEK;
```

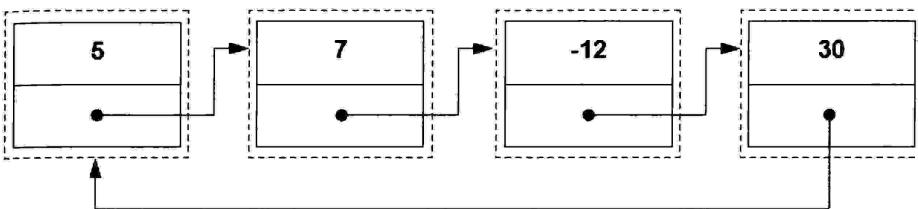
Předpokládejme nyní, že jsme při procházení prvků seznamu uložili ukazatel aktuální pozice p. (Například: Hledali jsme prvek, který vyhovuje určité podmínce, a hledání bylo úspěšné právě s ukazatelem p.) Jak můžeme prvek p ze seznamu odstranit? Jak si pamatujete z předechozího textu, ke správnému provedení této operace potřebujeme znát hodnotu ukazatelů pred a po, které odkazují na předechozí a následující buňku. V případě obousměrného seznamu se obě hodnoty již nacházejí v buňce, na kterou směřuje ukazatel p, takže je stačí přečíst:

```
void odstran_obousm(PRVEK *p)
{
    if(p->predchozi!=NULL) // není to první prvek
        p->predchozi->dalsi=p->dalsi;
    if(p->dalsi!=NULL)      // není to poslední prvek
        p->dalsi->predchozi=p->predchozi;
}
```

V závislosti na konkrétních okolnostech můžeme prvek p buď fyzicky odstranit z paměti pomocí instrukce `delete`, nebo jej v paměti ponechat pro případné další operace. Proceduru `odstran_obousm` znázorňuje obrázek 5.14 (potřebné úpravy ukazatelů jsou vyznačeny tučnou čarou).

**Obrázek 5.14:** Odstraňování dat z obousměrného seznamu

cyklický seznam – viz obrázek 5.15 – je uzavřen do kruhu: ukazatel posledního prvku směruje na „první“ prvek.

**Obrázek 5.15:** Cyklický seznam

Jeden z prvků označujeme jako „první“ spíše na základě dohody a slouží výhradně ke vstupu do „kouzelného“ kruhu cyklického seznamu.

Každý z výše uvedených seznamů má své výhody i nevýhody. V naší prezentaci jsme se snažili ukázat existující řešení, ale vybrat jedno z nich už musíte při psaní svého programu samostatně.

Zásobník

Koncepce zásobníku má značný význam v informatice a zvláště v systémovém programování¹². Toto tvrzení sice může znít poněkud hroznivě, ale ve skutečnosti se nemusíme ničeho obávat. Zásobník je stručně řečeno datová struktura, která usnadňuje řešení mnoha problémů algoritmické povahy, a právě na tento aspekt se zaměříme. Než se dostaneme k aplikacím zásobníku, pokusme se jej implementovat v jazyce C++.

Princip fungování zásobníku

Zásobník je datová struktura, která je přístupná pouze ze strany tzv. vrcholu neboli prvního volného místa, které se v této struktuře nachází. Princip fungování zásobníku se proto často popisuje anglickou zkratkou LIFO: *Last In First Out*, což znamená „nejdříve ten poslední“. K ukládání dat na vrchol zásobníku slouží obvykle funkce s názvem `push(X)`, kde X je hodnota určitého typu. Může se jednat o libovolnou jednoduchou či složenou proměnnou: číslo, znak, záznam atd.

Obdobně chceme-li načíst prvek ze zásobníku, použijeme k tomu funkci s názvem `pop(X)`, která do proměnné X zapíše hodnotu získanou na vrcholu zásobníku. Kromě toho, že plní svůj hlavní úkol, obě tyto základní funkce ještě vracejí kód chyby¹³. Je to konstanta celočíselného typu, která programátora informuje, zda náhodou nenastala nežádoucí situace, např. pokus načíst prvek

¹² Touto problematikou se v knize nebudeme zabývat.

¹³ Není to však povinné.

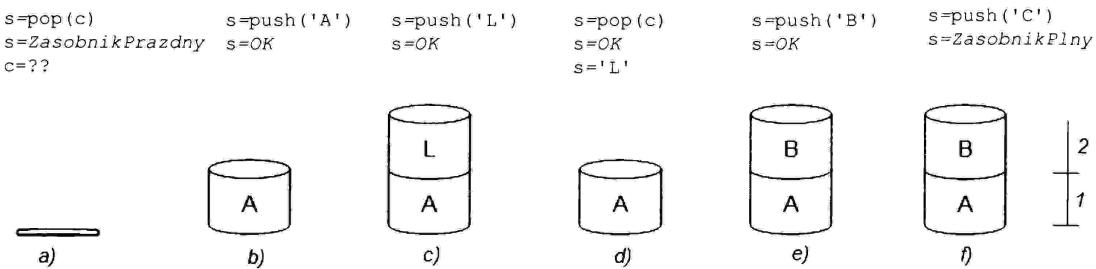
KAPITOLA 5 Datové typy a struktury

z prázdného zásobníku nebo uložit do zásobníku další hodnotu, když nebylo k dispozici volné místo (chyběla paměť). Programové realizace zásobníku se vzájemně liší drobnými detaily (poslední slovo má vždy programátor), ale celková koncepce se od našeho popisu příliš neodchyluje.

Princip fungování zásobníku lze tedy shrnout do následujících jednoduchých pravidel:

- Po provedení operace `push(X)` se prvek X stane novým vrcholem zásobníku a překryje předchozí vrchol (samozřejmě za předpokladu, že předtím zásobník nebyl prázdný).
- Jediným bezprostředně dostupným prvkem zásobníku je jeho vrchol.
- Pokus o vložení prvku do plného zásobníku vždy končí chybou.
- Pokus o načtení prvku z prázdného zásobníku vždy končí chybou.

Zásady fungování zásobníku a výše uvedená pravidla si můžeme ověřit na několika operacích se zásobníkem, které znázorňuje obrázek 5.16.



Obrázek 5.16: Zásobník a základní operace s ním

Obrázek představuje zásobník, který umožňuje ukládat znaky. Symbolické konstanty `ZasobnikPrazdny`, `OK` a `ZasobnikPlny` definuje programátor v modulu, který obsahuje deklaraci datové struktury typu zásobníku. Tyto konstanty se vyjadřují celočíselnými hodnotami (které nemají speciální význam). Ukázkový zásobník má kapacitu pouhých dvou prvků, což je samozřejmě absurdní, ale můžeme díky tomu snáze ilustrovat efekt přeplnění zásobníku.

Symbolický zásobník, který vidíme pod každou ze šesti skupin instrukcí, vždy znázorňuje stav po provedení příslušné skupiny instrukcí. Na první pohled vidíme, že operace se zásobníkem probíhaly úspěšně až do chvíle, kdy jsme dosáhli jeho maximální kapacity. Tehdy zásobník oznámil chybovou situaci.

Jak se zásobník obvykle implementuje? Nejoblíbenější způsob využívá pole a rezervuje jednu proměnnou, která zaznamenává, kolik se v zásobníku aktuálně nachází položek. Stejné schéma představuje obrázek 5.10, ale s jednou výhradou: i když víme, jakým způsobem je zásobník implementován, v žádném případě nesmíme povolit přímý přístup k jeho prvkům. Při všech operacích ukládání a vyjmání dat ze zásobníku je nutné používat metody `push` a `pop`. Rozhodneme-li se data i příslušné obslužné funkce zapouzdřit do třídy¹⁴, automaticky tím zajistíme bezpečné používání, které poskytuje samotná koncepce objektově orientovaného programování. Právě tento postup zvolíme.

Zásobníky lze realizovat mnoha způsoby. Je to dáno tím, že tato struktura se dokonale hodí k ilustraci mnoha algoritmických problémů. Pro naše účely se omezíme na velmi jednoduchou implementaci formou pole. Nejde o hotovou implementaci, ale můžeme ji považovat spíše za výchozí bod.

Vzhledem ke zmíněnému úmyslnému zjednodušení je definice třídy `ZASOBNIK` velice krátká:

¹⁴ Provedeme tzv. *hermetizaci*.

```
stos.h
const int DELKA_MAX=2;
enum stav {OK=0, ZASOBNIK_PLNY=1, ZASOBNIK_PRAZDNY=2};

template <class ZaklTyp> class ZASOBNIK
{
    public:
        ZASOBNIK() { vrchol=0; } // vrchol = první VOLNÁ buňka
        void clear() { vrchol=0; } // nulování zásobníku
        int push(ZaklTyp x); // zasobnik=t[0]...t[DELKA_MAX-1]
        int pop (ZaklTyp &w);
        int StavZasobniku();
    private:
        ZaklTyp t[DELKA_MAX]; // zasobnik=t[0]...t[DELKA_MAX-1]
        int vrchol;
}; // konec definice třídy ZASOBNIK
```

Náš zásobník by teoreticky mohl ukládat data různého druhu. Právě z tohoto důvodu je vhodné deklarovat tzv. *šablonu třídy*, kterou označuje klíčové slovo **template**.

Princip šablony třídy spočívá ve vytvoření vzorového kódu, ve kterém určitý datový typ (proměnné, hodnoty vracené funkcí atp.) není určen přesně, ale nahrazuje jej určitá symbolická konstanta. V našem případě se jedná o konstantu **ZaklTyp**.

Tento přístup je výhodný tím, že zajišťuje značnou univerzálnost nové třídy. Teprve v samotném kódu (např. ve funkci **main**) lze totiž určit, že např. parametr **ZaklTyp** bude nahrazen typem **float**, **char*** či nějakým složeným strukturálním typem. Nevýhoda šablony třídy spočívá v poměrně podivné syntaxi, kterou musíme při definování jejich metod dodržovat. Dokud se ještě definice nacházejí v těle třídy (tzn. mezi jejími složenými závorkami), syntaxe připomíná normální kód jazyka C++. Ovšem v okamžiku, kdy chceme definici metody umístit mimo třídu (tedy mimo složené závorky, které uzavírají její definici), dostáváme podivnosti tohoto typu¹⁵:

```
template <class ZaklTyp> int ZASOBNIK<ZaklTyp>::push(ZaklTyp x)
{
    if(vrchol<DELKA_MAX)
    {
        t[vrchol++]=x;
        return (OK);
    }
    else
        return (ZASOBNIK_PLNY);
}
```

Metoda **push**, jejíž kód jsme právě uvedli, je velmi jednoduchá, což je ostatně pro všechny implementace pomocí pole typické. Nový prvek **x** (libovolného typu) se zapisuje na vrchol zásobníku,

¹⁵ Vždycky se samozřejmě můžeme utěšovat, že by věci případně mohly být ještě komplikovanější. Ale žerty stranou: uvedené problémy mají jednoduchou příčinu. C++ patří do skupiny jazyků, jejichž komplátory potřebují přesně znát datové typy, které se v programu vyskytují. Všechny metody, které zdánlivě snižují „citlivost“ tohoto jazyka na datové typy, proto zákonitě musí být poněkud umělé. Při této příležitosti je vhodné připomenout, že existují jazyky, jejichž koncepce vůbec nezahrnuje datové typy – např. *Smalltalk-80* (jedná se o objektový jazyk s filozofií úplně odlišnou od jazyka C++, který vedle něj působí jako nějaký *objektový assemblér*).

KAPITOLA 5 Datové typy a struktury

na který odkazuje soukromá proměnná třídy `vrchol`. Poté probíhá inkrementace hodnoty vrcholu zásobníku – za předpokladu, že zásobník není zaplněn!

Metoda `pop` má opačnou úlohu – prvek odebraný ze zásobníku je uložen do proměnné `w` (která je při volání předána odkazem). Proměnná `vrchol` je přitom samozřejmě dekrementována. Musí být však splněna podmínka, že zásobník není prázdný:

```
template <class Zak1Typ> int ZASOBNIK<Zak1Typ>::pop(Zak1Typ &w)
{ // do proměnné "w" bude uložena hodnota vyzvednutá ze zásobníku
    if (vrchol>0)
    {
        w=t[--vrchol];
        return (OK);
    }
    else
        return (ZASOBNIK_PRAZDNY);
}
```

Někdy potřebujeme zkontrolovat obsah zásobníku, aniž bychom přitom měnili jeho obsah. Přitom se nám může hodit následující funkce:

```
template <class Zak1Typ> int ZASOBNIK<Zak1Typ>::StavZasobniku()
{
// vrací informace o stavu zásobníku
switch(vrchol)
{
    case 0          :return (ZASOBNIK_PRAZDNY);
    case DELKA_MAX :return (ZASOBNIK_PLNY);
    default         :return (OK);
}
}
```

Jakými dalšími způsoby lze zásobník definovat? Nemělo by nás překvapit, že z implementací pomocí polí logicky vycházejí dynamické struktury, například seznamy. Pokud bychom však integrovali seznam přímo do zásobníku, například místo pole `t` jako ve výše uvedeném příkladu, nebylo by takové řešení příliš efektivní. Stojí za to, abychom věnovali trochu času na vytvoření samostatné třídy od začátku.

Ještě chvíli se však budeme věnovat využití zásobníku. Zásadní význam má syntaxe použití šablony třídy ve funkci `main`. Zásobník `s`, který má sloužit k uchovávání proměnných třeba typu `char*`, můžeme deklarovat takto:

```
ZASOBNIK<char*> s;
```

Podobně postupujeme v případě kteréhokoli jiného datového typu. Stačí jej pouze umístit do špičatých závorek `< >`.

```
stos.cpp
int main()
{
    ZASOBNIK<char*> s1; // zásobník na ukládání textů
    ZASOBNIK<double> s2; // zásobník na ukládání čísel
    cout << "Vložení do zásobníku s1: ";

    for(i=0; i<3;i++)
        s1[i]=i;
    cout << "Obsah zásobníku s1: ";
    for(i=0; i<3;i++)
        cout << s1[i] << " ";
}
```

```

{
    if (s1.StavZasobniku() != ZASOBNIK_PLNY)
    {
        cout << "Pokus o vložení prvku:" << tab1[i] << ",";
        s1.push(tab1[i]);
    }
    else
    {
        cout << "Zásobník je plný, stop!\n";
        break;
    }
}

for(i=0; i<3;i++)
{
    char *z;
    if (s1.pop(z)==OK)
        cout << "\nVyjmutí ze zásobníku s1: " << z << endl;
    else
    {
        cout << "Zásobník je prázdný, konec!\n";
        break;
    }
}
}

```

Program poskytne následující výsledky (v archivu ke stažení je dostupná poněkud delší verze programu):

```

Vložení do zásobníku s1:
Pokus o vložení prvku:Ema, Pokus o vložení prvku:ma, Zásobník je plný, stop!
Vyjmutí ze zásobníku s1: ma
Vyjmutí ze zásobníku s1: Ema
Zásobník je prázdný, konec!

```

Fronty FIFO

Dalším datovým typem, kterým se budeme zabývat, jsou fronty typu FIFO (ang. *First In First Out*, což můžeme přeložit jako „nejdříve ten první“). Tato struktura funguje na stejném principu jako fronta lidí u pokladny v obchodě, samozřejmě s odhlédnutím od výjimek, jako jsou přednostně obsluhovaní klienti.

Stejně jako v případě zásobníku se jedná o datovou strukturu s omezeným přístupem. Předpokládá dvě základní operace:

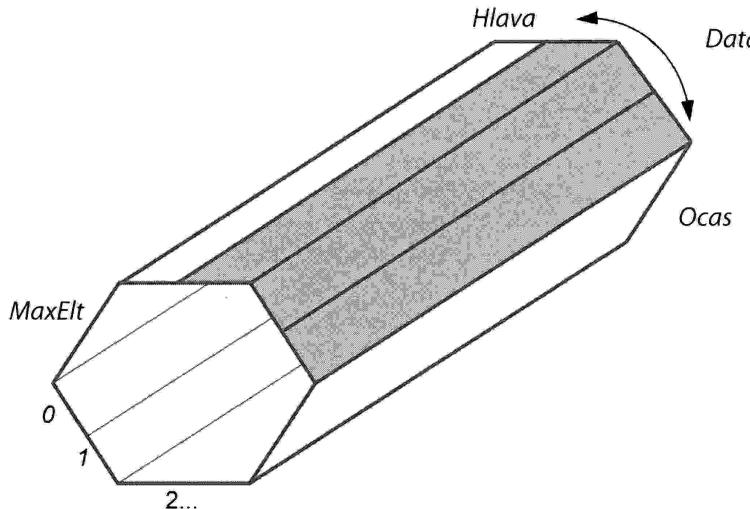
- **vloz** – umístí data (klienta) na konec fronty,
- **obsluz** – odstraní data (klienta) ze začátku fronty.

Fronty se v programátorské praxi nepoužívají tak často jako zásobníky. Některé úlohy algoritmické povahy můžeme však poměrně snadno řešit právě pomocí této datové struktury, a proto stojí za to, abychom se s ní seznámili.

Stejně jako mnoho jiných datových struktur lze také fronty implementovat několika možnými způsoby. Časově efektivní realizace pomocí jednosměrných seznamů připomíná implementaci, kterou

KAPITOLA 5 Datové typy a struktury

jsme popisovali v části o zásobníku. Nebudeme ji proto znova rozebírat a omezíme se jen na jednoduchou implementaci pomocí pole (která se principiálně blíží realizaci zásobníku pomocí pole). Implementaci fronty FIFO založenou na poli znázorňuje obrázek 5.17.



Obrázek 5.17: Realizace fronty FIFO pomocí pole – koncepce

Fronta se skládá z prvků mezi *hlavou* a *ocasem* – tyto dvě hodnoty budou samozřejmě soukromými proměnnými třídy FIFO. Po připojení nového prvku k frontě dochází k inkrementaci proměnné *ocas* a zapsání prvku na konec „šedé zóny“. V určité chvíli se samozřejmě může ukázat, že proměnná *ocas* dosáhla konce pole. Tehdy se fronta otočí vzhůru nohama.

Celá šedá zóna se pak otočí kolem nulového prvku pole. Chceme-li obsloužit „klienta“, který se aktuálně nachází na začátku fronty, potřebujeme si pamatovat první prvek a inkrementovat proměnnou *hlava*. Kromě toho si musíme ujasnit, jak budeme interpretovat výrok, že je fronta prázdná. Místo toho, abychom si komplikovali život složitým testováním obsahu pole, můžeme jednoduše prohlásit, že fronta je prázdná, když *hlava*=*ocas*. Současně musíme rezervovat jeden dodatečný prvek pole, který vzhledem k principu fungování metody *vloz* nikdy nebudeme používat. Po tomto podrobném vysvětlení by nás programová realizace fronty již neměla ničím zaskočit:

```
kolejka.h
enum stav {OK, CHYBA};
template <class ZaklTyp> class FIFO
{
    private:
        ZaklTyp *t;
        int hlava, ocas, MaxElt;
    public:
        FIFO(int n)
    {
        MaxElt=n;
        hlava=ocas=0;
        t=new ZaklTyp[MaxElt+1];
    }
}
```

```

void vloz(ZaklTyp x)
{
    t[ocas++]=x;
    if(ocas>MaxEl) ocas=0;
}

int obsluz(ZaklTyp &w)
{
    if (hlava==ocas)
        return CHYBA; // informace o chybě při operaci
    w=t[hlava++];
    if(hlava>MaxEl) hlava=0;
    return OK;
}

bool prazdne()
{ // je fronta prázdná?
    if (hlava==ocas)
        return true; // fronta je prázdná
    else
        return false;
}
};


```

Podobně jako v případě zásobníku jsme definovali nový datový typ pomocí šablony třídy. Díky tomu lze snadno definovat různé fronty, které budou obsluhovat odlišné datové typy. Definice třídy FIFO není úplná: chybí v ní například destruktor s veřejným přístupem a kontrola operací by mohla být poněkud pečlivější... Tato vylepšení můžete zpracovat samostatně jako jednoduchý domácí úkol. Podívejme se, jak se nová datová struktura používá v praxi:

```

kolejka.cpp
#include <iostream>
using namespace std;
#include "kolejka.h"
static char *tab[]={ "Kovářová", "Franěk", "Berka", "Pravec"};
int main()
{
    int i;
    FIFO<char*> fronta(5); // fronta s 5 prvky
    for(int i=0; i<4; i++)
        fronta.vloz(tab[i]);
    char *s;
    for(i=0; i<5;i++)
    {
        int res=fronta.obsluz(s);
        if (res==OK)
            cout << "Byl obsloužen klient: "<<s<<endl;
        else
            cout << "Fronta je prázdná!\n";
    }
}

```

KAPITOLA 5 Datové typy a struktury

V civilizovaných zemích mají při obsluze fronty přednost klienti, kteří přišli dříve. Totéž platí i v případě programové fronty, jak o tom svědčí výstup spuštěného programu:

```
Byl obslužen klient: Kovářová  
Byl obslužen klient: Franěk  
Byl obslužen klient: Berka  
Byl obslužen klient: Pravec  
Fronta je prázdná!
```

Haldy a prioritní fronty

V předchozích odstavcích jsme se mohli seznámit mj. se dvěma datovými strukturami, které představují svůj ideový protipól:

- *frontou* – nejdříve jsme odstraňovali „nejstarší“ prvek,
- *zásobníkem* – nejdříve jsme odstraňovali „nejmladší“ prvek.

Tyto datové struktury ze své podstaty slouží k uložení neuspořádaných dat. Veškeré operace s těmito strukturami jsou díky tomu rozhodně jednodušší. Následující datová struktura, kterou se budeme zabývat (prioritní fronta), je založena na úplně jiných principech, i když si zachovává výhodu zpracování neuspořádané množiny dat. (Tvrzení o neuspořádaných datech platí v globálním smyslu – jak se již zakrátko přesvědčíme, místní fragmenty haldy jsou v jistém konkrétním smyslu uspořádané.) Dvě základní operace s prioritními frontami spočívají ve:

- vkládání nového prvku
- odstraňování největšího prvku¹⁶

Jedna z nejjednodušších metod realizace prioritních front je založena na datové struktuře, která se označuje jako *halda* (ang. *heap*). O této datové struktuře jsme se již zmínili v souvislosti s tříděním (viz kapitolu 4). Nyní však toto téma rozvineme poněkud jiným způsobem.

Halda patří mezi binární stromy, k nimž se vzhledem k jejich speciálním vlastnostem ještě vrátíme. (Terminologická poznámka: Halda i prioritní fronty patří mezi datové struktury, ale pouze prioritní fronta má čistě abstraktní povahu.)

Třídění prvků přidávaných k haldě lze sledovat na obrázku 5.18, který představuje haldu se 12 prvků. Jedná se zároveň o příklad tzv. *úplného binárního stromu*. Jestliže trochu zjednodušíme definici, mohli bychom říci, že je to „strom bez dér“. Když se podíváme na čísla přiřazená uzlům stromu, vidíme, že jsou seřazena určitým charakteristickým způsobem: pod stávající uzly připojujeme nejvýše dva nové, dokud neumístíme všech 12 prvků. Dá se to samozřejmě vyjádřit poněkud formálněji, ale taková definice by rozhodně nebyla tak srozumitelná.

Z lineárního pořadí zaplňování stromu automaticky vyplývá, jakým způsobem jej budeme ukládat do pole¹⁷:

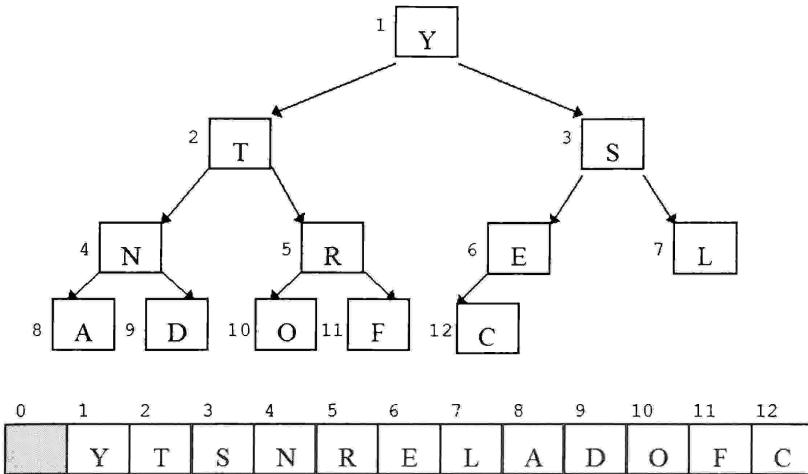
- Vrchol (tedy vlastně kořen, protože strom je obrácený) = 1.
- Levý potomek *i*-tého uzlu je umístěn na indexu $2*i$.
- Pravý potomek *i*-tého uzlu je umístěn na indexu $2*i+1$.

¹⁶ Budeme-li do prioritní fronty ukládat záznamy s konkrétní strukturou, bude jedna z položek záznamu určovat jeho prioritu vyjádřenou pomocí kladného či záporného celého čísla. V našich příkladech se pro jednoduchost omezíme jen na ukládání celých čísel.

¹⁷ Nulová buňka pole neslouží k ukládání dat.



Poznámka: Daný uzel může mít od 0 do 2 potomků.



Obrázek 5.18: Realizace fronty FIFO pomocí pole – příklad

Definovali jsme způsob ukládání dat, ale zatím jsme nevysvětlili, jaké mezi nimi panují vztahy. Halda se vyznačuje tím, že *každý uzel má větší hodnotu¹⁸ než oba jeho potomci – samozřejmě pokud existují*. Díky této struktuře stromu (a v důsledku i pole) lze snáze vkládat a odstraňovat prvky. Nový prvek můžeme totiž bez problému zapsat na konec pole. Pochopitelně nám to sice naruší dosavadní pořádek, ale pomocí poměrně prostých úprav můžeme poté vlastnosti haldy daného pole (stromu) zase obnovit. Ukažme na příkladu, jakým způsobem se tvoří halda z prvků 37, 41, 26, 14, 19, 99, 23, 17, 12, 20, 25 a 42, které postupně připojujeme ke stromu. Celý proces je znázorněn na obrázku 5.19.

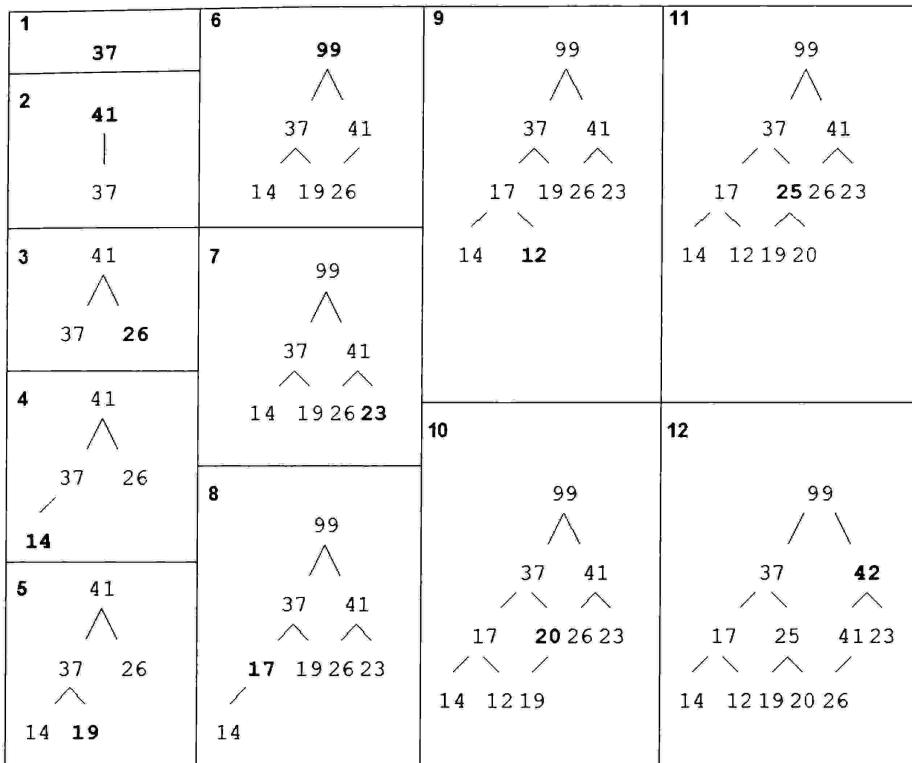
Na obrázku vidíme, kam putuje nový prvek (označený tučným písmem). Změny struktury stromu jsou snadno patrné při porovnání s předchozí fází. Předpokládejme, že na konec stromu doplníme číslo 99 (viz fázi 5). Strom již zahrnuje 5 prvků, takže nový prvek se v poli posune na pozici číslo 6 – pod číslo 26. V tomto okamžiku je ale porušena zásada konstrukce haldy: potomek uzlu má větší hodnotu než uzel, ke kterému je připojen. Jak postupovat, abychom obnovili pořádek? V takové situaci obvykle stačí, když zaměníme čísla 26 a 99 a problém se *lokálně* vyřeší. Všimněme si, že tato místní záměna obnoví pořádek jen na aktuálně analyzované úrovni a na následující úrovni jej může naopak poškodit! Chceme-li tedy zajistit, že bude podle pravidel seřazena celá halda, musíme pokračovat¹⁹ směrem nahoru, dokud nedosáhneme kořene. (V našem příkladu bude ještě nutné zaměnit čísla 99 a 41.) Výše popsané kroky provádí procedura s názvem Nahoru. Popsanou situaci ilustruje obrázek 5.20.

Když už nyní víme, CO to je halda a JAK se tvoří, musíme konečně vysvětlit, proč halda umožňuje snadno implementovat prioritní fronty. Řekli jsme, že prioritní fronty se od podobných datových struktur liší zejména tím, že nejdříve zpracovávají prvek, který má nejvyšší hodnotu (nebo v případě záznamů nejvyšší hodnotu určité vybrané položky).

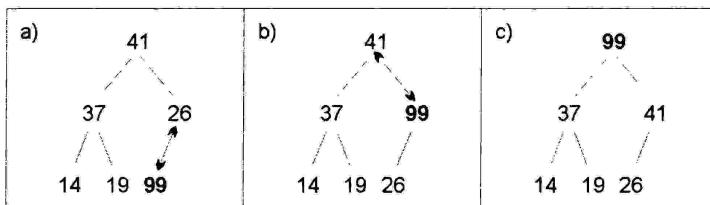
¹⁸ Existují i implementace, ve kterých hodnota není větší.

¹⁹ Je-li to samozřejmě potřebné.

KAPITOLA 5 Datové typy a struktury



Obrázek 5.19: Příklad konstrukce haldy



Obrázek 5.20: Správné vkládání nového prvku do haldy

Budemeli se držet analogie s frontou v obchodě, mohli bychom prohlásit, že všichni se způsobně řadí na konec fronty, ale pokladní si zákazníky prohlíží a nejdříve obsluhuje ty nejvíce privilegované (případně nejhezčí...).

U seznamů a běžných polí bychom měli problém právě s hledáním největšího prvku – museli bychom kvůli tomu prohledat celou strukturu, což vyžaduje čas úměrný hodnotě N (velikosti pole či seznamu). A jak je tomu v našem případě? Když se znova podíváme na pole na obrázku 5.18, můžeme se přesvědčit, že největší prvek vůbec hledat nemusíme, protože se z principu nachází v buňce pole s indexem 1!

Nadšení nás však poněkud přejde, když si uvědomíme, jak to bude s vkládáním. Prvky se sice vždy vkládají na konec, ale potom vždy musíme vyvolat proceduru Nahoru, která v haldě případně obnoví narušený pořádek. Nebude snad taková procedura natolik nákladná, že nás připraví o teoretické výhody, které plynou z použití haldy? Naštěstí se ukazuje, že nikoli. Časové nároky všech algoritmů pracujících s haldou jsou úměrné délce trasy, kterou je nutné projít binárním stromem

reprezentujícím haldu. Co platí pro tuto délku, víme-li, že binární strom je úplný? Například to, že libovolný vrchol je od kořenového vrcholu vzdálen nejvýše $\log_2 N$ uzlů! Díky tomu algoritmy založené na haldě obecně fungují v „logaritmickém“ čase. To je dobrý výsledek, který často rozdruhuje o výběru právě této datové struktury.

Po tomto dlouhém úvodu konečně přejděme k několika řádkům kódu v jazyce C++, které programůrům řeknou více než dlouhé vysvětlování. Třída `Halda`²⁰ je definována takto:

```
sterta.h
#include <iostream>
using namespace std;

class Halda
{
public:
    Halda(int nMax)
    {
        t=new int[nMax+1];
        L=0;
    }
    void vloz(int x);
    int obsluz();
    void Nahoru();
    void Dolu();
    void pis();
private:
    int *t;
    int L; // počet prvků
}; // konec definice třídy Halda
```

Konstruktor třídy tvoří pole, do kterého se budou ukládat prvky – `t[0]` se nepoužívá, a proto deklarujeme pole velikosti `nMax+1`, a nikoli `nMax` (tentot implementační detail je uživateli skrytý).

Nejdříve do haldy zkusíme vložit nový prvek:

```
void Halda::vloz(int x)
{
    t[++L]=x;
    Nahoru();
}
```

O proceduře `Nahoru` jsme se již zmínili: umožňuje obnovit uspořádání haldy po připojení nového prvku na konec pole `t`.

Obsah procedury `Nahoru` by nás neměl překvapit. Jediný rozdíl oproti obrázku 5.20 spočívá v tom, že chybí záměna prvků. V praxi se ukazuje, že rychlejší je přesunout prvky stromu, abychom vytvořili místo pro poslední prvek tabulky, který přenášíme nahoru:

```
void Halda::Nahoru()
{
    int temp=t[L];
    int n=L;
    while( (n!=1) && (t[n/2]<=temp) )
```

20 Kvůli zjednodušení uvádíme příklad haldy s celými čísly.

KAPITOLA 5 Datové typy a struktury

```
{
    t[n]=t[n/2];
    n=n/2;
}
t[n]=temp;
}
```

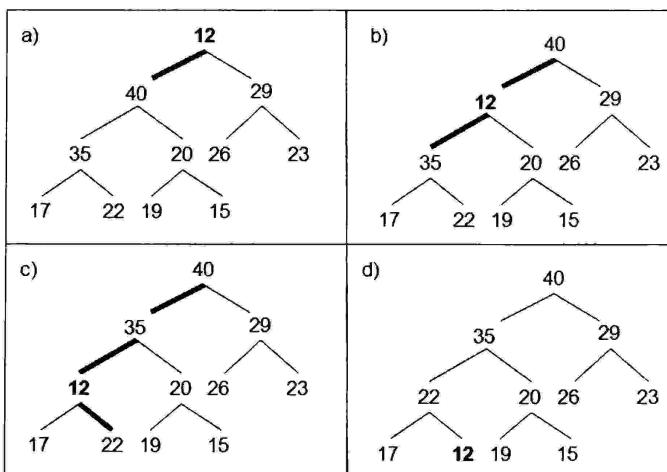
Když tento trik porovnáme s původním algoritmem, který spočívá v systematickém vyměňování prvků (v případě potřeby) při procházení uzlů stromu, zjistíme, že se bez něj můžeme obejít. Umožnuje však proceduru poněkud urychlit²¹.

Co se týče prioritních front, zmínili jsme, že se dají snadno implementovat pomocí haldy. Vložení „klienta“ na samotný konec prioritní fronty (čili haldy) jsme již provedli. Připomeňme, že jako první je v prioritní frontě obslužen „klient“, který má nejvyšší hodnotu – $t[1]$. Po odstranění tohoto prvku zůstane v poli mezera. Poslední prvek pole proto vložíme na místo kořene, dekrementujeme hodnotu L a vyvoláme proceduru `Dolu`. Tato procedura odpovídajícím způsobem upraví haldu, která po vložení prvku nemusí být uspořádaná správně:

```
int Halda::obsluz()
{
    int x=t[1];
    t[1]=t[L--]; // nekontrolují se chyby!
    Dolu();
    return x;
}
```

(Čtenáři by měli samostatně rozšířit výše uvedenou metodu a doplnit ji o základní kontrolu chyb.)

Jak by procedura `Dolu` měla fungovat? Změna hodnoty kořene mohla narušit rovnováhu jak u levého, tak i pravého potomka. Nový kořen je potřeba pomocí záměny s větším z jeho potomků přenášet ve stromu směrem dolů až do okamžiku, kdy pro něj nalezneme vhodné místo. Podívejme se, jaké výsledky poskytne procedura `Dolu`, kterou spustíme pro určitou haldu (viz obrázek 5.21).



Obrázek 5.21: Ukázka procedury `Dolu`

21 Procedura samozřejmě i nadále zůstává logaritmická – zázraky se nedějí ani v informatice!

Prvek 12 je označen tučným písmem. Pomocí silnější čáry je znázorněna cesta, kterou prvek 12 sestoupil na místo svého „posledního odpočinku“.

Proceduru `Dolu` lze realizovat takto:

```
void HAlda::Dolu()
{
    int i=1;
    while(1)
    {
        int p=2*i;           // levý potomek uzlu 'i' je (p), pravý potomek je (p+1)
        if(p>L)
            break;
        if(p+1<=L)          // pravý potomek nemusí nutně existovat!
            if(t[p]<t[p+1]) p++; // přechod na následující
        if(t[i]>=t[p]) break;
        int temp=t[p];       // výměna
        t[p]=t[i];
        t[i]=temp;
        i=p;
    }
}
```

HAlda se používá podobným způsobem jako dříve popsáne datové struktury, takže bychom při tom neměli mít žádné problémy. Poněkud zajímavější ukázka představuje efektivní využití haldy při... třídění dat.

Stačí totiž nejdříve pomocí metody `vloz` do haldy uložit libovolné pole, které chceme setřídit, a následně je zapsat pozpátku metodou `obsluz`:

```
#include "sterta.h"
int main()
{
    int i, tab[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
    HAlda s(14);
    for (i=0;i<14;i++)
        s.vloz(tab[i]);
    for (i=14;i>0;i--)
        tab[i-1]=s.obsLuz();
    cout<<"Setříděné pole:\n";
    for (i=0;i<14;i++)
        cout << " " << tab[i];
}
```

Jedná se samozřejmě o jednu z možných aplikací haldy – jednoduchou a efektivní metodu třídění dat, která je v průměru sotva dvakrát pomalejší než rychlé třídění, s nímž jsme se seznámili v předchozí kapitole (v části o algoritmu Quicksort).

Výše uvedenou proceduru lze dále urychlit vložením kódu metod `vloz` a `obsLuz` přímo do třídicí funkce, abychom se vyhnuli zbytečným a nákladným procedurálním voláním. V tom případě však potřebujeme nahlížet do soukromých dat třídy – pole `t` (viz soubor `sterta.h`). Třídicí procedura by tedy musela být spřátelenou funkcí. Tím bychom však porušili jeden z principů objektového programování (oddělení soukromého vnitřku třídy od jejího vnějšího rozhraní)!

Tuto cenu platíme za efektivitu – spřátelené funkce se do jazyka C++ dostaly jistě i s ohledem na použití tohoto jazyka při programování praktických aplikací namísto pouhé prezentace algoritmů

(jazyk Pascal je oproti tomu záměrně vybaven mechanizmy, které brání používat podivné triky, bez nichž by programy v reálných počítačích fungovaly příliš pomalu).



Tip: Viz také kapitolu 4, kde je uvedena stručnější verze samotného algoritmu třídění haldou.

Stromy a jejich reprezentace

Téma tzv. *stromů* by mohlo lehce zaplnit několik kapitol. Tato problematika je značně široká a kvůli různorodým aspektům souvisejícím se stromy je hodně těžké se rozhodnout, co vybrat a co vypustit. Nakonec převážila praktická hlediska: podrobně rozebereme otázky, které lze s vysokou pravděpodobností uplatnit v každodenní programátorské práci. Podrobnější rozbory stromů nalezneme ve většině knih, které se věnují teorii datových struktur. Tyto struktury však nejsou cílem samy o sobě (na to autoři knih o algoritmech často zapomínají). Autor tedy doufá, že většině čtenářů bude vyhovovat praktičtější přístup k tématu.

Úvahy ohledně stromů budeme ilustrovat hlavně pomocí nejoblíbenějších a nejčastěji používaných *binárních stromů*, které mají nezpochybnitelný význam při řešení nejrůznějších algoritmických úloh.

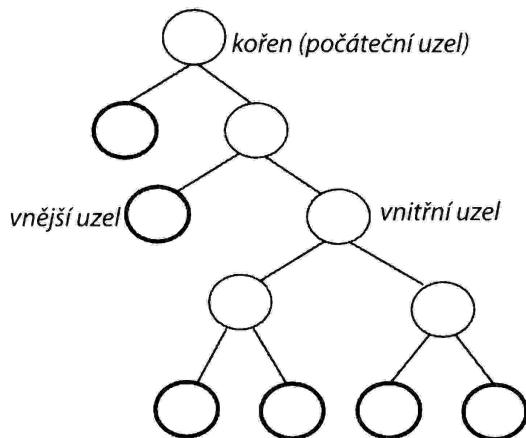
Stromy, které představují např. organizační struktury, systémy složek v operačním systému, rodičmeny atp., všichni dobře známe. Ze značného rozšíření stromů v různých oblastech praktického života je zřejmé, že se jedná o velmi užitečné struktury.

Strom je reprezentován *množinou uzlů*, které jsou vzájemně spojeny relací „rodičovského“ typu: podle konvence může z jednoho uzlu vycházet jeden nebo více následnických uzlů.

Speciální uzel stromu, od kterého se přímo či nepřímo odvozují všechny zbývající uzly, nazýváme *kořenem*. Uzel znázorňujeme podle potřeby kolečkem nebo jiným tvarem. Jednoduchý příklad představuje obrázek 5.22.

Čáry spojující uzly stromu mohou být doplněny šipkami, ale taková notace je v zásadě nadbytečná. Šipky na schématech stromů se obvykle vynechávají, protože směr procházení vyplývá z „rodičovského“ principu. Na druhou stranu šipky ničemu nevadí a občas je můžeme vidět.

Uzlům bez potomků říkáme *listy*, případně – již bez biologických analogií – *koncové uzly* (na obrázku 5.22 je takový uzel označen jako *vnější*).



Obrázek 5.22: Strom a související základní pojmy

Výškou uzlu označujeme délku nejdelší trasy, která vede od daného uzlu k listu.

Hloubka uzlu je délka trasy, která jej spojuje s kořenem.

Strom, ve kterém musí z uzlu vycházet pevný počet potomků, se nazývá *m-strom*. Nejznámějším příkladem takového typu stromu jsou binární stromy, ve kterých z daného uzlu vycházejí dva následnické uzly – levý a pravý. Strom, kde záleží na pořadí následných uzlů, se označuje jako *uspořádaný*.

V *m-stromech* se občas rozlišují dva druhý uzly: vnitřní a vnější; uzly druhého typu nemají žádné potomky. Všimněme si malé terminologické nuance: „list“ v *m-stromu* je ve skutečnosti vnitřním uzlem (!), jehož všechni potomci jsou vnější.

Je vhodné seznámit se s matematickými vlastnostmi binárních stromů, které usnadňují analýzu složitosti algoritmů založených na těchto strukturách:

- Binární strom obsahující n vnitřních uzlů má $n + 1$ vnějších uzlů.
- Binární strom s n vnitřními uzly má výšku alespoň $\lg n$ a nejvýše $n - 1$.

V kapitole 10 se setkáme se zobecněnou stromovou strukturou, která se nazývá graf. Tématu grafů věnujeme samostatnou kapitolu, takže se nyní nebudem pouštět do vysvětlování vztahů mezi grafy a stromy. Zatím se spokojíme s tvrzením, že všechny obecné případy se budou týkat stromů. Jestliže však ve stromu neurčíme kořen a neomezíme počet přiléhajících uzlů, najednou začínáme mluvit o grafové struktuře.

Cím jsou v praxi binární stromy? Jedná se o struktury, které se velmi podobají jednosměrným seznamům, ale navíc mají ještě jeden *rozměr* (neboli *směr*).

Základní buňka, která slouží ke konstrukci binárního stromu, vypadá takto:

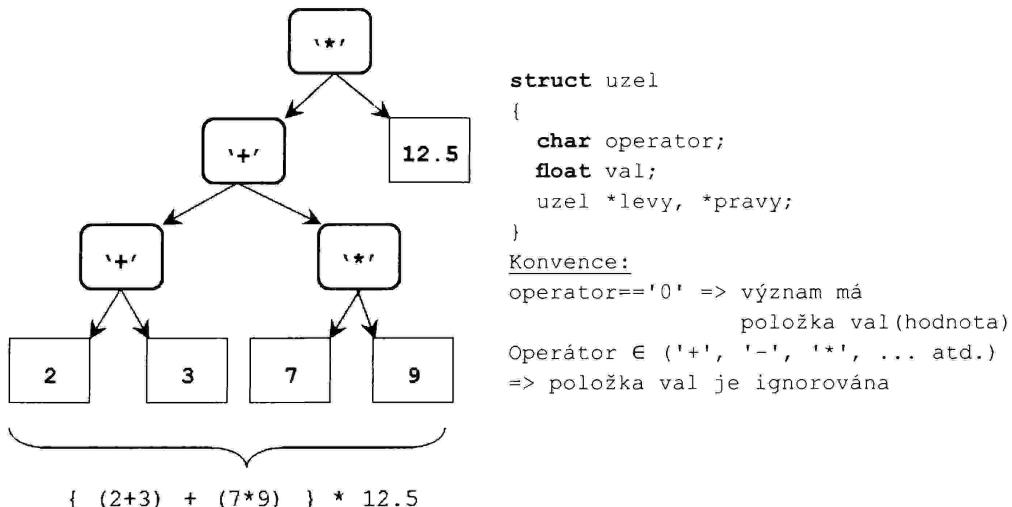
```
struct uzel
{
    int info; // neboli libovoľný jiný datový typ
    struct uzel *levy,*pravy;
}
```

Je zřejmé, že namísto jednoho ukazatele *dalsi* (jako v jednosměrném seznamu) pracujeme se dvěma ukazateli s názvy *levy* a *pravy*, které směřují na levou a pravou větev binárního stromu. Pokud náš algoritmus vyžaduje, abychom dokázali stromem procházet oběma směry (od kořene dolů k dalším následnickým uzlům i opačně), musí datová struktura obsahovat také ukazatel na „*předka*“, od nějž je následnický uzel odvozen, např.:

```
struct uzel
{
    int info; // neboli libovoľný jiný datový typ
    struct uzel *levy,*pravy; // následnické uzly
    struct uzel *predek; // předek
}
```

Jestliže stromy procházíme pomocí rekurze, nemusíme na uzel *predek* odkazovat. Rekurze si totiž ze své podstaty „pamatuje“, odkud jsme při procházení stromem přišli.

Chceme-li dobře pochopit, jak binární stromy fungují a proč jsou důležité, podívejme se na obrázek 5.23.



Obrázek 5.23: Binární stromy a aritmetické výrazy

Obrázek představuje jeden z možných příkladů uplatnění binárních stromů k reprezentaci algoritmických výrazů. K tomuto příkladu se ještě vrátíme v následujících částech kapitoly. Zatím se spokojíme s obecným popisem toho, jak lze takovou reprezentaci používat. Libovolný aritmetický výraz můžeme zapsat několika rozdílnými způsoby, které se liší umístěním operátorů: *před svými argumenty, po nich nebo klasicky mezi nimi* (pro jednoduchost zde samozřejmě předpokládáme, že všechny výrazy mají pouze dva argumenty).

Struktura dat na tomto obrázku patří mezi typické binární stromy. Obsahuje dvě položky určené k ukládání dat (*operator* a *val*) a také tradiční ukazatele na levou i pravou větev stromu, který je orientován kořenem nahoru. Dále předpokládejme, že v případě, kdy bude položka *operator* inicializována nějakou nesmyslnou hodnotou (zde 0, což není žádný známý operátor), položka *val* má nějakou smysluplnou hodnotu. Díky této duální reprezentaci můžeme v rámci pouhého jednoho typu záznamů snadno rozlišovat dva typy uzlů:

- *hodnoty* (list stromu),
- *aritmetické operátory*, které v obecném případě vytvářejí tři kombinace uzlů:
 - Levý a pravý potomek reprezentují výrazy.
 - Levý potomek je výrazem a pravý hodnotou.
 - Pravý potomek je výrazem a levý hodnotou.

Napíšeme-li odpovídající funkce, které budou obsluhovat výše uvedenou datovou strukturu podle předem určených pravidel, můžeme pomocí takové jednoduché reprezentace vyjadřovat libovolně složité aritmetické výrazy, počítat jejich hodnotu, derivovat je atd. Vše záleží jen na tom, čeho chceme dosáhnout – existuje mnoho různých aplikací. Jak navíc brzy ukážeme, při uplatnění rekurze budou obslužné algoritmy binárních stromů (a nejen jich) mnohem prostší a srozumitelně na první pohled.

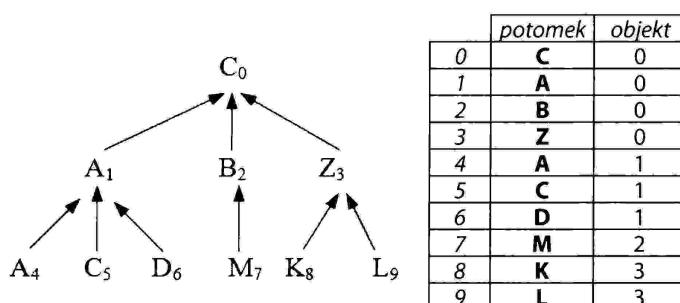
Je reprezentace pomocí rekurzivních datových struktur optimální? Na tuto otázku můžeme rozumně odpovědět jen s ohledem na konečné použití implementovaného stromu: jestliže nám příliš nezáleží na obsazené paměti a dáváme přednost snadné implementaci, může se ukázt, že reprezentace založená na poli je dokonce lepší než *klasická* reprezentace, kterou jsme představili výše.

Jak můžeme uložit strom do pole? Tento problém neřešíme poprvé. Již jsme se seznámili s jednoduchou metodou, která umožňuje uložit do pole jinou stromovou datovou strukturu – haldu, a s metodou tzv. *souběžných polí*, pomocí níž lze reprezentovat seznamy s mnoha třídicími kritérii. Jak je patrné, chytré použití polí nám může otevřít možnosti, jaké bychom těžko získali při práci s optimálními datovými strukturami typu seznamů.

Podívejme se nyní na příklad implementace stromů pomocí polí, kdy neukládáme informace o *potomcích* daného uzlu (tzn. nezajímá nás přechod směrem k listům), ale informace o *předcích* příslušného potomka.

S terminologií, která používá výrazy *otec*, *syn*, *levý potomek*, *pravý potomek* atd., se můžeme běžně setkat v různých knihách věnovaných stromovým strukturám – včetně knih v angličtině. Na tomto místě můžeme zmínit příhodu související právě s termíny tohoto typu, které mohou nepoučené osoby poněkud zmást. V roce 1993 jsem při svém pobytu ve Francii navštěvoval kurz angličtiny, který byl určen pro Francouze a vedla jej dosti výstřední Američanka. Během kurzu si měli studenti připravit krátké vystoupení na libovolné odborné téma. Jeden z francouzských studentů začal popisovat určitý algoritmus, který pracoval s distribuovanými databázemi. Hodně prostoru přitom věnoval vysvětlování stromové datové struktury, jejímž úkolem bylo reprezentovat jistá data, která příslušný algoritmus potřeboval. Při popisu stromu používal stejné termíny, jaké jsme uvedli výše: otec, syn, potomek atp. Rodilí mluvčí anglického jazyka obecně dosti citlivě rozlišují osobní formy (on, ona) od neosobních forem se zájmenem *it*, kterými označují v zásadě všechno kromě osob. Zmíněný student mluvil o něčem, co mělo nepochyběně neosobní charakter – datové struktury, ale občas používal výrazy, které se obvykle vyhrazují pro živé osoby – otec, syn... Američanka jeho přednášku poslouchala dobrých několik minut a přitom se dívala stále nechápavěji. Nakonec to nevydržela, vrhla se doprostřed třídy a skočila Francouzovi do řeči: „What father? What child? Please show me where is the *zizi*²² here!“ – a ukazovala přitom na stromovou strukturu nakreslenou na tabuli...

Vraťme se však k našemu tématu a předveděme nyní slibovanou implementaci stromů pomocí polí, která umožní získat informace o nadřazených uzlech (předcích). Obrázek 5.24 představuje strom, který umožňuje ukládat písmena (tzn. položka *val* má typ *char*).



Obrázek 5.24: Reprezentace stromu pomocí pole

Čísla vedle uzlů jsou pouze ilustrativní. Byla vybrána náhodně a neřídí se nějakými speciálními pravidly (pokud je s ohledem na konkrétní aplikaci sami nevymyslíme). Dále se dohodneme, že pokud se *predek[x]* rovná x, jedná se o první prvek stromu.

22 Tímto slovem francouzské děti označují typický atribut každého muže.

KAPITOLA 5 Datové typy a struktury

Když nyní víme, jak lze stromy reprezentovat pomocí dostupných mechanizmů jazyka C++ (nebo každého moderního programovacího jazyka), podívejme se na způsoby, kterými lze procházet větve stromů.

Binární stromy a aritmetické výrazy

Ve svém výkladu o stromech budeme vycházet z dosti rozsáhlého příkladu. Pomocí něj vysvětlíme jevy, s nimiž se programátor může setkat, a mechanizmy, se kterými musí pracovat, aby mohl nově poznanou datovou strukturu efektivně používat.

Jeden z aspektů problematiky naznačuje již obrázek 5.23. Na tomto obrázku jsme viděli, že strom se dokonale hodí k počítačové reprezentaci aritmetických výrazů. Přitom dovoluje velmi přirozeně ukládat nejen informace obsažené ve výrazu (tzn. *operandy* a *operátory*), ale i jejich logickou strukturu, kterou lze názorně představit právě v podobě stromu.

Znovu připomeňme, jak vypadá buňka, ve které lze – v souladu s ideou znázorněnou na obrázku 5.23 – uchovávat jak operátory (omezíme se zde na operátory +, -, * a operátor dělení vyjádřený dvojtečkou nebo lomítkem /), tak i operandy (reálná čísla).

```
struct vyraz
{
    double val;
    char op;
    vyraz *levy, *pravy;
};
```

Na způsobu iniciace takové buňky závisí, jak se bude později interpretovat její obsah. Pokud do polohy op umístíme hodnotu 0, budeme předpokládat, že buňka nepředstavuje operátor a smysl má hodnota, kterou jsme si zapamatovali v položce val. V opačném případě se budeme zabývat výhradně položkou op a tomu, co se nachází v položce val, nebudeme věnovat pozornost. Podívejme se na obrázek 5.25, který znázorňuje několik prvních fází výstavby binárního stromu pro aritmetický výraz.

Následující prvky:

1

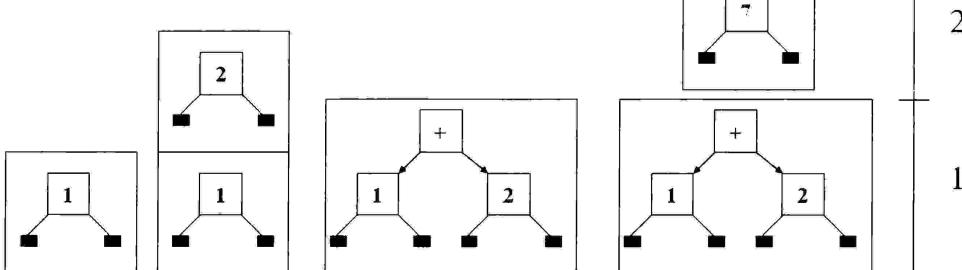
2

+

7

atd.

Fáze tvorby binárního stromu (obsah zásobníku):



Obrázek 5.25: Konstrukce binárního stromu pro aritmetický výraz

Při budování binárního stromu využijeme zásobník, který dobře známe z předchozí části kapitoly (viz stranu 127). Zásobník tentokrát umožní ukládat ukazatele na záznamy typu `struct vyraz`. Z toho vyplývá, že jej budeme deklarovat příkazem `ZASOBNÍK<vyraz*>` s. (Nyní vidíme, že mělo smysl vynaložit trochu úsilí a vytvořit zásobník v podobě šablony třídy.)

Typický aritmetický výraz zapsaný v běžné formě (označované jako infixová) lze vyjádřit také v tzv. *reverzní polské notaci* (RPN, postfixové). Místo toho, abychom psali a op b, používáme formu: a b op. Stručně řečeno: operátor následuje až po svých argumentech. Víme-li, kolik operandů daný operátor vyžaduje, lze snadno rekonstruovat i klasickou formu aritmetické operace.

Výraz bez závorek analyzujeme následujícím způsobem:

- Čteme argumenty po jednotlivých znacích a odkládáme je do zásobníku.
- Jakmile se objeví nějaký operátor, vyjmeme ze zásobníku příslušný počet argumentů. Výsledek operace pak umístujeme do zásobníku jako následující argument.

Výše popsaný proces můžeme na obrázku 5.25 sledovat v poněkud názornější podobě. První dva argumenty (čísla 1 a 2) nepatří mezi operátory, takže je odkládáme do zásobníku (spuštěný program tehdy vytvoří dvě paměťové buňky, jejich položky ukazatelů levý a pravý jsou inicializovány hodnotou NULL). Třetím načteným prvkem je operátor +. Vznikne nová paměťová buňka a vzhledem k tomu, že se objevil operátor, budou ze zásobníku vyjmuty dva argumenty – neboli buňky, které obsahují čísla 1 a 2. Tyto buňky jsou připojeny k položkám ukazatelů buňky, která obsahuje operátor +. Dále následuje číslo 7, které je uloženo do zásobníku, a celý proces může pokračovat.

Výše popsaným způsobem fungují komplátory, když pomocí zásobníku vyčíslují výrazy. Jediný rozdíl spočívá v tom, že do zásobníku neukládají následující dílčí větve, ale již vypočítané fragmenty aritmetických výrazů, které mohou být prakticky libovolně složité. Asi se shodneme, že z hlediska počítače je notace RPN velmi praktická²³.

Podívejme se na konkrétní instrukce jazyka C++, které zajišťují inicializaci binárního stromu.

```
wyrazen.cpp
typedef struct
{
    double val;
    char op;
}VAL;

int main()
{
    ZASOBNIK<vyraz*> s;
    // příklad SPRÁVNÉ posloupnost dat - v případě chybné
    // posloupnosti, která např. neobsahuje druhý operand, získáme
    // rovněž nesmyslný strom (jak se můžeme sami přesvědčit)
    VAL t[9]={{2,'0'}, {3,'0'}, {0,'+'}, {7,'0'}, {9,'0'}, {0,'*'}, {0,'+'},
              {12.5,'0'}, {0,'*'}};

    vyraz *x;
    for(int i=0;i<9;i++)
    {
        x=new vyraz;
        if((t[i].op=='*')||(t[i].op=='+')||(t[i].op=='-')
            ||(t[i].op=='/')||(t[i].op==':'))
            x->op =t[i].op;
        else
```

²³ Notace RPN sice vypadá poněkud podivně, ale navzdory tomu se v určitých oblastech využívá dosti často. Jako příklady lze uvést kalkulačky firmy *Hewlett Packard*, jazyk Forth či jazyk popisu tiskových stran Postscript, kterým jsou vybaveny některé laserové tiskárny. V jistých sférách je tato notace velmi rozšířená.

KAPITOLA 5 Datové typy a struktury

```
{  
    x->val=t[i].val;  
    x->op='0'; // dohodnutá konvence k označení hodnoty, nikoli operátoru  
}  
x->levy=NULL;  
x->pravy=NULL;  
if((t[i].op=='*')||(t[i].op=='+')||(t[i].op=='-')  
    ||(t[i].op=='/')||(t[i].op==':'))  
{  
    vyraz *l1,*p1;  
    s.pop(l1);  
    s.pop(p1);  
    x->levy =l1; // Připojení pod uzel x  
    x->pravy=p1; // Připojení pod uzel x  
}  
s.push(x);  
}  
pis_infix(x); cout << "=" << vypocitej(x) << endl; // viz následující popis  
pis_prefix(x);cout << "=" << vypocitej(x) << endl; // viz následující popis  
}
```

Datová sekvence v poli t uvedeného výpisu je *správná*, což znamená, že určitě vytvoří smysluplný binární strom. Budeme-li trochu experimentovat s obsahem pole, zjistíme, jak algoritmus zareaguje na chybnou datovou posloupnost. Můžeme předpokládat, že pokud bude například chybět druhý operand či operátor, dostaneme rovněž chybné výsledky. To je sice pravda, ale stojí za to, abychom se o tom přesvědčili sami.

Jak si můžeme prohlédnout obsah stromu, který jsme pečlivě „vypěstovali“? Tento úkol je možná nečekaně úplně triviální, protože využívá „topografické“ vlastnosti binárního stromu. Způsob interpretace formy výrazu (zda je *infixový*, *prefixový* či *postfixový*) totiž záleží pouze a výhradně na tom, jak procházíme větve stromu.

Podívejme se na realizaci funkce, která umožňuje vypsat strom v klasické (infixové) podobě. Funkce je založena na jednoduchém rekurzivním algoritmu:

```
vypis(v)  
{  
    jestliže výraz v je číslo, pak je vypiš;  
    jestliže výraz v je operátor op, pak vypiš prvky v tomto pořadí:  
        (vypis(v->left) op vypis(v->right))  
}
```

Programová realizace se pochopitelně doslovně drží výše uvedeného pseudokódu:

```
void pis_infix(struct vyraz *v)  
{ // funkce vypisuje výraz v infixové formě  
    if(v->op=='0') // číselná hodnota...  
        cout << v->val;  
    else  
    {  
        cout << "(";  
        pis_infix(v->levy);  
        cout << v->op;  
        pis_infix(v->pravy);  
        cout << ")";  
    }
```

```

    }
}
```

Analogickým způsobem můžeme realizovat algoritmus, který vypíše výraz bez závorek neboli ve formě RPN:

```

void pis_prefix(struct vyraz *v)
{ // funkce vypisuje výraz v prefixové formě
    if(v->op=='0') // číselná hodnota...
        cout<<v->val<<" ";
    else
    {
        cout << v->op << " ";
        pis_prefix(v->levy);
        pis_prefix(v->pravy);
    }
}
```

Na první pohled vidíme, že v závislosti na způsobu procházení stromu můžeme jeho obsah prezentovat různými způsoby. Strukturu vlastního stromu přitom nemusíme vůbec měnit!

Reprezentace aritmetických výrazů by samozřejmě nebyla úplná, kdybychom ji nerozšířili o funkce, které s těmito výrazy počítají. Než se však pustíme do výpočtů, potřebujeme ještě funkci, která dokáže zkontovalovat, zda má výraz ve stromu správnou syntaxi (tzn. zda například neobsahuje neznámý aritmetický operátor).

Všimněme si, že o správnosti stromu rozhoduje už samotný způsob, jakým jej konstruujeme pomocí zásobníku. Dodatečnou funkci, která kontroluje správnost stromu, však můžeme vytvořit velmi snadno – stačí k tomu pouhých pár řádků kódu – a o užitečnosti takové funkce nemůže být pochyb.

```

int spravne(struct vyraz *v)
{ // má výraz správnou syntaxi?
    if(v->op=='0')
        return 1; // OK, podle naší konvence se jedná o číslo
    switch (v->op)
    {
        case '+':
        case '-':
        case '*': // jsou to známé operátory
        case ':';
        case '/': return ( spravne(v->levy) * spravne(v->pravy) );
        default : return (0); // chyba, neznámý operátor!
    }
}
```

Výše uvedené funkce se nebude snažit převést do iterativní podoby – samozřejmě se to dá udělat, ale výsledný kód zrovna nepatří mezi zvláště čitelné a elegantní.

Konečně se dostáváme k prezentaci funkce, která bude počítat hodnotu aritmetického výrazu. Svou strukturou velmi připomíná funkci `spravne`:

```

double vypocitej(struct vyraz *v)
{
    if(spravne(v)) // je výraz správný?
        if (v->op=='0')
            return (v->val); // samotná hodnota
```

KAPITOLA 5 Datové typy a struktury

```
else
    switch (v->op)
    {
        case '+':return vypocitej(v->levy)+vypocitej(v->pravy);
        case '-':return vypocitej(v->levy)-vypocitej(v->pravy);
        case '*':return vypocitej(v->levy)*vypocitej(v->pravy);
        case ':':
        case '/':
            if(vypocitej(v->pravy)!= 0)
                return (vypocitej(v->levy)/vypocitej(v->pravy));
            else
            {
                cerr << "\nDělení nulou!\n";
                return -1; // primitivní kontrola chyb
            }
        }
    else
        cerr << "Syntaktická chyba!\n";
}
```

Po tomto poměrně dlouhém výpisu se můžeme podívat, jaké výsledky poskytne funkce main:

```
(12.5*((9*7)+(3+2)))=850
* 12.5 + * 9 7 + 3 2 =850
```

Se stromovými strukturami stojí za to si trochu pohrát, protože se jedná o zajímavé téma a výsledky bývají docela působivé.

Univerzální slovníková struktura

Výklad věnovaný stromovým strukturám zakončíme ukázkou konkrétní implementace tzv. *univerzální slovníkové struktury* (kterou budeme dále označovat zkratkou *USS*). Tento poměrně složitý příklad ukazuje, jaké možnosti stromové struktury nabízí. Čtenáři, kteří v praxi strukturu *USS* nebudou potřebovat, mohou informace a techniky z této části kapitoly využít při řešení jiných podobných problémů.

S ohledem na srozumitelnost výkladu budeme ve všech příkladech týkajících se struktury *USS* prozatím pomíjet otázku české diakritiky: písmen jako á, é či ü. Toto téma otevřeme teprve na konci podkapitoly, kde navrhнемe jednoduchý způsob řešení problému – ve skutečnosti postačí drobné a vlastně jen kosmetické úpravy algoritmů, které si zkrátka ukážeme.

Je nejvyšší čas vysvětlit, cím se vlastně budeme zabývat. Existuje mnoho programů různého zaměření, kterým je však společné, že zpracovávají uživatelsky zadáný text. Tyto programy mohou obsahovat funkci na kontrolu pravopisu zadávaných údajů (viz např. tabulkové procesory či textové editory). Programátoři často uvažují o tom, že by takovou funkci do svých programů mohli doplnit, ale vzhledem ke značné složitosti problému se o to nakonec nepokusí. S problematikou kontroly pravopisu úzce souvisí následující otázky, na které nelze jednoznačně a jednoduše odpovědět:

- Jaké datové struktury jsou vhodné k reprezentaci slovníku?
- Jak zapsat slovník na disku?
- Jak načíst „základní“ slovník do paměti?
- Jak aktualizovat obsah slovníku?

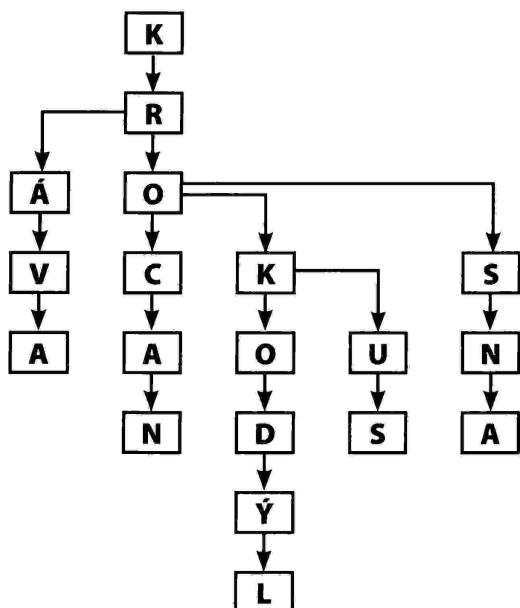
Princeznu a půl království tomu, kdo na tyto otázky dokáže okamžitě odpovědět! Nestačí samozřejmě odpovědět jen na některé z nich. Kdybychom totiž nevyřešili například problém zápisu na disk, nemá vůbec smysl uvažovat o těch zbývajících.

Slovníky všeho druhu se rovněž vyznačují značnými rozdíly. S obsazením velké části disku se sice můžeme snadno smířit, ale v případě operační paměti už rozhodování není tak jednoduché – pravopisný slovník střední velikosti může snadno zabrat celou dostupnou paměť, kde nezůstane místo pro samotný program. To nám nebude vadit jen v případě, že se spokojíme se zprávou: „Out of memory“²⁴... Situaci ještě komplikuje složitost naší materštiny s veškerým tím skloňováním, časováním a výjimkami z výjimek. Může se ukázat, že všechna data se bez vhodné komprese vůbec nedají uložit.

Existuje mnoho metod na komprezi dat, ale většina z nich má archivační charakter – slouží k ukládání dat a neumožňují s nimi dynamicky operovat. Ideální by bylo mít datovou strukturu, která z principu automaticky zajišťuje komprezi dat už v operační paměti a neomezuje dostupnost uchovávaných informací.

Každého už asi napadlo, že mezi datové struktury tohoto typu patří právě USS.

Struktura USS vychází z následujícího pozorování: mnoho slov má stejné kořeny a liší se pouze svým zakončením (příponami). Uvažujme například následující skupinu slov: KRÁVA, KROCAN, KROKODÝL, KROKUS, KROSNA. Pokud bychom je do paměti dokázali zapsat v podobě stroamu, který je znázorněn na obrázku 5.26, problém s kompresí bychom měli vyřešen. Místo původních 31 znaků jich nyní stačí uložit jen 19. Dosažená úspora nás možná neohromí, ale v případě rozsáhlých slovníků by mohla být mnohem větší. Pochopitelně předpokládáme, že ve slovníku budeme uchovávat hlavně řady slov, které začínají stejným písmenem – čili třeba všechny tvary jednotlivých podstatných jmen.



Obrázek 5.26: Kompreze dat je výhodou univerzální slovníkové struktury

KAPITOLA 5 Datové typy a struktury

Nyní se na onu tajemnou strukturu *USS* podíváme podrobněji. Její realizace je poněkud překvapivá. Obejdeme se totiž bez zapamatování slov i jejich fragmentů, ale přesto dosáhneme svého cíle! Program představíme na podrobně komentovaných úsecích kódu. První z nich obsahuje programovou realizaci struktury *USS*:

```
uss.cpp
const int n=29;

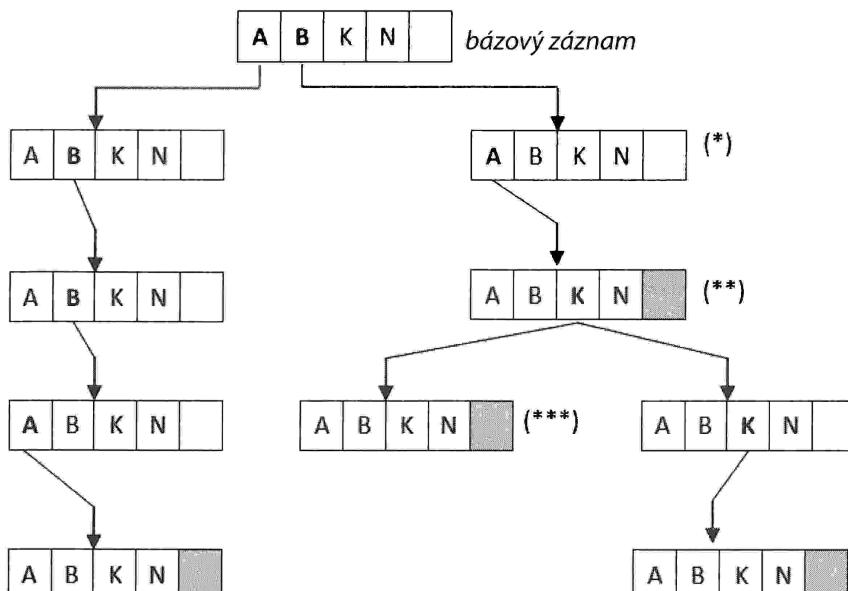
typedef struct slovnik
{
    struct slovnik *t[n];
} USS, *USS_PTR;
```

Máme zde pro jazyk C++ typickou deklaraci rekurzivního typu, jejímž jediným prvkem je pole ukazatelů na stejný datový typ. (Zní to podivně, ale nedá se nic dělat.) Písmenu *a* (nebo *A*) odpovídá buňka *t[0]* a písmenu *z* (či *Z*) analogicky buňka *t[25]*. Další paměťové buňky umožní ukládat speciální znaky, které sice nepatří mezi základní písmena anglické abecedy, ale často se v českých slovech vyskytují (jedná se např. o pomlčku či písmena s diakritikou).

Kvůli šetření místem budeme slova před uložením převádět tak, aby obsahovala pouze velká písmena. Slovo *odpovídá* zde má značný význam, protože struktura *USS* nezaznamenává slova bezprostředně.

Poznámka: Zacyklení ukazatele *t[n-1]* do svého vlastního pole označuje konec slova.

Princip fungování struktury *USS* můžeme vysvětlit na příkladu z obrázku 5.27.



Obrázek 5.27: Reprezentace slov ve struktuře USS

Při analýze budeme předpokládat, že abeceda se skládá z pouhých 4 písmen: A, B, K, N. Struktura *USS* obsahuje pole *t* délky 5 položek: poslední buňka slouží jako symbol konce slova. Jestliže uka-

zatel v buňce $t[4]$ směřuje na pole t , znamená to, že se na tomto místě příslušného slova nachází značka jeho konce. O které slovo se jedná? Vraťme se ještě k obrázku 5.26. Původní buňka po-skytuje přístup ke všem slovům z naší abecedy se čtyřmi písmeny. Ukazatel v buňce $t[1]$ (neboli $t['B']$) obsahuje adresu buňky označené jako (*). V ní je umístěn ukazatel $t[0]$ (čili $t['A']$), který směřuje na (**). Ovšem pozor! Ukazatel $t[4]$ v buňce (**) je zacyklený, tzn. nachází se zde symbol konce slova. Písmena tohoto slova lze přitom sestavit pomocí indexů, které jsme zatím prošli: nejdříve 'B', potom 'A' a posléze symbol konce slova, takže získáváme slovo BA.

Procházení stromu jsme však zatím neskončili: od buňky (**) vychází šipka k buňce (***)¹, kde také dochází k zacyklení. Jaké slovo jsme přečetli tentokrát? Samozřejmě slovo BAK. Podobným postupem můžeme přečíst ještě slova BANK a ABBA.

Myšlenku struktury USS, kterou lze bez grafického znázornění vysvětlit jen těžko, dokážeme programově realizovat překvapivě snadno. Na tomto místě pochopitelně nevytvoríme kompletní modul práce se slovníkem. Doplnění chybějících částí kódu (obsluha zápisu dat na disk, elegantní zobrazovací procedury atd.) však patří jen mezi běžné dokončovací práce.

Proberme nyní postupně procedury, které tvoří základní kostru modulu obsluhy USS.

Funkce `do_indexu` a `z_indexu` mají převodní úlohu. Z číselných indexů pole t (stavebního prvku záznamu USS) můžeme rekonstruovat písmena odpovídající konkrétním pozicím a naopak. Jestliže zvětšíme hodnotu konstanty n a poněkud upravíme tyto dvě funkce, zajistíme, že bude modul obsluhy USS podporovat česká písmena.

```
int do_indexu(char c)      // znak ASCII -> index
{
    if ( (c<='Z') && (c>='A') || (c<='z') && (c>='a') )
        return toupper(c)-'A'; // toupper = převod malých písmen na velká
    else
    {
        if (c==' ') return 26;
        if (c=='-') return 27;
    }
}

char z_indexu(int n)      // index -> znak ASCII
{
    if ( n>=0 && n<= ('Z'-'A') )
        return toupper((char) n+'A');
    else
    {
        if (n==26) return ' ';
        if (n==27) return '-';
    }
}
```

Funkce `zapis` přijímá ukazatel na první buňku slovníku. Než vytvoří novou paměťovou buňku, funkce zkонтroluje, zda je to skutečně nezbytné. Řekněme, že ve struktuře USS již existuje slovo *KOST* a nyní chceme do slovníku doplnit název černého ptáka se žlutým zobákem: *KOS*. Všechny úrovně odpovídající písmenům „K“, „O“ a „S“ již existují, takže není potřeba vytvářet žádné nové paměťové buňky. Pouze na úrovni písmene „S“ vznikne buňka, ve které bude na indexu $t[n-1]$ zapsán ukazatel „na sebe sama“. Připomeňme, že tento ukazatel funguje jako symbol konce slova.

KAPITOLA 5 Datové typy a struktury

```
void zapis(char *slovo,USS_PTR p)
{
    USS_PTR q; // pomocná proměnná
    int pos;
    for (int i=1; i<=strlen(slovo); i++)
    {
        pos=do_indexu(slovo[i-1]);
        if (p->t[pos] != NULL) p=p->t[pos];
        else
        {
            q=new USS;
            p->t[pos]=q;
            for (int k=0; k<n; q->t[k++]=NULL);
            p=q;
        }
    }
    p->t[n-1]=p; // cyklus jako konec slova
}
```

Funkce `pis_slovnik` umožňuje vypsat obsah slovníku – možná sice v nepříliš čitelné formě, ale není tak těžké se zorientovat, která slova jsou ve struktuře `USS` uložena.

```
void pis_slovnik(USS_PTR p)
{
    for (int i=0; i<26; i++)
        if (p->t[i] != NULL)
    {
        if ((p->t[i])->t[n-1]==p->t[i]) // na konci slova
            cout << z_indexu(i) << endl << " "; // zapsání znaku konce rádku
        else
            cout << z_indexu(i);
        cout << "---"; // kvůli přehlednosti
        pis_slovnik(p->t[i]); // rekurzivní výpis
    }
}
```

Funkce `hledej` realizuje celkem intuitivní algoritmus vyhledávání určitého slova ve stromu: projdeme-li všechny větve (úrovně) odpovídající písmenům hledaného slova a nalezneme symbol konce slova, je jasné, jak jsme dopadli.

```
void hledej(char *slovo,USS_PTR p)
{ // hledání slova ve slovníku
    int test=1, i=0;
    while ((test==1) && (i<strlen(slovo)) )
    {
        if (p->t[do_indexu(slovo[i])]==NULL)
            test=0; // příslušná větev chybí, slovo neexistuje!
        else
            p=p->t[do_indexu(slovo[i++])]; // vyhledávání pokračuje
    }
    if ( (i==strlen(slovo)) && (p->t[n-1]==p) && test)
        cout << "Slovo bylo nalezeno!\n";
    else
```

```

    cout << "Slovo se ve slovníku nenachází\n";
}

```

Následuje ukázková funkce main:

```

int main()
{
    int i;
    char obsah[100];
    USS_PTR p=new USS; // vytvoření nového slovníku
    for(i=0; i<n; p->t[i++]=NULL);
    for(i=1; i<=7; i++) // načtení 7 slov
    {
        cout << "Zadejte slovo, které chcete umístit do slovníku:";
        cin >> obsah;
        zapis(obsah,p);
    }
    pis_slovnik(p); // výpis obsahu slovníku
    for(i=1 ;i<=4;i++) // hledání 4 slov
    {
        cout << "Zadejte slovo, které chcete vyhledat ve slovníku:";
        cin >> obsah;
        hledej(obsah,p);
    }
}

```

Předpokládejme, že jsme při testování programu uvedli následující slova: kos, kost, kostra, krev, krysa, kULAtiNA, Kyvadlo (slova záměrně obsahují směs velkých i malých písmen). Po načtení této série slov by měl program vypsat obsah slovníku sice v dosti zvláštní, ale relativně čitelné podobě, která naznačuje, jak v tomto případě vypadá strom USS:

```

K---O---S
---T
---R---A
---R---E---V
---Y---S---A
---U---L---A---T---I---N---A
---Y---V---A---D---L---O

```

Množiny

Při programové implementaci matematických množin narázíme na řadu omezení, která souvisejí s použitým programovacím jazykem.

Příznivci jazyka Pascal nepochybňně znají definice tohoto druhu:

```

type Pismena = 'A'..'Z';
  MnozinaPismen = set of Pismena;
var Abeceda:MnozinaPismen;
  c: char;
begin
  Abeceda:=['A'..'Z'];
  read(c);
  if c in Abeceda then {atd.}
end.

```

KAPITOLA 5 Datové typy a struktury

To, co programátor v Pascalu považuje za množinu, z hlediska matematika samozřejmě množinou vůbec není. Definice množiny totiž předpokládá, že uložené prvky jsou *shodného typu*.

U jednoduchých aplikací se však konvence jazyka Pascal dokonale osvědčují, protože umožňují s množinami provádět například následující typy operací: přidávat do nich prvky, násobit je, odečítat, kontrolovat, zda do nich patří určitý objekt, atd.

V této knize popisujeme algoritmy a prezentujeme datové struktury pomocí jazyka C++, který svou roli plní poměrně dobře. Bohužel neobsahuje integrovanou obsluhu množin, a proto ji musíme explicitně doplnit. Přitom lze využít různé metody, které závisejí na aktuálních úkolech. Uvedeme jako příklad *implementaci množiny znaků*, kdy není nutné používat struktur *seznamů* a dynamicky přidělovat paměť. Předpokládejme, že počítač zpracovává „pouze“ 256 znaků (mimo jiné velká a malá písmena abecedy, čísla a tzv. netisknutelné řídicí znaky).

K simulaci množiny v tomto případě stačí nejjednodušší pole typu `unsigned char` jako v následujícím kódu:

```
set.cpp
class Mnozina
{
    private:
        unsigned char mnozina[256]; // celá tabuľka ASCII
    public:
        Mnozina()
        { // nulování množiny v konstruktoru
            for(int i=0;i<256;i++)
                mnozina[i]=0;
        }
        Mnozina& operator +(unsigned char c)
        { // přidání znaku 'c' do množiny a vrácení změněného objektu
            mnozina[c]=1;
            return *this;
        }
        Mnozina& operator -(unsigned char c)
        { // odstranění znaku 'c' z množiny a vrácení změněného objektu
            mnozina[c]=0;
            return *this;
        }
        bool patri(unsigned char c) // patří 'c' do množiny?
        {
            return mnozina[c]==1;
        }

        Mnozina& pridej(Mnozina s2) // přidání obsahu množiny 's2' do objektu
        {
            for(int i=0; i<256;i++)
                if(s2.patri(i)) // pokud se prvek nachází v s2,
                    mnozina[i]=1; // je přidán do množiny
            return *this; // vrácení upraveného objektu
        }

        int kolik() // vrací počet prvků množiny
    {
```

```

int n;
for(int i=0; i<256; i++)
    if(mnozina[i]==1)           // prvek je přítomen
        n++;
return n;
}

void pis()
{ // vypíše obsah množiny
    int i;
    cout << "{ ";
    for(i=0; i<256; i++)
        if(mnozina[i]==1)       // vypsání přítomného prvku
            cout << (char)i << " ";
    if(i==0)
        cout << "Množina je prázdná!";
    cout << "}\n";
}
}; // konec definice třídy Mnozina

```

Uvedená implementace je sice značně prostá, ale přesto již poskytuje typické operace s množinami:

```

int main()
{
    Mnozina s1, s2;
    s1=s1+'A'; s1=s1+'A'; s1=s1+'B'; s1=s1+'C';
    s2=s2+'B'; s2=s2+'B'; s2=s2+'E'; s2=s2+'F';
    cout << "Množina S1 ="; s1.pis();
    s1=s1-'C';
    cout << "Množina S1 - 'C' ="; s1.pis();
    cout << "Množina S2 ="; s2.pis();
    s1.pridej(s2);
    cout << "Množina S1 + S2 = ";
    s1.pis();
}

```

Po spuštění programu by se na obrazovce měly objevit následující zprávy:

```

Množina S1 = { A B C }
Množina S1 - 'C' = { A B }
Množina S2 = { B E F }
Množina S1 + S2 = { A B E F }

```

K operacím v uvedené implementaci třídy `Mnozina` můžeme samostatně doplnit průnik (součin) a odečítání množin.

Samozřejmě lze vytvořit obecnější implementaci množin, která bude přijímat data proměnné velikosti (tato implementace vyžaduje dynamické přidělování paměti, např. pomocí seznamů) a umožní ukládat složené prvky, například datové záznamy. Jestliže však do množiny potřebujeme ukládat pouze znaky abecedy, pak bychom si při návrhu třídy `Mnozina` šablonou třídy a seznamy zbytečně komplikovali práci.

Úlohy

Úloha 1

Zamyslete se, jak lze snadno upravit model *univerzální slovníkové struktury* (viz stranu 148), aby ji bylo možné použít jako dvoujazyčný slovník, např. česko-anglický. Odhadněte, jak vzrostou nároky slovníku (jedná se o množství obsazené paměti) pro následující data: 6 000 záznamů USS v paměti, které obsahují 25 000 uložených slov.

Úloha 2

Sada dosti podobných úkolů. Napište funkce, které budou odstraňovat:

- první prvek seznamu,
- poslední prvek seznamu,
- určitý prvek seznamu, který odpovídá vyhledávacím kritériím zadáným formou parametru funkce (abyste zajistili univerzálnost funkce, využijte metodu předání ukazatele na funkci jako parametru).

Úloha 3

Napište funkci, která:

- Vrátí počet prvků seznamu (a).
- Vrátí k -tý prvek seznamu (b).
- Odstraní k -tý prvek seznamu (c).



Upozornění: Při řešení úloh 2 a 3 si důkladně promyslete, jak efektivním způsobem informovat o chybových situacích (např. pokusu o odstranění k -tého prvku, když takový prvek neexistuje atd.).

Řešení úloh

Úloha 1

Úprava struktury USS:

```
typedef struct slovnik
{
    struct slovnik *t[n];
    char           *preklad;
} USS,*USS_PTR;
```

Překlad se do funkce zapis „připisuje“ (alokuje) při vyznačování konce slova – tímto způsobem neztratíme vztah *slovo-překlad*.

Náklady:

- bez druhého jazyka:
 - Náklady = ($n = 29 \cdot 4$ bajty („velký“ model paměti)) = 696 000 bajtů = asi 679 kB.
- s druhým jazykem:
 - Předpoklad: anglická slova mají průměrnou délku 9 bajtů + oddělovač, tedy 10 bajtů.

- Náklady = předchozí případ plus $25\ 000 \cdot 10$ plus určitý počet nevyužitých ukazatelů na překlady – odhadněme zhruba na 1 000. Nakonec dostáváme: $25\ 000 \cdot 10 + 1\ 000 \cdot 4 = 254\ 000$ bajtů, tedy kolem 248 kB.

V daném případě vzrostly nároky na obsazení paměti přibližně o 36 %.

Úloha 3

Návrh řešení úlohy 3a:

```
int cpt(PRVEK *q, int res=0)
{
    if (hlava==NULL)
        return res;
    else
        cpt(q->dalsi, res+1);
}
```

Ukázkové volání: `int mnozstvi=cpt(inf -> hlava);`

KAPITOLA 6

Odstraňování rekurze a optimalizace algoritmů

V této kapitole:

- Jak funguje kompilátor?
- Trocha formalizmu neuškodí
- Několik příklad odstraňování rekurze v algoritmech
- Odstraňování rekurze s využitím zásobníku
- Metoda opačných funkcí
- Klasické postupy odstraňování rekurze
- Shrnutí

Téma převodu rekurzivních algoritmů do jejich iterativní podoby – která samozřejmě musí být z funkčního hlediska ekvivalentní – logicky navazuje na kapitolu o rekurzi. Kdysi se programátoři tímto tématem zabývali výhradně kvůli jazykům, které neumožňovaly rekurzivní programování (FORTRAN, COBOL), ale i v současnosti může být obecný přehled o problematice docela užitečný.

Téma odstraňování rekurze nemá výhradně algoritmickou povahu. Proto je poněkud sporné, zda do knihy, která je věnována programovacím technikám, vůbec patří. Přesto je vhodné o tomto tématu něco vědět, protože odstraňování rekurze má nepochybně svůj praktický význam. Proč o takové operaci uvažujeme? Programy vyjádřené v rekurzivní formě jsou z podstaty věci dobře čitelné a poměrně krátké. Nemusíme mít kdovíjaké zkušenosti s programováním, abychom si domysleli, že iterativní verze stejných programů budou hůře čitelné a ještě k tomu delší. Z jakého důvodu se tedy vůbec pouštět do takové práce, která na první pohled nemá smysl?

Když věci postavíme tímto způsobem, samozřejmě nás to nijak nemotivuje. Dozvěděli jste se o několika podstatných výhodách, které souvisejí s využitím rekurzivních technik, a nyní tyto techniky chceme zase zavrhnout! Naštěstí není tak zle. Nikdo nás přece nenutí, abychom se rekurze úplně vzdali. Našim úkolem bude běžná optimalizace kódu, aby programy efektivněji fungovaly v reálných operačních systémech a počítačích.

Achillovou patou většiny rekurzivních funkcí je intenzivní využívání zásobníku, který umožňuje uchovávat „zmrazené“ exempláře stejné funkce. Pro každý takový dočasně neaktivní exemplář funkce je potřeba uložit kompletní sadu parametrů jeho volání a lokálních proměnných spolu s návratovou adresou. To zvyšuje nároky na obsazení paměti. Kromě toho nezapomínejme, že když musí kompilátor obsluhovat celý ten nepořádek, vyžaduje to cenný procesorový čas, takže takové programy obecně fungují pomaleji.

Můžeme tedy zvolit následující přístup: Při programování využijeme celou sílu i eleganci rekurzivních algoritmů, avšak když se dostaneme k závěrečné verzi kódu (té, kterou chceme nasadit

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

v praxi), provedeme transformaci na analogickou iterativní podobu¹. Vzhledem k tomu, že tento proces není pokaždě triviální, stojí za to seznámit se s několika standardními metodami, které se přitom uplatňují.

Výsledkem transformace je program, který z funkčního hlediska zcela odpovídá původní verzi. Z toho mimo jiné vyplývá, že pokud jsme si jistí správností daného rekurzivního programu, nemusíme již dokazovat správnost jeho iterativní varianty. Jinými slovy: dobrý rekurzivní algoritmus se po správné transformaci nezhorší.

Jak funguje kompilátor?

Strukturální jazyky, které obsahují mnoho vysoce abstraktních konstrukcí, by bez kompilátorů vůbec nemohly plnit svou roli. Kompilátory jsou také programy, které převádějí programový kód do podoby srozumitelné mikroprocesoru.

Poznamenejme ještě, že výsledek tohoto překladu příliš nepřipomíná to, co jsme s takovým úsilím napsali a spustili. Když zatracovaná instrukce **goto** (či v každém případě její varianty) se ve výsledném kódu objevuje mnohem častěji. Podívejme se například na strojový překlad² běžné podmíkové instrukce:

```
if (podminka)
    Instr1;
else
    Instr2;
```

Je to jednoduchá strukturální instrukce, kterou je ovšem potřeba provést sekvenčně:

```
if_not podminka goto et1
ASM(Instr1)
goto konec
et1:
ASM(Instr2)
konec:
```

Zápis **ASM(instr)** označuje posloupnost assemblerových instrukcí, které odpovídají instrukci **instr**, a **if**, **if_not** a **goto** jsou elementární instrukce procesoru (klíčová slova jazyka assembler).

Libovolnou strukturální instrukci lze přeložit na její sekvenční podobu (to je mimo jiné úkolem skutečných kompilátorů). Ani u procedurálních volání nejde o komplikovanou operaci, jak bychom se mohli domnívat. Posloupnost instrukcí:

```
Instr1;
P(x);
Instr2;
```

přeložená kompilátorem bude s jistým zjednodušením odpovídat následující sekvenci:

```
ASM(Instr1)
tmp=x
navr_adr=et1
goto et2
```

1 Za podmínky, že to vyžadují časové parametry naší aplikace nebo omezená kapacita paměti, kterou má k dispozici. Ve všech ostatních případech se odstraňování rekurze příliš nevyplatí.

2 Znázorněný – samozřejmě symbolicky – pomocí assemblerového pseudokódu.

```

et1:
    ASM(Instr2);
et2:
    ASM(P(tmp));
...
goto navr_adr

```

Lze výše uvedeným způsobem zpracovat také rekurzivní volání (kdy v proceduře P znova voláme proceduru P)? Fragment kódu odpovídající proceduře P samozřejmě nebudeme opakovat tolikrát, abychom obsloužili všechny exempláře dané procedury – to by bylo absurdní a v praxi neproveditelné. Zbývá nám pouze simulovat rekurzivní volání běžným vícenásobným použitím bloku instrukcí, který odpovídá proceduře P – ovšem s jednou výhradou: rekurzivní volání nemůže zastírat informace, které jsou nezbytné k tomu, aby mohl program rádně pokračovat ve své činnosti.

Výše uvedený způsob tuto podmínu bohužel nesplňuje. Podívejme se na následující ukázku rekurzivního programu³:

```

P(int x)
{
    Instr1;
    P(F(x));
    Instr2;
}

```

Jak můžeme odlišit návrat z procedury P, který zajistí její definitivní ukončení, od toho, který předává kontrolu instrukci Instr2? Ukazuje se, že tento úkol lze snadno automatizovat pouze jediným způsobem: pomocí tzv. zásobníku rekurzivních volání.

Kvůli řízení návratů z rekurzivních volání je nutné předem uložit dva údaje: tzv. kontext (např. hodnoty lokálních proměnných) a návratovou adresu, kterou dobře známe z předchozího příkladu. Při rekurzivním volání jsou tyto informace zaznamenány v zásobníku a řízení přebírá procedura. Pokud v jejím rámci dojde k dalšímu rekurzivnímu volání, budou do zásobníku uloženy další hodnoty kontextu a návratové adresy, které se liší od hodnot předchozí instance. Při návratu z rekurzivní procedury lze jednoduše obnovit předchozí stav kontextových proměnných jejich vyjmutím ze zásobníku.

Kompilátor „ví“, kde má dojít k návratu z procedury, protože v zásobníku byla uložena i adresa (argument instrukce goto⁴). Okamžik ukončení procedury lze určit kontrolou stavu zásobníku: když je zásobník prázdný, proběhla již všechna rekurzivní volání.

Předchozí příklad by bylo možné sekvenčně realizovat takto:

```

start:
    ASM(Instr1)
    push(Kontext, et1)
    x=F(x)           // procedura + rekurzivní
    goto start        // volání
et1:
    ASM(Instr2)
    if_not(ZasobnikPrazdny)
    {
        pop(Kontext, Adr)
        Obnov(Kontext)
    }

```

³ Funkce F označuje skupinu transformací parametrů funkce.

⁴ Stojí za zmínku, že instrukce goto existuje i v jazyce C++.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

```
    goto Adr          // návraty z rekurzivních  
}
```

Tím můžeme úvodní část zakončit. V další části kapitoly již představíme několik ukázek převodu rekurzivních algoritmů na iterativní.

Trocha formalizmu neuškodí

Základem této příručky jsou sice hlavně příklady, ale občas bychom si měli zatvářit poněkud „vědecky“ – a ke zdůraznění významu problematiky se ideálně hodí *Definice a tvrzení*. Tady jsou:

Definice 1: Iterativní procedura I je ekvivalentní rekurzivní proceduře P, pokud plní přesně stejný úkol jako procedura P a poskytuje přitom identické výsledky.

Například: Dvě následující procedury symetrie1 a symetrie2 můžeme považovat za ekvivalentní. Obě procedury se zabývají nepříliš užitečnou činností – vypisováním řetězců typu <<<<->>>> s šírkou, která závisí na parametru x.

```
void symetrie1(int x)  
{  
    if (x==0)  
        cout << "-";  
    else  
    {  
        cout << "<";  
        symetrie1(x-1);  
        cout << ">";  
    }  
}  
  
void symetrie2(int x)  
{  
    for(int i=1; i<=x; i++)  
        cout << "<";  
    cout << "-";  
    for(i=1; i<=x; i++)  
        cout << ">";  
}
```

Definice 2: Rekurzivní volání procedury P se označuje jako *koncové* (ang. *end-recursion*), jestliže po něm nenásleduje žádná instrukce této procedury.

Příklad:

```
void KoncRek(int n)  
{  
    if (x==0)  
        cout << ".";  
    else  
    {  
        cout << "A";  
        KoncRek(n-1);  
    }  
}
```

Poznámka: Rekurzivní volání procedury P obsažené v libovolném cyklu, např.:

```
void P(int n)
{
    ...
    while(V)
    {
        Instr1;
        P(n-1);
    }
}
```

se nepovažuje za koncové, protože v závislosti na podmínce V se může, ale také nemusí jednat o poslední volání P(n-1).

Tvrzení 1: Následující procedury P1 a P2 jsou ekvivalentní za podmínky, že P1 obsahuje pouze jedno koncové rekurzivní volání.

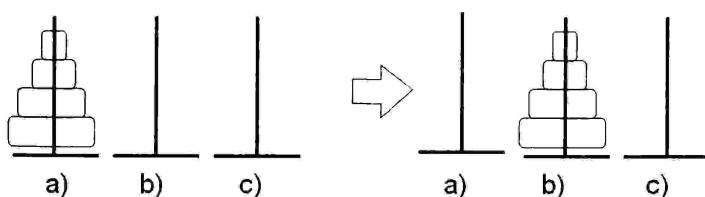
```
void P1(x)
{
    if (Podm(x))
        Instr1(x);
    else
    {
        Instr2(x);
        P1(F(x));
    }
}

void P2(x)
{
    while(!Podm(x))
    {
        Instr2(x);
        x=F(x);
        Instr1(x);
    }
}
```

Několik příkladů odstraňování rekurze v algoritmech

Vyzkoušejme nyní své nové vědomosti na klasickém příkladu tzv. *hanojských věží*. Původ tohoto hlavolamu vysvětluje legenda, kterou zde nebudeme uvádět. Místo toho se raději soustředíme na logický problém a jeho řešení.

Máme následující úlohu: Dostaneme n disků zmenšujícího se průměru. Uprostřed každého z nich je otvor, aby bylo možné disk nasadit na jeden ze 3 svislých kolíků. Na obrázku 6.1 vidíme situaci na začátku (vlevo) i na konci (vpravo) pro 4 disky.



Obrázek 6.1: Hanojské věže – představení problému

Naším úkolem je přemístit disky z kolíku označeného a na kolík b s využitím pomocného kolíku c. Přitom musíme dodržovat pravidlo, že nikdy nesmíme položit větší disk na menší. Předpoklá-

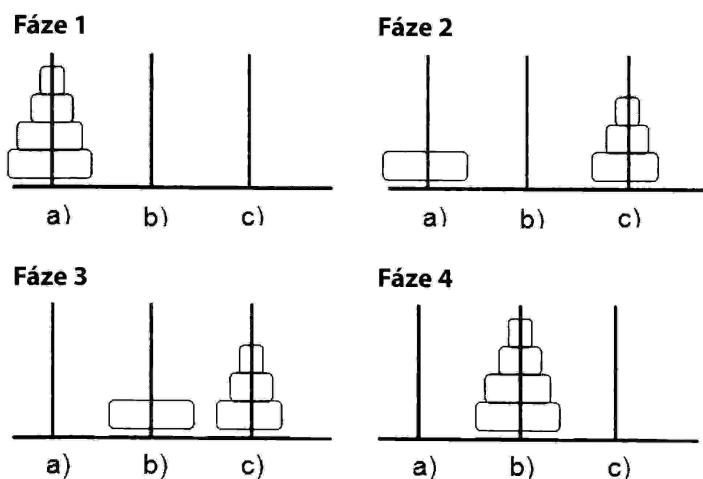
KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

dáme, že disk s číslem 1 má nejmenší průměr a disk s číslem n zase největší. Dále s ohledem na výsledný program označíme kolíky a, b a c čísly 0, 1 a 2.

Rekurzivní analýzou zadání dojdeme k následujícím závěrům:

- Máme-li k dispozici jeden disk, zadání se zjednodušuje na přemístění disku z kolíku a na kolík b (elementární případ).
- Jestliže máme přemístit $n \geq 2$ disků, pak za předpokladu, že dokážeme z jednoho kolíku na druhý přenést $n-1$ disků, můžeme úlohu převést na sérii přesunů, kterou symbolicky znázorňuje obrázek 6.2.

První fáze představuje výchozí situaci. Předpokládejme nyní, že jsme nějakým „kouzelným“ způsobem přenesli $n-1$ disků z kolíku a na kolík c. Na kolíku a zůstal největší disk, který má číslo n . V tomto okamžiku jsme se dostali k naprostu jednoduchému elementárnímu případu a už bez jakékoli další magie můžeme disk s číslem n přemístit z kolíku a na kolík b. Tímto způsobem jsme dospěli do situace, která je na obrázku označena jako *fáze 3*. Jak můžeme na základě uvedených informací hlavolam vyřešit?



Obrázek 6.2: Hanojské věže – způsob řešení

Využijeme zkušenosť z první fáze a budeme postupovat přesně stejným způsobem: vezmeme $n-1$ disků z kolíku c a tajemným způsobem je přemístíme na kolík b.

Nikoho, kdo již přečetl kapitolu 2, by zmíněný „tajemný způsob“ neměl nijak překvapit. Pochopitelně se jedná o aktivaci řady rekurzivních volání, která budou sledovat náš cíl a na základě předem daných pravidel vyřešit celý hlavolam.

Všimněme si, že vzhledem k výchozímu značení proměnných dostáváme vztah $a+b+c=0+1+2=3$ čili $c=3-a-b$. Procedura, která řeší problém hanojských věží, je tedy mimořádně jednoduchá:

```
hanoi.cpp
void hanoi(int n, int a, int b)
{
    if (n==1)
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
    else
    {
```

```

        hanoi(n-1, a, 3-a-b);
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
        hanoi(n-1, 3-a-b, b);
    }
}

```

Algoritmus je bohužel dosti nákladný, protože čas jeho realizace dosahuje až $(2^n - 1) \cdot t_e$, kde t_e udává čas přemístění jednoho disku mezi dvěma kolíky⁵.

Celkový čas nelze příliš ovlivnit (ze samotné povahy problému vyplývá, že řešení je dosti zdlouhavé). Můžeme však kompilátoru poněkud usnadnit tvorbu kódu tím, že eliminujeme druhé rekursivní volání, které splňuje podmínu uvedenou v *Tvrzení 1* (viz stranu 163). Na základě pravidla z tohoto tvrzení je možné kód procedury hanoi okamžitě upravit takto:

```

void hanoi2(int n, int a, int b)
{
    while (n!=1)
    {
        hanoi2(n-1,a,3-a-b);
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
        n=n-1;
        a=3-a-b;
    }
    cout << "Přesuň disk číslo 1 z " << a << " na " << b << endl;
}

```

Pomocí transformace popsané v *Tvrzení 1* lze docela snadno převést velkou skupinu procedur. Hodně procedur se navíc dá drobnými úpravami kódu přepsat do „transformovatelné“ podoby. Právě takový příklad budeme nyní analyzovat.

V kapitole o rekurzi jsme se mohli seznámit s programovou realizací funkce, která počítá faktoriál:

```

int faktorial(int x)
{
    cout << "x" << x << endl;
    if (x==0)
        return 1;
    else
        return x*faktorial(x-1);
}

```

Podáří se nám tuto funkci nahradit její iterativní verzí? První potíž spočívá v tom, že pracujeme se zkratkou. Vkládáme totiž rekursivní volání do rovnice, která vrací výsledek funkce. Nic nám však nebrání problematický rádek rozepsat, čímž dostaneme následující verzi (která je samozřejmě zcela ekvivalentní):

```

int faktorial(int x)
{
    if (x==0)
        return 1;
    else
    {
        int tmp=faktorial(x-1);

```

⁵ K tomuto výsledku se dostaneme snadno, ale kvůli čitelnosti výkladu příslušný postup přeskočíme.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

```
    return x*tmp;
}
}
```

Bohužel nám to příliš nepomohlo, protože rekuzivní volání není koncové? a nelze tedy uplatnit *Tvrzení 1*. Tuto překážku však můžeme snadno překonat, když provedeme další transformaci:

```
int faktorial(int x, int res1)
{
    if (x==0)
        return res;
    else
        faktorial(x-1,x*res);
}
```

Na první pohled vidíme, že jsme se vrátili k typu rekurze „s pomocným parametrem“, jejíž výhody jsme zdůrazňovali v kapitole o rekurzi. Neoznačují tedy termíny „koncová“ rekurze a rekurze „s pomocným parametrem“ stejný jev? Proč jsme se o tom tedy nezmínili dříve a místo toho jsme zavedli nové označení?

Odpověď bude možná znít poněkud šalamounsky: Tyto dva typy rekurze jsou totožné a zároveň nejsou. Když jsme zavedli termín rekurze s pomocným parametrem, uvažovali jsme o určité třídě problémů numerické či kvazinumerické povahy. Dá se to vyjádřit ještě přesněji: o skupině programů, které vracejí konkrétní výsledek, například číslo, pole, znakový řetězec atd. Tento výsledek získáme díky pomocnému parametru a odtud vyplývá i název metody. Oproti tomu koncově rekuzivní může být třeba procedura hanoi, která neposkytuje žádný „hmatačelný“ výsledek – jen návod na řešení hlavolamu. Spokojme se tedy s tímto vysvětlením a převeďme konečně funkci faktorial na její iterativní verzi. Neměli bychom narazit na žádné překvapení – kód lze přeložit téměř automaticky:

```
int faktorial_it(int x, int res=1)
{
    while (x!=0)
    {
        res=x*res;
        x--;
    }
    return res;
}
```

Odstraňování rekurze s využitím zásobníku

V této části kapitoly se seznámíme s novou metodou *odstraňování rekurze*, která je dosti kontroverzní. Budeme totiž nuteni svým způsobem porušovat zásady strukturovaného programování a navržená řešení navíc vůbec nebudou splňovat estetické nároky na programový kód. Příčinou je práce s pojmy na velmi nízké úrovni abstrakce, která se velmi přibližuje běžnému jazyku assembler. Vycházíme z jednoduchého principu: Když víme, jak kompilátor zpracovává rekuzivní volání, pokusíme se dělat totéž, ale přitom se vynasnažíme, abychom kompilátoru jeho úkol poněkud usnadnili. Máme totiž k dispozici něco, co současné kompilátory postrádají: naši inteligenci. Kompilátor je obyčejný program, který postupuje automaticky: textový soubor s kódem programu ve vysokoúrovňovém jazyce převádí na strojovou reprezentaci, jakou dokáže vykonávat procesor počítače. Programy zkoumá z hlediska syntaktické správnosti a nedokáže analyzovat jejich smysl

a cíl. My ovšem tyto znalosti máme a z toho vychází celá idea metody odstraňování rekurze s využitím zásobníku.

Tato metoda se dělí na dvě fáze:

1. záměnu lokálních proměnných na globální,
2. transformaci rekurzivního programu bez lokálních proměnných na jeho iterativní podobu.

Oba kroky podrobně rozebereme v následujících odstavcích.

Eliminace lokálních proměnných

Než začneme cokoli eliminovat, měli bychom si ujasnit, čeho se naše zásahy vlastně budou týkat. Lokální proměnné plní v strukturálním jazyce důležitou roli: umožňují formulovat algoritmy čitelným způsobem, a programátoři se tak nemusí obávat, že modifikují nějakou důležitou proměnnou z jiné části programu, jak se to stávalo v někdejším jazyce BASIC. Z tohoto hlediska může vypadat nelogicky, že někdo doporučuje návrat do téhoto prehistorických časů, kdy neexistovaly procedury, lokální proměnné, překrývání názvů ani jiné vymoženosti. Nic takového naštěstí nikdo nenavrhuje. Popsaný proces v žádném případě nepatří mezi programovací metody, ale jedná se o běžnou optimalizační techniku – v tom je zásadní rozdíl. Vraťme se tedy k lokálním proměnným a definujme, co jsou zač.

- Jako *lokální proměnnou* určité procedury P budeme označovat takovou proměnnou, kterou může modifikovat pouze příslušná procedura.
- *Globální proměnná* z hlediska procedury P bude taková proměnná, kterou lze změnit vně této procedury.

Každá proměnná deklarovaná uvnitř bloku jazyka C++, který je vymezen složenými závorkami { i }, se považuje za lokální proměnnou pro tento blok. V následující proceduře tedy najdeme dvě různé lokální proměnné var_loc a jednu globální proměnnou var_glob:

```
int var_glob;
void P()
{
    int var_loc;
    ...
    while(nějaká_podmínka)
    {
        int var_loc;
        ...
    }
}
```

Po tomto úvodu se můžeme podívat, jakým způsobem lze převést rekurzivní proceduru obsahující lokální proměnné `prom_loc` a jisté parametry volání `param_vol`⁶ na obdobně fungující proceduru, která však bude používat pouze globální proměnné (nová verze procedury P bude zároveň volána úplně bez parametrů).

Uvažujme dosti obecnou formu volání rekurzivní procedury P:

```
void P(param_vol)
{
    ...
}
```

⁶ Výrazy `prom_loc` i `param_vol` reprezentují seznamy proměnných, aby byl výpis kratší.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

```
F(param_vo1));  
...  
}
```

První fáze transformace spočívá v odstranění funkce F z volání P:

```
void P(param_vo1)  
{  
...  
param_vo1=F(param_vo1);  
P(param_vo1);  
...  
}
```

Stačí k tomu jednoduchá úprava formy kódu. Z proměnných `prom_loc` a `param_vo1` chceme udělat globální proměnné. Momentálně však při rekursivním volání jejich hodnotu mění následující exemplář procedury P! Jak si s tím poradíme? Musíme nějakým způsobem zachovat hodnoty `prom_loc` a `param_vo1`, aby – kromě eventuálních změn obsahu těchto proměnných během činnosti procedury P – byla situace předtím i potom stejná. Tento požadavek samozřejmě dokážeme splnit díky zásobníku:

```
void P()  
{  
...  
push(param_vo1)  
push(prom_loc)  
param_vo1=F(param_vo1);  
P(param_vo1);  
pop(prom_loc);  
pop(param_vo1);  
...  
}
```

Dosáhli jsme tedy svého cíle: zbavili jsme proceduru P všech lokálních parametrů a navzdory tomu tato procedura (stejně jako celý program) funguje stále stejně. Musíme však pamatovat na to, abychom nyní již globální proměnné `prom_loc` a `param_vo1` iniciovali správnými hodnotami⁷. Tím zajistíme, že přepracovaný program bude plně funkčně ekvivalentní s původní verzí. Při rozboru této metody je vhodné zmínit optimalizaci, která se okamžitě nabízí. V zásobníku stačí uchovávat pouze ty hodnoty lokálních proměnných, které potřebujeme. Konkrétně do zásobníku vůbec nemusíme ukládat ty lokální proměnné, které již po rekursivním volání nebudeme používat.

Na ukázku výše popsaného procesu znova analyzujme svůj klasický příklad *hanojských věží* (viz stranu 163). Jednoduchou transformací algoritmu získáme následující verzi:

```
void hanoi3()  
{  
while (n!=1)  
{  
push(n); push(a); push(b);  
n=n-1;  
b=3-a-b;  
hanoi3();  
pop(b); pop(a); pop(n);
```

⁷ Před prvním voláním procedury P.

```

    cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
    n=n-1;
    a=3-a-b;
}
cout << "Přesuň disk číslo 1 z " << a << " na " << b << endl;
}

```

Metoda opačných funkcí

Použití zásobníku – volání typu push a pop – se vyznačuje vysokými časovými nároky na obsluhu této datové struktury a zároveň vysokými nároky na paměť, kterou je nutné rezervovat pro do-statečně velký zásobník. Jak velký? Problém spočívá v tom, že to předem nevíme, takže musíme počítat s nejhorším případem. Proto jistě ochotně sáhneme po jakékoli metodě, díky níž se bez zásobníku obejdeme. Takovou metodu zakrátko představíme.

Nová technika má jeden zásadní nedostatek: nelze ji snadno formalizovat. Z praktického hlediska se jedná o to, že neexistuje jednoduchý návod, který bychom mohli alespoň částečně aplikovat automaticky⁸. Budeme muset zapojit svou představivost a intuici, a občas se dokonce smířit s tím, že řešení nelze najít. Nyní přejdeme k podrobnostem.

Ještě jednou připomeňme obecný tvar rekurzivní procedury:

```

void P1(param_vo1)
{
    ...
    param_vo1=F(param_vo1);
    P1(param_vo1);
    ...
}

```

Víme, že volání P(**param_vo1**) modifikuje (nebo může modifikovat) proměnné **prom_loc** a **param_vo1**. Aby k tomu nedocházelo, využívali jsme v předchozí metodě paměťové vlastnosti zásobníku. Princip této metody spočívá v tom, že do procedury P1 doplníme jisté instrukce. Protože víme, jak volání P1(**param_vo1**) mění proměnné **prom_loc** a **param_vo1**, můžeme do procedury vložit takové instrukce, které provedou opačnou činnost a tím obnoví původní hodnoty příslušných proměnných. Jinak řečeno, snažíme se převést program do tvaru:

```

void P2()
{
    ...
    (1)
    param_vo1=F(param_vo1);
    P2;
    OPACNA_FUNKCE(prom_loc,param_vo1);
    ...
}

```

Úkolem tajemné *opačná funkce* je zajistit, aby měly proměnné **prom_loc** a **param_vo1** v bodech programu označených (1) a (2) stejnou hodnotu. Jak toho dosáhnout? To je správná otázka. Odpověď bude pro každý program odlišná a nezbývá nám nic jiného než předvést nějaký konkrétní příklad.

⁸ To neznamená, že bez rozmyslu!

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

Následující procedura P1 počítá prvky smyšlené matematické posloupnosti:

```
odwrotna.cpp
void P1(int a,int& b)
{
    if(a==0)
        b=1;
    else
    {
        P1(a-1,b);
        // zde bude opačná funkce?
        b=b+a;
    }
}
```

Na matematickém smyslu této procedury nám nezáleží. V tomto okamžiku nás zajímá jedině to, jak ji převést na proceduru void P2, která bude fungovat identickým způsobem a přitom bude využívat pouze globální proměnné a a b.

V první fázi své analýzy musíme odpovědět na otázku: „Které proměnné se mění při rekursivním volání P1?“ Proměnná b má globální charakter, protože ji procedura P1 nemodifikuje. Tato proměnná slouží výhradně k předání výsledku z volané procedury. *Opačná funkce* tedy hodnotu proměnné b nemusí uchovávat. Mění se pouze proměnná a. Proceduře P1 se předává dekrementovaná hodnota proměnné a, zatímco po ukončení práce této procedury chceme mít k dispozici nezměněnou hodnotu proměnné a.

Při použití metody *eliminace lokálních proměnných*, s níž jsme se seznámili v předchozí části, bychom jednoduše museli zachovat původní hodnotu proměnné a v zásobníku. V tomto příkladu víme, že procedura P1 nemůže provést jinou změnu proměnné a než dekrementaci. Původní hodnotu proměnné a tedy jednoduše obnovíme tím, že ji inkrementujeme. A to je ta tajemná „opačná funkce“.

Podívejme se na upravený kód procedury:

```
int a,b; // globální proměnné!
void P2()
{
    if(a==0)
        b=1;
    else
    {
        a=a-1;
        P2();
        a=a+1;
        b=b+a;
    }
}
```

Zkontrolujme nyní, zda program opravdu funguje správně⁹:

```
int main()
{
```

⁹ Tento test samozřejmě nestačí jako formální důkaz ekvivalence procedur P1 a P2, ale úloha je natolik prostá, že zmíněný důkaz můžeme pominout.

```

for (int i=0; i<17;i++)
{
    P1(i,b);
    cout << b << " ";
}
cout << endl;
for (int i=0; i<17;i++)
{
    a=i;
    P2();
    cout << b << " ";
}
cout << endl;
}

```

Na obrazovce se objeví:

2	4	7	11	16	22	29	37	46	56	67	79	92	106	121	137
2	4	7	11	16	22	29	37	46	56	67	79	92	106	121	137

Výstup programu potvrzuje náš předpoklad, že procedury P1 a P2 jsou skutečně ekvivalentní.

Klasické postupy odstraňování rekurze

Výše uvedené metody eliminace lokálních proměnných z procedur a odstranění jejich parametrů měly jeden hlavní cíl: způsob provádění rekurzivních procedur co nejvíce přiblížit typickému iterativnímu programu. Cím se vlastně vyznačují programy, které označujeme jako „iterativní“? Tento termín se v zásadě týká systematického opakování určitých fragmentů kódu, např. pomocí instrukcí **for**, **while** či **do-while**. Z principiálního hlediska mají rekurzivní volání a iterativní způsoby vykonávání programu hodně společného (v obou případech jde o systematické opakování určitých činností). Prakticky se však tyto přístupy dosti liší. Iterace jsou založeny na kombinaci běžných instrukcí **goto**¹⁰ a testování podmínek. Rekurzivní volání se oproti tomu nachází minimálně o úroveň¹¹ výše. Díky odstranění lokálních proměnných a parametrů funkce jsme je značně přiblížili iterativnímu schématu.

Do rekurzivních procedur musíme doplnit určité testy, které v procesu rekurzivních volání kontroloují tzv. elementární případy¹².

Když například rekurzivně počítáme faktoriál čísla n , neustále ověřujeme, zda se číslo n rovná nule. Pokud je odpověď kladná, procedura vrátí hodnotu 1. V opačném případě dochází k dalšímu rekurzivnímu volání. V závislosti na splnění určitých podmínek se tedy vykoná jeden ze dvou odlišných fragmentů kódu. Oproti tomu při iteracích program obecně řečeno systematicky vykonává jisté neměnné fragmenty kódu. Tím se iterace liší od rekurzivních procedur.

Nabízí se tedy následující řešení: Do výkonné části iterativního kódu můžeme vložit podmínkové instrukce, které zajistí, že se kód prováděný v iteraci číslo i může lišit od kódu iterace $i+1$.

¹⁰ V různých variantách podle sady instrukcí procesoru.

¹¹ Abstrakce, složitosti apod.

¹² Tyto testy zajišťují, že bude řetězec rekurzivních volání dříve či později ukončen.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

Tímto způsobem se pokusíme najít způsob, jak odstranit rekurzi v kódech určitého typu, s jakými se při programování rekurzivními metodami můžeme často setkat.

Poznámka: Všechna dále analyzovaná schémata se týkají procedur, ze kterých jsme již odstranili parametry a lokální proměnné.

Schéma typu while

Další schéma, kterým se budeme zabývat, vypadá takto:

```
void P()
{
    while(podminka(x))
    {
        A(x);
        P();
        B(x);
    }
    C(x);
}
```

Při hledání ekvivalentní iterativní formy zapišme proceduru P v poněkud jiném tvaru s použitím instrukce `goto`. Díky této změně se zbavíme instrukce `while` (je potřeba přiznat, že značně umělým způsobem). Kromě toho zavedeme další globální proměnnou N – kterou jsme ostatně již dříve použili:

```
void P()
{
    N=0;
    start:
    if(podminka(x))
    {
        A(x);
        N++; P; N--;
        B(x);
        goto start;
    }
    else
        C(x);
}
```

Tato forma je nepochybně ekvivalentní, ačkoli zatím není jasné, k čemu je dobrá. Pustme se však do podrobnější analýzy činnosti tohoto programu a pokusme se sledovat sekvenční způsob volání skupin instrukcí, které symbolicky označujeme jako A(x), B(x) a C(x).

Ihned vidíme, že pokaždé, kdy je splněna podmínka instrukce `if... else`, určitě se provedou příkazy A(x) a N++. V případě, že podmínka splněna není, dojde k jednomu volání C(x). Tolik můžeme vyčíst z kódu vykonávaného před rekurzivním voláním P.

Co se ale děje během volání a návratů z rekurzivní procedury? Provádí se instrukce B(x) samozřejmě s příkazem N--. Pokud se nyní rozhodneme simulovat operaci rekurzivního volání a návratu pomocí příkazů N++, resp. N--, můžeme navrhnut následující ekvivalentní formu procedury P:

```

int N=0;
void P()
{
    do
    {
        while(podminka(x))
        {
            A(x);
            N++;
        }
        C(x);
        if(N==0)
            goto konec;
        N--;
        B(x);
    } while(N!=0);
    konec:
}

```

Čtenář, kterého tato úvaha nepřesvědčila, může najít matematicky přesnější důkaz správnosti výše uvedené transformace v knize [Kro89]. V této příručce jsem se rozhodl uvést méně formální vysvětlení – tím spíše, že pokud bychom u tematiky odstraňování rekurze zabíhali do přílišných podrobností, dostali bychom se dost daleko od algoritmiky a naopak bychom se mohli zaplést do programátorských triků, které vedou ke špatnému stylu.

Schéma typu if-else

Uvažujme nyní rekurzivní schéma, které představuje následující výpis:

```

void P()
{
    if(podminka(x))
    {
        A(x);
        P();
        B(x);
    }
    else
        C(X);
}

```

Předpokládáme-li N -násobné volání procedury P , můžeme si její činnost názorně představit jako sekvenci instrukcí:

$$\overbrace{A(x); \dots A(x)}^N; C(x); \overbrace{B(x); \dots B(x)}^N;$$

Tato forma zápisu algoritmu okamžitě naznačuje, že bychom ji mohli zapsat v iterativní formě... samozřejmě za podmínky, že známe hodnotu N . Počet volání procedury P bohužel nikdy předem neznáme – závisí výhradně na globálním parametru, který je uveden při jejím volání.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

Nepropadejme však předčasně pesimizmu a podívejme se na následující verzi procedury P:

```
int N=0;
void P()
{
    if podminka(x)
    {
        A(x);
        N++; P(); N--;
        B(x);
    }
    else
        C(x);
}
```

Řekněme, že činnost tohoto programu byla v určitém náhodně zvoleném okamžiku přerušena a pomocí ladicího programu jsme odečetli hodnotu N. Z kódu programu vyplývá, že globální proměnná N se inkrementuje při každém rekurzivním volání procedury P a při návratu z této procedury se dekrementuje. Pomocí této proměnné lze tedy odečíst aktuální úroveň rekurze procedury P¹³.

Další transformace procedury P je založena na této myšlence: Rekurzivní volání procedury P budeme simulovat pomocí skoku na její začátek. Při následném provádění procedury P můžeme velmi snadno ověřit, zda byla ukončena všechna předchozí volání – poznáme to z hodnoty N, ke které můžeme uvnitř procedury P kdykoli přistupovat¹⁴.

Následující úvahy bezprostředně vedou k další verzi programu:

```
int N=0;
void P()
{
    start:
    if podminka(x)
    {
        A(x);
        N++;
        goto start;
        navrat:
        N--;
        B(x);
    }
    else
        C(x);
    if (N!=0)
        goto navrat;
}
```

Instrukci goto bychom měli používat jen v případech, kdy k tomu máme zásadní důvody. Naše jednoduchá ukázka tento požadavek ovšem nesplňuje, protože kód můžeme snadno přepsat do strukturálního tvaru.

13 Viz kapitolu 2.

14 Pokud má proměnná N hodnotu 0, všechna dočasně přerušená volání již byla ukončena.

Dále uvádíme obě verze procedury P: původní i iterativní, kterou jsme tak dlouho hledali:

```
void P()
{
    if(podminka(x))
    {
        A(x);
        P();
        B(x);
    }
    else
        C(x);
}

int N=0;
void P()
{
    while(podminka(x))
    {
        A(x);
        N++;
    }
    C(X);
    while(N--!=0)
        B(x);
}
```

Zkontrolujme nyní, zda se výše popsaná transformace opravdu povedla. Přitom se vraťme k ukázkovému programu ze strany 170 (který je nyní zapsán v poněkud stručnější podobě).

```
odwrot2.cpp
void P2()
{
    if(a!=0)
    {
        a--;
        P2();
        b=b+(++a);
    }
    else
        b=1;
}
```

Když uplatníme popsanou transformaci, ihned získáme:

```
void P2_ITERAT()
{
    int k=0;
    while (a!=0)
    {
        a--;
        k++;
    }
    b=1;
    while (k--!=0)
        b=b+(++a);
}
```

Po spuštění programu se můžeme přesvědčit, že obě procedury jsou rovnocenné.

Schéma s dvojitým rekurzivním voláním

Poslední rekurzivní schéma, které představíme, se v praxi vyskytuje jen zřídka. Kromě toho důraz správnosti transformace je dosti složitý, takže se v dalším textu omezíme na analýzu hotového kódu a popíšeme příklad využití této transformace.

KAPITOLA 6 Odstraňování rekurze a optimalizace algoritmů

Obě následující formy algoritmu jsou ekvivalentní:

```
void P()
{
    if(podminka(x))
    {
        A(x);
        P();
        B(x);
        P();
        C(x);
    }
    else
        D(x);
}

int N=1;
void P()
{
    do
    {
        while(podminka(x))
        {
            A(x);
            N*=2;
        }
        D(x);
        while((N!=1)&&(N%2))
        {
            N=N/2;
            C(x);
        }
        if(N==1)
            goto konec;
        N=N+1;
        B(x);
    }while(p!=1);
}
```

Abychom metodu mohli předvést, vrátme se ještě jednou k problému *hanojských věží*, který jsme představili na straně 163. Když nasadíme metodu *opačných funkcí* (strana 169), snadno dojdeme k následující verzi procedury:

```
hanoi_it.cpp
int a,b,n;

void hanoi()
{
    if (n!=1)
    {
        n--; b=3-a-b;
        hanoi();
        n++; b=3-a-b;
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
        n--; a=3-a-b;
        hanoi();
        n++; a=3-a-b;
    }
    else
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
}
```

Všimněme si, že instrukce `n++` a `n--` se navzájem ruší, takže je můžeme odstranit. Jestliže proceduru hanoi převedeme na její iterativní verzi, měli bychom získat tento kód:

```
void hanoi_iter()
{
    int M=1;
    do
    {
        while (n!=1)
        {
            n--;
            b=3-a-b;
            M*=2;
        }
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
        while ((M!=1) && (M%2))
        {
            M=M/2;
            n=n+1;
            a=3-a-b;
        }
        if(M==1)
            goto KONEC;
        M++;
        n++;
        b=3-a-b;
        cout << "Přesuň disk číslo " << n << " z " << a << " na " << b << endl;
        n--;
        a=3-a-b;
    } while (M!=1);
    KONEC:
    ;
}
```

Po spuštění programu se můžeme přesvědčit, že obě procedury plní přesně stejný úkol a dávají identické výsledky.

Shrnutí

Kromě technik, které jsme uvedli v této kapitole, se k odstraňování rekurze algoritmů používají i jiné metody. Popsané techniky však nabízejí dostatečně širokou paletu možností, která by k transformaci většiny běžných rekurzivních procedur měla postačovat. Představené metody mohou také posloužit jako vzor k řešení úloh, které popsáným schématům zcela neodpovídají, ale poněkud se od nich liší.

KAPITOLA 7

Vyhledávací algoritmy

V této kapitole:

- Lineární vyhledávání
- Binární vyhledávání
- Hešování

S pojmem „vyhledávání“ jsme se v této knize v rámci příkladů a úloh již několikrát setkali. Koncepte vyhledávání je však natolik důležitá, že jí musíme věnovat samostatnou kapitolu. Abychom se neopakovali, téma popsaná v jiných částech knihy již nebudeme kompletne rozebírat znova, ale místo odkážeme na příslušné kapitoly. Podrobně probereme metodu *hešování*. Tematiku prohledávání textů s ohledem na její význam a specifika soustředujeme do následující kapitoly.

Lineární vyhledávání

Tématu lineárního vyhledávání jsme se již dotkli při ilustrování principu rekurze. Iterativní verze tehdy navrženého programu je triviální – k jejímu vytvoření nepotřebujeme ani znalosti kapitoly 6. Dále představíme příklad prohledávání pole celých čísel. Tato metoda samozřejmě funguje i v poněkud složitějších případech. Stačí pouze upravit funkci, která porovnává hodnotu x s aktuálně analyzovaným prvkem. Pokud pole obsahuje záznamy s komplikovanou strukturou, je vhodné použít univerzální funkci *hledej*, která bude jako parametr přijímat ukazatel na porovnávací funkci¹.

```
linear.cpp
int hledej(int tab[n], int x)
{
    int i;
    for(i=0; (i<n)&&(tab[i]!=x); i++);
    return i;
}
```

Nalezení čísla x v poli tab zjistíme pomocí hodnoty funkce. V případě čísla z intervalu $0\dots, n-1$ se jedná o index buňky, kde je číslo x uloženo. Vrácená hodnota n nás informuje o tom, že prvek x nebyl nalezen. Vzhledem k zásadám vyhodnocení logických výrazů v jazyce C++ máme zaručeno, že pokud program při analýze výrazu $(i < n) \&\& (tab[i] != x)$ zjistí, že první činitel logického součinu není pravdivý, zbytek výrazu už nebude kontrolovat (protože nemůže ovlivnit výsledek). Program tedy nebude testovat hodnoty mimo interval přípustných indexů pole. Tato vlastnost je o to cennější, že jazyk C++ na přehlédnutí tohoto typu nijak neupozorňuje.

¹ Použití ukazatelů na funkce jsme představili na příkladu v kapitole 5.

Typ vyhledávání, který je založen na obyčejné kontrole jednoho prvku po druhém, funguje značně pomalu. Jedná se totiž o algoritmus třídy $O(n)$. Lineární vyhledávání můžeme aplikovat tehdy, když nic nevíme o struktuře prohledávaných dat či případně o tom, jakým způsobem jsou uložena v paměti.

Binární vyhledávání

Jak jsme již naznačili v předchozím odstavci, prohledávání dat si můžeme mimořádně usnadnit, máme-li informace o tom, jak se uchovávají. V praxi se často setkáváme s daty, která jsou již v paměti počítače usporádána, např. se záznamy seřazenými podle abecedy, do neklesající posloupnosti podle určité položky záznamu atp. Budeme tedy předpokládat, že pole je setříděné. V praxi se jedná o dosti častý případ, protože lidem vyhovuje, když jsou informace organizované. V tomto případě můžeme aplikovat své „metaznalosti“ a prohledávání dat si zjednodušit. Z prohledávání lze například snadno vyloučit ty oblasti pole, kde se prvek x určitě nemůže nacházet. S podrobně vysvětlenou ukázkou *binárního vyhledávání* jsme se již setkali v kapitole 2 – viz úlohu 2 a její řešení. Na tomto místě můžeme pro změnu uvést iterativní verzi algoritmu:

binary-i.cpp

```
int hledej(int tab[], int x)
{ // vrácení indexu hledané hodnoty nebo čísla -1
    enum {ANO, NE} Nalezeno=NE;
    int left=0, right=n-1, mid;
    while( (left<=right) && (Nalezeno!=ANO) )
    {
        mid=(left+right)/2;
        if(tab[mid]==x)
            Nalezeno=ANO;
        else
            if(tab[mid]<x)
                left=mid+1;
            else
                right=mid-1;
    }
    if(Nalezeno==ANO)
        return mid;
    else
        return -1;
}
```

Proměnné mají naprostě stejné názvy i význam jako v předchozí verzi programu. Proto je vhodné oba výpisy alespoň jednou porovnat. Musíme ještě vysvětlit, jakým způsobem se volí střední prvek (`mid`). V našich příkladech se konkrétně jedná o střed aktuálně prohledávané oblasti. Ve skutečnosti to však samozřejmě může být libovolný index mezi indexy `left` a `right`. Snadno si však můžeme všimnout, že rozdelením pole na polovinu prohledávanou oblast omezíme nejvíce. Neúspěšný výsledek vyhledávání signalizuje vrácená hodnota `-1`. V případě úspěšného hledání funkce tradičně vrací index prvku v poli.

Binární vyhledávání patří mezi algoritmy třídy $O(\log_2 N)$ (viz část „Logaritmický rozklad“ kapitoly 3). Abychom si dobře uvědomili výhody tohoto algoritmu, uvažujme konkrétní číselný příklad:

Lineární vyhledávání umožňuje v čase úměrném rozměru pole (seznamu) odpovědět na otázku, zda se v dané struktuře nachází prvek x . V případě pole s 20 000 prvky bychom tedy při hledání odpovědi na uvedenou otázku museli v nejhorším případě provést 20 000 porovnání. U binárního vyhledávání ve stejně situaci stačí pouze $\log_2 20\,000$ (přibližně 14) porovnání.

Tento výpočet dokonale ilustruje, jak je binární vyhledávání výhodné.

Hešování

Než se začneme zabývat tématem hešování², musíme přesně určit, kde se tato metoda uplatňuje. Metoda se používá tam, kde předem známe maximální počet prvků patřících do určitého obooru³ R (E_{max}), zatímco všech možných (rozdílných) prvků z tohoto obooru může potenciálně existovat velmi mnoho (C). Tolik, že v praxi lze sice přidělit paměť pro pole délky E_{max} , ale v případě pole pro všech potenciálních C prvků z obooru R by se to fyzicky nedalo provést.

Proč je tak důležité, abychom si uvědomili fyzickou nemožnost přidělení takového pole? Kdybychom totiž dokázali takové mimořádně rozsáhlé pole vytvořit, mohli bychom v něm vyhledávat prvky běžným přímým adresováním a všechny vyhledávací funkce by patřily do třídy $O(1)$. Jak si pamatujeme z kapitoly 3, notace $O(1)$ z definice označuje, že počet operací prováděných algoritmem nezávisí na rozsahu problému. Pokud budeme mít k dispozici jistou funkci H , která nám umožní najít hledaný záznam ve víceméně stejném konečném čase, na velikosti prohledávané množiny teoreticky nebude záležet.

Uvedme příklad, abychom celý princip objasnili:

- Chceme uložit $R_{max} = 250$ slov délky 10 znaků (plně nám postačuje pole velikosti $250 \cdot 11 = 2\,750$ bajtů⁴).
- Různých slov této délky může teoreticky existovat $C = 26^{10}$ (přitom neuvažujeme písmena s českou diakritikou!). Přidělit paměť poli, do kterého by se všechna tato slova dala umístit, prakticky není možné.

Princip hešování spočívá v tom, že se snažíme najít takovou funkci H ⁵, která by na základě určité množiny dat předané formou parametru poskytla index v poli T , kde se příslušná data nacházejí (pokud jsme je tam předtím uložili).

Jinak řečeno: hešování je založeno na přiřazení:

data → adresa buňky v paměti.

Za předpokladu takového uspořádání dat by umístění nového záznamu do paměti *teoreticky* nemělo záviset na umístění dříve uložených záznamů. Jak si jistě vzpomínáme z kapitoly 5, netýkalo se to setříděných seznamů, binárních stromů, haldy atp. Z nového způsobu ukládání dat přirozeně vyplývá, že se značně zjednoduší proces vyhledávání. Když totiž hledáme prvek x , který se vyznačuje určitým klíčem⁶ v , můžeme k tomu využít jistou funkci H . Výpočet $H(v)$ by měl poskytnout konkrétní paměťovou adresu, kde můžeme zkontrolovat, zda prvek x existuje. Celý proces vyhledávání je pak u konce!

2 Píše se také ve tvaru „hašování“.

3 „Oboru“ v matematickém smyslu.

4 Jeden dodatečný bajt pro znak „\0“, kterým je ukončen textový řetězec jazyka C++.

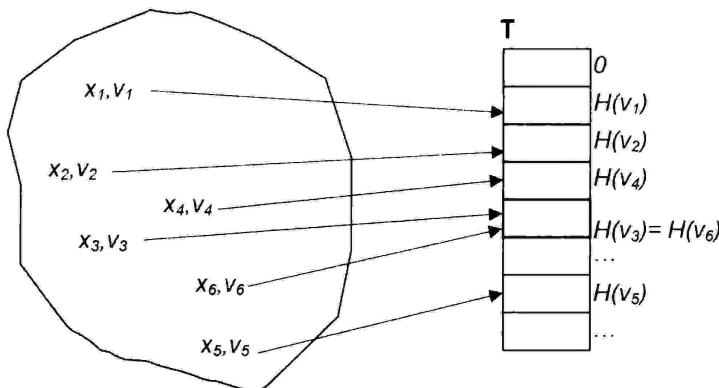
5 Dříve se bylo možné setkat i s názvem „rozptylovací funkce“.

6 Pojem „klíče“ pochází z teorie databází a často se používá i v jiných oblastech informatiky. Klíč označuje množinu atributů, které jednoznačně identifikují záznam (neexistují dva různé záznamy, které by měly stejnou hodnotu kličových atributů).

KAPITOLA 7 Vyhledávací algoritmy

Myšlenka hešování je znázorněna na obrázku 7.1.

C - obor všech klíčů



Obrázek 7.1: Princip hešování

Předpokládejme, že z celého oboru klíčů C nás zajímá výhradně šest hodnot klíče: v₁, v₂, ..., v₆ (odpovídají prvkům x₁, x₂, ..., x₆) a máme k dispozici funkci, která přiřazuje hodnotu prvku k indexu pole T.

Prvek s klíčem v je zapsán do pole T na index, který vypočítáme pomocí výrazu H(v). Hodnota NULL v tomto poli znamená, že prvek neexistuje.

Vzhledem k tomu, že velmi velký obor C převádíme na malou množinu prvků (pole T má omezenou velikost), nemůžeme se vyhnout kolizím přiřazení hodnoty H(v) na index pole T. Na obrázku to vidíme například u prvků x₃ a x₆ s klíči v₃ a v₆, pro které funkce H vypočítala stejný index pole. (Jak se pokusíme zdůvodnit v další části kapitoly, tento nedostatek naštěstí metodu nediskvalifikuje.)

U metody hešování naprostě odpadá jakékoli prohledávání množiny dat: známe-li funkci H, můžeme automaticky určit umístění libovolného prvku v paměti. Zároveň tedy okamžitě víme, kde jej případně hledat.

Čtenář se na tomto místě může oprávněně zeptat: Nač v takovém případě ztrácat čas seznamy, stromy či jinými datovými strukturami, můžeme-li jednoduše použít hešování? To je velmi dobrá otázka, ale bohužel se na ni nedá odpovědět jednou větou. Pro začátek můžeme uvést dva pádné důvody, kvůli nimž nelze metodu hešování pokaždé uplatnit:

- omezení paměti (je nutné předem rezervovat pole T pro E_{max} prvků),
- potíže s nalezením dobré funkce H.

První důvod je samozřejmý a nejspíš také méně závažný, ale nalézt vhodnou funkci není triviální. S vlastnostmi dobré funkce H se seznámíme v následující části kapitoly.

Hledání funkce H

Funkce H na základě klíče v poskytuje index v poli T, které slouží k ukládání dat. Jak si snadno domyslíme, existuje mnoho potenciálních funkcí, které z hodnoty daného klíče v vypočítají konkrétní adresu adr. Na komplikovanost funkce H mají hlavní vliv následující parametry: délka pole, do kterého chceme ukládat datové záznamy, a nepochybně také hodnota klíče v. Než se vrhneme ke klávesnici, abychom naprogramovali rychle vymyšlenou funkci H, měli bychom se dobře za-

myslet, které atributy datového záznamu budou tvořit klíč. Z logického hlediska by se tato funkce měla vyznačovat následujícími vlastnostmi:

- Měla by se snadno počítat, abychom místo náročného prohledávání datové množiny nemuseli systém zatěžovat zdlouhavými výpočty hodnoty $H(v)$.
- Různým hodnotám klíče v by měly odpovídat odlišné hodnoty indexu v poli T , aby nedocházelo ke kolizím přístupu (prvky by se měly v poli T rozmisťovat rovnoměrně).
- Funkce H by měla zajistit, že prvky budou v poli T rozloženy rovnoměrně a náhodně.

První bod není nutné komentovat. K následujícím bodům se ještě vrátíme, protože se jedná o základní požadavky na metodu hešování. V následující části kapitoly se seznámíme s typickými postupy při sestavování funkce H . V reálných aplikacích se využívají nejrůznější kombinace uvedených funkcí a na tomto místě v zásadě nelze uvést nějaká pevná pravidla, podle kterých bychom mohli postupovat. Často je potřeba experimentovat a provádět simulace, které umožní najít funkci H splňující výše uvedené podmínky pro konkrétní datovou množinu.

Nejznámější funkce H

Je nejvyšší čas, abychom představili několik typických matematických funkcí, které slouží k tvorbě funkcí využívaných při hešování. Tyto metody jsou poměrně jednoduché, ale samy o sobě nestačí. V praxi se většinou neuplatňují jednotlivě, ale v kombinaci. Čtenáře, kteří se s tematikou hešování setkávají poprvé, mohou navržené funkce (modulo, násobení atd.) poněkud překvapit. Nemáme zde totiž k dispozici žádnou vědeckou metodu – nic není pevně určeno a programátor si může v zásadě vybrat to, co se mu osobně líbí. Algoritmy vyhledávání či vkládání dat přitom budou fungovat v každém případě. V dalších příkladech budeme předpokládat, že klíče jsou znakové řetězce, které lze řadit za sebe a celkem libovolně interpretovat jako celá čísla. Každý znak abecedy budeme pro jednoduchost výpočtu kódovat pomocí pěti bitů (viz tabulku 7.1) – kód je přitom zvolen nahodile.

Tabulka 7.1: Příklad kódování písmen pomocí 5 bitů

A = 00001	B = 00010	C = 00011	D = 00100	E = 00101	F = 00110	G = 00111
H = 01000	I = 01001	J = 01010	K = 01011	L = 01100	M = 01101	N = 01111
O = 01110	P = 10000	Q = 10001	R = 10010	S = 10011	T = 10100	U = 10101
V = 10110	W = 10111	X = 11000	Y = 11001	Z = 11010		

Zmíněný nedostatek metody je naštěstí jen zdánlivý. Mnoho autorů příruček algoritmiky se dopouští chyby, když představuje hešování a soustředí se na vlastní postupy. Přitom podrobně nevysvětluje, PROČ vlastně chceme provádět aritmetické operace se zakódovanými klíči. Celá záležitost je přitom poměrně jednoduchá:

- Kódování slouží k tomu, abychom hodnotu klíče (nemusí být nutně číselná) nahradili číslem. Samotný kód přitom není důležitý. Záleží jen na tom, abychom jako výsledek dostali určité číslo, které lze dosadit do pozdějších výpočtů.
- Snažíme se o to, abychom záznamy umístili do pole velikosti M pokud možno co nejnáhodněji: funkce H má v závislosti na argumentu v poskytovat adresy od 0 do $M-1$. Celý problém spočívá v tom, že nemůžeme dosáhnout náhodného rozložení prvků, když disponujeme vstupními hodnotami, které z principu náhodné nejsou. Musíme tedy postupovat tak, abychom tuto „náhodnost“ nějakým způsobem vhodně simulovali.

KAPITOLA 7 Vyhledávací algoritmy

Praktické experimenty prováděné s velkými sadami vstupních dat ukázaly, že existuje skupina jednoduchých aritmetických funkcí (modulo, násobení, dělení), které se k tomuto účelu poměrně dobře hodí⁷. Budeme se jimi postupně zabývat v následujících odstavcích.

Součet modulo 2

Vzorec: $H(v_1v_2\dots v_n) = v_1 \oplus v_2 \oplus \dots \oplus v_n$

Příklad:

Pro

$$R_{max} = 37, H(\text{,,KOT"}) = (010110111010100)_2$$

dává

$$(01011)_2 \oplus (01110)_2 \oplus (10000)_2 = (10001) = (17)_{10}$$

Výhody:

- Funkce H se snadno počítá; součet modulo 2 oproti součinu a logickému součtu své argumenty nezvětšuje (jako logický součet) ani nezmenšuje (jako součin).
- Operátory $\&$ a $|$ způsobují, že se data shlukují na začátku, resp. na konci pole T, takže se jeho celková kapacita nevyužívá efektivně.

Nevýhody:

- Permutace stejných písmen poskytují v důsledku identický výsledek. To lze ovšem ošetřit systematickým cyklickým posunováním bitové reprezentace: první znak o jeden bit doprava, druhý znak o dva bity doprava atd.

Příklad:

- bez posunu:

$$H(\text{,,KTO"}) = (01011)_2 \oplus (10100)_2 \oplus (01110)_2 = (17)_{10}, \text{ zároveň } H(\text{,,TOK"}) = (17)_{10};$$

- s posunem: $H(\text{,,KTO"}) = (10101)_2 \oplus (00101)_2 \oplus (11001)_2 = (9)_{10}$, avšak $H(\text{,,TOK"}) = (01010)_2 \oplus (10011)_2 \oplus (01101)_2 = (10100)_2 = (20)_{10}$.

Součet modulo Rmax

Vzorec: $H(v) = v \% R_{max}$

Příklad:

Pro

$$R_{max} = 37: H(\text{,,KOT"}) = (01011\mathbf{0111010100})_2 \% (37)_{10} = (11732)_{10} = 3.$$

Výhody:

- funkce H se snadno počítá.

Nevýhody:

- Vypočítaná hodnota paradoxně závisí spíše na parametru R_{max} než na klíči!

⁷ Chcete-li si zopakovat pravidla aritmetických operací s logickými hodnotami, podívejte do přílohy B.

Například: Pokud je číslo R_{max} sudé, všechny získané indexy dat se sudými klíči budou s jistotou také sudé. Kromě toho jisté dělitele vedou k tomu, že mnoho datových hodnot dostane stejný index. Částečně se tomu můžeme vyhnout tím, že jako hodnotu R_{max} zvolíme prvočíslo. Opět se však budeme potýkat s akumulací prvků v jisté části pole. Na začátku jsme ovšem zmínili důležitý požadavek, že funkce H by měla indexy rozdělovat rovnoměrně po celém poli.

- V případě velkých binárních čísel, která přesahují rozsah vnitřní reprezentace počítače, již nemůžeme funkci modulo pomocí běžného aritmetického dělení.

Co se týče poslední vady, jednoduché řešení pro textové řetězce v jazyce C++ (vnitřně se přeče jedná o pouhé posloupnosti bajtů) je následující funkce, která je založena na interpretaci textu jako řady osmibitových čísel:

```
int H(char *s, int Rmax)
{
    for(int tmp=0; *s != '\0'; s++)
        tmp = (64*tmp+(*s)) % Rmax;
    return tmp;
}
```

Násobení

Vzorec: $H(v) = [((v \cdot \Theta) \% 1) \cdot E_{max}]$, kde $0 < \Theta < 1$

Výše uvedený vzorec má tento význam: Klíč v násobíme určitým číslem Θ z otevřeného intervalu $(0, 1)$. Z výsledku vezmeme desetinnou část, vynásobíme ji číslem E_{max} a dostaneme číslo, u kterého nás zajímá celočíselná část.

Existují dvě hodnoty parametru Θ , které rozptylují klíče v rámci tabulky celkem rovnoměrně:

$$\theta_1 = \frac{\sqrt{5} - 1}{2} = 0,6180339887 \quad a \quad \theta_2 = 1 - \theta_1 = 0,3819660113$$

Uvedenou informaci jsme dostali od matematiků, a protože se říká „*darovanému koni na zuby nehled*“, nebudeme se příliš šourat v tom, jak k tomuto poznatku přišli.

Příklad:

Pro

$$\Theta = 0,6180339887, E_{max} = 30 \text{ a klíč } v = \text{„KOT“} = 11\ 732 \text{ dostáváme}^8 H(\text{„KOT“}) = 23.$$

Řešení kolizí

Když provedeme několik jednoduchých pokusů s funkcemi, které jsme představili v předchozí části, jejich výsledky nás rychle zklamou. Zjistíme, že nesplňují očekávané vlastnosti, takže můžeme snadno začít pochybovat o tom, zda má celá koncepce vůbec smysl. Věci jsou ve skutečnosti trochu složitější. Na jednu stranu již zjišťujeme, že ideální funkce H neexistují⁹, ale na druhou stranu by bylo divné začít výklad o hešování a poté dospět k závěru, že se tato koncepce v praxi nedá realizovat. Situace samozřejmě není tak špatná. Existuje několik metod, které umožňují tyto problémy uspokojivě vyřešit, jak se přesvědčíme v dalších odstavcích.

⁸ Programově můžeme tuto hodnotu získat pomocí instrukce `int (fmod (11732 * 0.6180339887,1) * 30);` navíc je potřeba na začátek programu doplnit řádek `#include <math.h>`.

⁹ Dá se to dokonce zdůvodnit teoreticky (viz např. ve statistice dobře známý tzv. *narozeninový paradox*).

Návrat ke kořenům

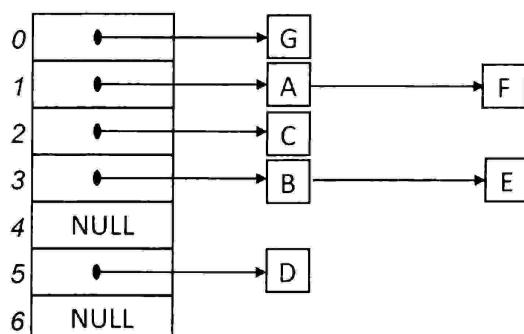
Co dělat v případě, kdy dojde ke kolizi dvou odlišných záznamů, kterým funkce H přidělila stejný index v poli T ? S takovými situacemi si můžeme poradit, když poněkud pozměníme samotný princip hešování. Jestliže se dohodneme, že do pole T budeme místo samotných záznamů ukládat *hlavy* seznamů prvků, které se vyznačují stejným klíčem, problém bude v zásadě vyřešen. Skutečně: pokud při vkládání prvku x do pole na index m zjistíme, že se tam již nachází jiný prvek, stačí připojit prvek x na konec seznamu, jehož hlava je uložena v položce $T[m]$ pole.

Analogicky funguje i vyhledávání: hledáme prvek x a funkce $H(x)$ vrací určitý index m . V případě, že položka $T[m]$ obsahuje hodnotu `NULL`, s jistotou víme že jsme hledaný prvek nenašli. V opačné situaci se musíme ještě přesvědčit, zda se prvek nevyskytuje v seznamu $T[m]$. (Na tomto místě si uvědomme, že seznamy budou zpravidla dosti krátké.)

Výše uvedený postup je znázorněn na obrázku 7.2.

Představuje situaci, která nastala poté, co jsme do pole T úspěšně vložili záznamy A, B, C, D, E, F a G, kterým funkce H přidělila následující adresy (indexy): 1, 3, 2, 5, 3, 1 a 0. Indexy pole, kde se neskrývají žádné datové záznamy, jsou inicializovány hodnotou `NULL` – viz např. buňky 4 a 6. Na pozici 1 dochází ke kolizi: záznamy A i F získaly stejnou adresu. Příslušná funkce `vloz` (kterou jsme předvídatě napsali) tuto situaci detekuje a vkládá prvek F na konec seznamu $T[1]$.

TAB



Obrázek 7.2: Řešení kolizí pomocí seznamů

Podobná situace nastává i u záznamů B a E. Proces vyhledávání prvků se podobá jejich vkládání. Měli bychom tedy snadno porozumět tomu, jak přesně probíhá prohledávání pole T , chceme-li dostat odpověď na otázku, zda pole obsahuje příslušný záznam, např. E.

Nemělo by nás na výše navržené metodě něco znepokojovalo? Počáteční představa o hešování byla velmi lákavá, protože naznačovala, že se obejdeme bez všech seznamů, stromů a jiných datových struktur s komplikovaným přístupem, a místo toho vystačíme s běžným přiřazením:

data → adresa buňky v paměti.

Při podrobnější analýze jsme narazili na jeden drobný problém a... vrátili jsme se ke starým dobrým seznamům. Právě z těchto důvodů můžeme usoudit, že uvedené řešení je poněkud umělé¹⁰ – docela dobře bychom mohli zůstat u seznamů a jiných dynamických datových struktur, aniž bychom zaváděli dodatečné prvky hešování! Můžeme v této situaci doufat, že se nám problémy s kolizemi přístupu podaří vyřešit? Pokud vás odpověď na tuto otázku zajímá, pusťte se do čtení následujících odstavců.

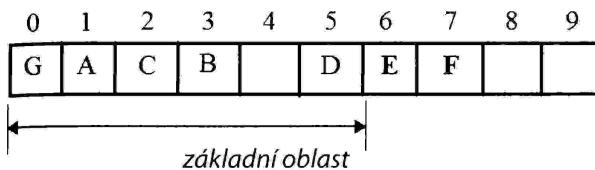
10 I když se metoda vyznačuje velmi příznivými časovými parametry.

Návrat k polím

Metoda hešování je z principu určena pro aplikace, které (díky možnosti odhadnout maximální počet uložených záznamů) umožňují rezervovat paměť pro statické pole, abychom k záznamům mohli snadno přistupovat pomocí indexů. Pokud dokážeme rezervovat pole pro všechny prvky, které chceme uložit, možná bychom mohli část pole vyhradit na řešení konfliktů přístupu.

Princip spočívá v tom, že pole T rozdělíme na dvě části: základní oblast a oblast přeplnění. Do druhé části by se prvky dostávaly v okamžiku, kdy by se pro ně nenašlo místo v základní části. Oblast přeplnění by se zaplňovala lineárně tak, jak by přicházely další „kolizní“ prvky. Abychom novou myšlenku ilustrovali, pokusme se využít data z obrázku 7.1. Budeme přitom předpokládat, že základní oblast a oblast přeplnění budou mít velikost odpovídající 6, resp. 4 záznamům.

Pole se bude zaplňovat způsobem, který je patrný na obrázku 7.3.



Obrázek 7.3: Dělení pole kvůli řešení kolizí

Záznamy E a F byly uloženy v okamžiku, kdy algoritmus zjistil přeplnění na následujících pozicích 6 a 7. Z toho můžeme usoudit, že někde „na pozadí“ musí existovat proměnná, která uchovává poslední volnou pozici oblasti přeplnění.

Musíme také přijmout nějakou konvenci ohledně označování volných pozic v základní oblasti. To už záleží na programátoru, který se při tom rozhoduje do značné míry podle struktury záznamů, které je potřeba ukládat.

Řešení, které zohledňuje rozdelení pole, není příliš složité, což nepochybňě patří k jeho výhodám. Funkce `vloz` a `hledej` lze napsat za několik minut, jak si můžete sami vyzkoušet.

Pro pořádek bychom však měli upozornit na jedno slabé místo. Ještě jsme neuvažovali o tom, co se stane, když se zaplní... oblast přeplnění! (Vypsáním „milé“ zprávy o chybě problém nevyřešíme.) Před nasazením této metody bychom měli důkladně analyzovat rozměry polí, aby nedošlo k selhání aplikace v nejméně vhodnou chvíli – například před zápisem dat na disk.

Lineární pokusy

Ve výše popsané metodě jsme poněkud umělým způsobem vyřešili problém s kolizemi při ukládání do pole T . Toto pole jsme rozdělili na dvě části, které sloužily k ukládání záznamů v různých situacích. Ovšem zatímco celkový rozměr pole R_{max} můžeme ve většině případů odhadnout snadno, v praxi bývá dosti těžké určit vhodnou velikost oblasti přeplnění. Důležitou roli zde mají jak samotná data, tak i funkce H . Pokud bychom chtěli přibližně odhadnout správnou délku obou částí pole, v zásadě bychom museli data i funkci analyzovat současně. Problém pochopitelně automaticky odpadá v situacích, kdy máme k dispozici značně velkou kapacitu volné paměti. Bylo by ovšem velmi riskantní takový stav automaticky předpokládat.

Jak jsme již dříve zjistili, kolizím se při hešování nemůžeme vyhnout. Důvod je jednoduchý: neexistuje ideální funkce H , která by rovnoměrně rozmístila všech R_{max} prvků po celém poli T . Místo toho, abychom s touto realitou bojovali (na tom byly založeny předchozí metody), měli bychom se jí možná raději přizpůsobit.

KAPITOLA 7 Vyhledávací algoritmy

Vycházíme z tohoto principu: Pokud v okamžiku zápisu nového záznamu¹¹ do pole zjistíme, že došlo ke kolizi, můžeme se pokusit o zapsání prvku na první volné místo. Funkce `vloz` bude v tomto případě používat následující algoritmus (předpokládáme pokus o zápis záznamu x charakterizovaného klíčem v do pole T):

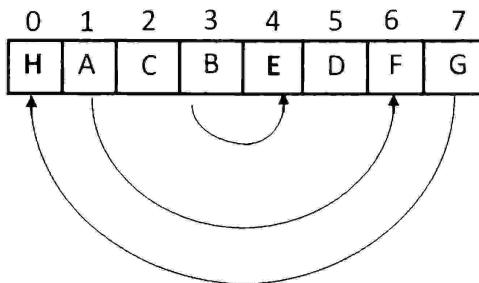
```
int pos=H(x.v);
while (T[pos] != VOLNE)
    pos = (pos+1) % Rmax;
T[pos]=x;
```

Řekněme nyní, že hledáme prvek, který se vyznačuje klíčem k . Funkce `hledej` by potom mohla vypadat takto:

```
int pos=H(k);
while ((T[pos] != VOLNE) && (T[pos].v != k))
    pos = (pos+1) % Rmax;
return T[pos]; // vrácení nalezeného prvku
```

V hešování není skutečně není velký rozdíl mezi vyhledáváním a vkládáním. Algoritmy jsou zájemně zapsané v pseudokódu, protože rozumný příklad využití této metody by musel obsahovat podrobné deklarace datového typu, pole, funkce H a speciální hodnoty `VOLNE`. To vše by vyžadovalo zdlouhavou analýzu. Instrukce `pos = (pos+1) % Rmax;` zajišťuje návrat na začátek pole v okamžiku, kdy se během (následných) iterací cyklu `while` dostaneme na jeho konec.

Na ukázku se podívejme, jak se nová metoda osvědčí, když se do pole T pokusíme postupně vložit záznamy A, B, C, D, E, F, G a H, kterým funkce H přidělila následující adresy (indexy): 1, 3, 2, 5, 3, 1, 7 a 2. Dále se dohodněme, že pole T bude mít délku 8 záznamů – samozřejmě v rámci této ukázky, protože v praxi by taková hodnota neměla přílišný smysl. Výsledky si můžeme prohlédnout na obrázku 7.4:



Obrázek 7.4: Řešení kolizí pomocí lineárních pokusů

Poměrně zajímavě vypadají teoretické výpočty *průměrného počtu pokusů*, které potřebujeme k nalezení záznamu x . V případě úspěšného hledání lze průměrný počet pokusů odhadnout pomocí vztahu:

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

kde α je koeficient zaplnění pole T . Analogický výsledek pro neúspěšné hledání je přibližně určen vztahem:

11 Může to být samozřejmě libovolná prostá proměnná.

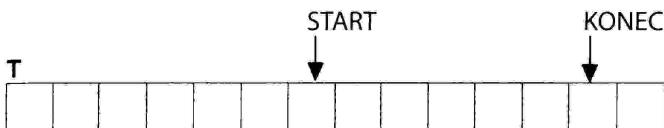
$$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

Například: Pro pole zaplněná z dvou třetin své kapacity ($\alpha = 2/3$) mají tato čísla hodnotu: 2 a 5. V praxi bychom si měli dávat pozor, aby nedocházelo k přílišnému zaplnění pole T , protože výše uvedená čísla se hodně zvětšují (koeficient α by neměl nabývat hodnot blízkých 1). Při odvození uvedených vzorců se vycházelo z předpokladu, že funkce H rovnoměrně rozmištuje prvky ve velkém poli T . Toto upozornění je poměrně důležité, protože výše uvedené výsledky mají jen statistickou povahu.

Dvojité hešování

Lineární pokusy vedou k nevhodnému lineárnímu zaplňování pole T , což odporuje původnímu požadavku na funkci H (viz stranu 182). Intuitivní řešení tohoto problému vypadá docela prostě: je potřeba udělat něco, abychom prvky rozmištili poněkud náhodněji. Metoda lineárních pokusů nebyla z tohoto hlediska rozumná, protože po nalezení určité zaplněné oblasti pole T umisťovala nový prvek těsně za ni, takže ji ještě zvětšovala. Někdo by se mohl zeptat: A proč by to vlastně mělo vadit? Estetická hlediska zde samozřejmě nehrají žádnou roli. Uvědomme si, že lineárně zaplněná oblast brání v tom, abychom rychle našli volné místo na vložení nového prvku. V této situaci se rovněž komplikuje efektivní vyhledávání dat.

Rozeberme jednoduchý příklad, který je znázorněn na obrázku 7.5.



Obrázek 7.5: Ztížené vyhledávání dat při lineárních pokusech

Šedě označené buňky pole označují místa, která jsou již obsazena. Funkce $H(k)$ poskytla určitý index, od kterého začíná prohledávání oblasti pole (hledáme určitý prvek, který se vyznačuje klíčem k). Řekněme, že prohledávání začíná od indexu, který je symbolicky označen jako START. Proces vyhledávání je úspěšný v případě, že je nalezen hledaný záznam. Abychom to ověřili, musíme provést poměrně náročné¹² porovnání $T[pos] \cdot v \neq k$ (viz algoritmus procedury hledej popsaný v předchozí části).

Toto porovnání navíc musí proběhnout při každém přesunu v rámci lineárně zaplněné oblasti! Informaci o případném neúspěchu vyhledávání dostaneme teprve poté, kdy projdeme celou oblast a narazíme na první volné místo. V příkladu znázorněném na obrázku se algoritmus teprve po sedmi porovnáních dostane k prázdné buňce (označené popiskem KONEC) a zjistí, že se namáhal nadarmo. Pokud by pole bylo zaplněno méně lineárně, ze statistického hlediska bychom mnohem rychleji došli k VOLNÉMU místu, kde by byl proces vyhledávání automaticky ukončen s neúspěšným výsledkem.

Naštěstí existuje jednoduchý způsob, jak se lineárnímu shlukování prvků vyhnout – jedná se o tzv. *dvojité hešování*. Když dojde ke konfliktu, následuje pokus o rozmištění prvků pomocí druhé pomocné funkce H .

Procedura vlož zůstává téměř beze změny:

¹² Náklady operace porovnání závisí na stupni složitosti klíče, tzn. na počtu a typu položek záznamu, ze kterých se skládá.

```

int pos = H1(x,v);
int krok=H2(x,v);
while (T[pos] != VOLNE)
    pos = (pos+krok) % Rmax;
T[pos]=x;

```

Procedura vyhledávání je velmi podobná a podle předchozího příkladu ji každý čtenář jistě dokáže vytvořit samostatně.

Rozeberme nyní problém výběru funkce $H2$. Snadno si domyslíme, že značně ovlivňuje efektivitu procesu vkládání (a přirozeně i vyhledávání). Funkce $H2$ se především musí lišit od funkce $H1$. V opačném případě bychom jen zkomplikovali tvoření „souvislých“ oblastí, čemuž se právě chceme vyhnout. Následující požadavek je samozřejmý: musí to být jednoduchá funkce, která proces vyhledávání a vkládání nebude zdržovat. Příkladem takové prosté a zároveň v praxi efektivní funkce může být $H2(k) = 8 - (k \% 8)$ – má dosti široký záběr a vyznačuje se nepopiratelnou jednoduchostí.

Metoda *dvojitého hešování* je zajímavá díky tomu, že viditelně urychluje vyhledávání dat. Podívejme se na výsledky teoretických výpočtů průměrného počtu pokusů při vyhledávání s úspěšným i neúspěšným výsledkem. V případě úspěšného hledání lze průměrný počet pokusů odhadnout pomocí vztahu:

$$\frac{1}{\alpha} \log \left(\frac{1}{1-\alpha} \right)$$

(kde α je tak jako výše koeficient zaplnění pole T).

Analogický výsledek pro neúspěšné hledání je přibližně určen vztahem:

$$\frac{1}{1-\alpha}$$

Využití hešování

Dosud jsme se omezovali pouze na elementární příklady: krátká pole, jednoduché znakové nebo číselné klíče atp. Reálné aplikace mohou být samozřejmě mnohem komplikovanější a teprve při jejich tvorbě dokážeme skutečně ocenit, jak jsou teoretické informace užitečné. Hešování lze uplatnit dosti neobvyklými způsoby: data se vůbec nemusí nacházet v operační paměti. U databázového programu lze poměrně snadno pomocí kódu H efektivně vyhledávat data. Když konstruujeme velký kompilátor nebo konsolidátor, můžeme pomocí metod hešování vyhledávat komplikované moduly ve velkých knihovních souborech.

Shrnutí metod hešování

Při hešování můžeme provádět nejrůznější srovnávací analýzy – získané výsledky jsou věrohodné a intuitivně odpovídají skutečnosti. Odvozování výsledků je však poměrně složité a na tomto místě se jím nebudeme zabývat. Stojí ovšem za zmínku uvést některé obecné praktické závěry:

- Při malém zaplnění¹³ pole T jsou všechny metody přibližně stejně efektivní.
- Metoda *lineárních pokusů* se dokonale osvědčuje u velkých a málo využitých polí T (čili tehdy, kdy máme k dispozici hodně volného místa v paměti). Výhodou jejího nasazení je také nepochybná jednoduchost.

¹³ Tzn. do přibližně 30–40 % celkové kapacity pole.

Nakonec bychom měli zdůraznit to, nač jsme ve víru prezentace různých metod a jejich vlastnosti mohli snadno zapomenout: hešování může být přímo ideálním nástrojem... ale jen v případě práce s daty, u kterých dokážeme s vysokou pravděpodobností předpovídat jejich rozsah. Nemůžeme si totiž dovolit, aby došlo k selhání aplikace kvůli tomu, že jsme neopatrně odhadli potřebnou délku pole!

Například: víme-li, že budeme zpracovávat množinu, která stabilně čítá třeba 700 záznamů, deklarujeme pole T velikosti 1000. Tím zaručíme rychlé vyhledávání i vkládání dat dokonce i při uložení všech 700 záznamů. Ukazuje se, že magickou hranicí je zaplnění pole na 70–80 %. Když tento limit překročíme, hešování přináší stále menší viditelný zisk – proto je vhodné, abychom se k uvedené hodnotě raději příliš nepřibližovali. Metoda je přesto zajímavá a stojí za uplatnění – samozřejmě tehdy, kdy zohledníme praktický kontext koncové aplikace.



Poznámka: Několik ukázkových funkcí H se nachází v souboru `hash.cpp` – doporučuji, abyste je vyzkoušeli a simulovali jejich fungování s různými datovými množinami.

KAPITOLA 8

Prohledávání textů

V této kapitole:

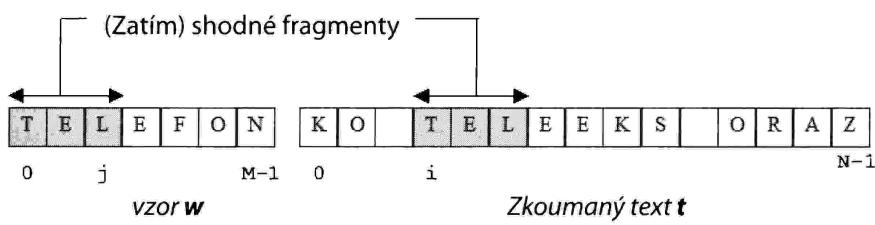
- Algoritmus vyhledávání hrubou silou
- Nové vyhledávací algoritmy

Než otevřeme téma nové kapitoly, měli bychom vysvětlit určité nedorozumění, ke kterému by mohlo na základě názvu této kapitoly dojít. Za *text* budeme považovat znakový řetězec v informatickém smyslu. V některých případech nebudou mít takové řetězce nic společného s lidskou řečí. Textem může být například i bitová posloupnost¹, kterou lze víceméně smluvně rozdělit na stejně velké části, kterým přiřazujeme určitý číselný kód².

Opakovaně se přesvědčujeme o tom, že mnoho operací s informacemi si můžeme usnadnit, když přijmeme jisté konvence, jak tyto informace interpretovat. Zůstaňme tedy u obecného označení „text“, ačkoli víme, že za tímto výrazem se může skrývat mnoho různých významů.

Algoritmus vyhledávání hrubou silou

Pokusme se nyní společně vyřešit úkol vyhledávání vzoru³ w o délce M znaků v textu t délky N . Snadno dokážeme navrhnut dosti triviální algoritmus, který bude tuto úlohu řešit na základě úvah, které jsou symbolicky znázorněny na obrázku 8.1.



Obrázek 8.1: Algoritmus vyhledávání v textu hrubou silou

Vyhraďme indexy j a i , které umožní pohyb ve vzoru i textu při operaci porovnávání shody vzoru s textem znak po znaku. Předpokládejme, že jsme během prohledávání nalezli shodu v oblastech, které jsou na obrázku znázorněny šedou barvou. V tom případě se přesuneme zároveň ve vzoru i v textu o jednu pozici dopředu ($i++$; $j++$).

1 Která reprezentuje např. obsah obrazovky.

2 Např. ASCII nebo libovolný jiný.

3 Ang. *pattern matching*.

KAPITOLA 8 Prohledávání textů

Co by se ale mělo stát s indexy i a j , když algoritmus zjistí, že se znaky neshodují? V takové situaci celé vyhledávání končí neúspěšně, takže musíme zrušit „šedou zónu“ shodnosti. Přitom přecházíme v textu zpět o tolik pozic, kolik se shodovalo, tedy o $j-1$ znaků. Při té přiležitosti vynulujeme hodnotu j . Popišme ještě okamžik, kdy algoritmus zjistí, že se vzor s textem zcela shoduje. Kdy k tomu dojde? Není těžké si všimnout, že když algoritmus detekuje shodu posledního znaku, měl by se index j vyrovnat hodnotě M . Snadno tedy dokážeme určit pozici, na které vzor ve zkoumaném textu začíná: bude to samozřejmě $i-M$.

Když uvedený popis přeložíme do jazyka C++, jednoduše dojdeme k následující proceduře:

```
txt-1.cpp
int hledej(char *w,char *t)
{
    int i=0,j=0, M=strlen(w), N=strlen(t);
    while( (j<M) && (i<N) )
    {
        // *
        if(t[i]!=w[j])
        {
            i-=j-1;
            j=-1;
        }
        i++;
        j++;           // **
    }
    if(j==M)
        return i-M;
    else
        return -1;
}
```

Funkci `hledej` lze používat způsobem, který je patrný v následující ukázce funkce `main`:

```
int main()
{
    char *b="abrakadabra",*a="rak";
    cout << hledej(a,b) << endl;
}
```

Funkce jako svůj výsledek vrací pozici v textu, kde začíná vzor, nebo hodnotu -1 v případě, kdy hledaný text nebyl nalezen – tuto konvenci již dokonale známe. Podívejme se pozorněji na příklad vyhledávání vzoru 10100 v určitém binárním řetězci (viz obrázek 8.2).

Obrázek je poněkud zjednodušený: vodorovný posun vzoru v praxi odpovídá instrukcím, které jsou ve výpisu označeny komentářem s jednou hvězdičkou (*), zatímco celá šedá zóna délky k symbolizuje k -násobné provedení instrukcí označených dvěma hvězdičkami (**).

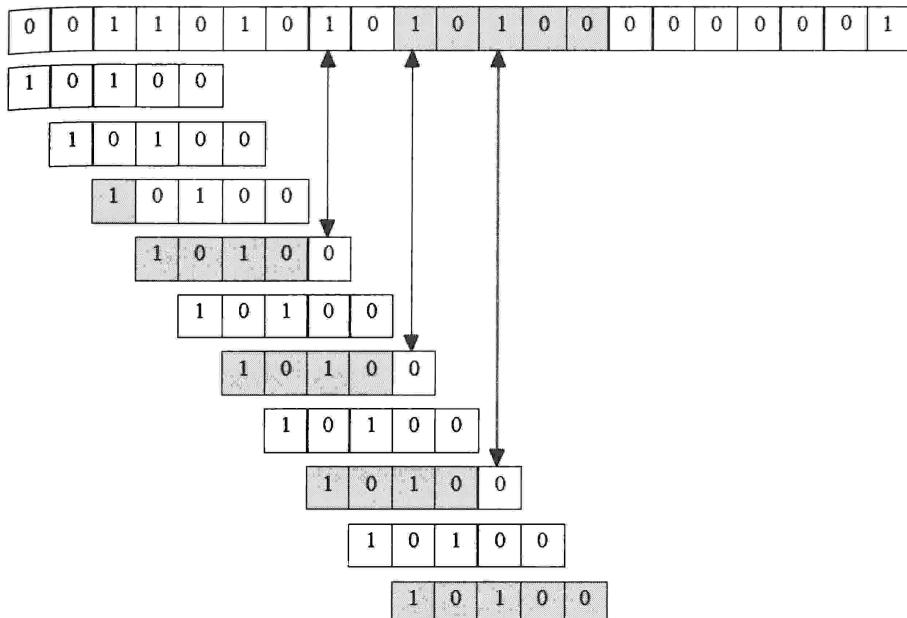
Na základě uvedeného příkladu se můžeme pokusit vymyslet nejhorší text i vzor, pro které bude proces vyhledávání trvat maximálně dlouho. Jedná se samozřejmě o text i vzor, které sestávají ze samých nul a končí jedničkou⁴.

Pokusme se určit třídu algoritmu pro extrémní nejhorší případ, který jsme před chvílí uvedli. Výpočet není příliš složitý:

4 Nuly a jedničky zde zastupují dva odlišné znaky.

Když předpokládáme, že algoritmus je nutné restartovat $(N-1) - (M-2) = N-M+1$ krát, a víme, že během každého cyklu je nutné provést M porovnání, okamžitě dostáváme $M(N-M+1)$, tedy přibližně⁵ $M \cdot N$.

Algoritmus, který předvádíme v této části, využívá počítač jako výkonnéjší mechanické počítadlo⁶. Kvůli své výpočetní složitosti se v praxi nedá použít při prohledávání binárních řetězců, v nichž se může objevit mnoho nevhodných kombinací hodnot. Jedinou výhodou algoritmu je jeho jednoduchost, která však nestačí k tomu, abychom se smířili s tím, jak pomalu funguje.



Obrázek 8.2: „Falešné starty“ při vyhledávání

Nové vyhledávací algoritmy

Algoritmus, který budeme analyzovat v této části, má zajímavou historii. Stojí za to, abychom ji zde stručně shrnuli. V roce 1970 S. A. Cook zdůvodnil teoretický výsledek, který se týkal jistého abstraktního stroje. Vyplývalo z něj, že existuje algoritmus vyhledávání vzoru v textu, který v nejhorším případě funguje v čase úměrném $M+N$. Vůbec se nepředpokládalo, že by se výsledek Cookovy práce dal prakticky využít, nicméně D. E. Knuth a V. R. Pratt na jeho základě získali algoritmus, který už bylo možné implementovat v počítači. Při té příležitosti se ukázalo, že mezi teoretickými úvahami a praktickými aplikacemi není tak velká mezera, jak by se mohlo zdát. Ve stejné době J. H. Morris objevil úplně stejný algoritmus, když řešil problém, s nímž se setkal při praktické implementaci textového editoru. Algoritmus $K-M-P$ – tak jej budeme dále označovat – patří k mnoha příkladům souběžných objevů, které jsou ve vědě poměrně časté: z nějakých neznámých důvodů několik osob, které pracují nezávisle, najednou dojde ke stejnemu cennému výsledku. Jistě se shodneme, že se jedná o docela zvláštní jev, který si přímo říká o nějaké metafyzické vysvětlení. Knuth, Morris a Pratt publikovali svůj algoritmus teprve roku 1976. Mezitím se objevil další „zázračný“ algoritmus, jehož autory byli tentokrát R. S. Boyer a J. S. Moore. Ukázalo se, že jejich al-

5 Hodnota M bude obvykle mnohem menší než N .

6 Jeden z mých známých termín *brute-force* původně přeložil jako „metodu mastodonta“.

KAPITOLA 8 Prohledávání textů

goritmus je v určitých případech mnohem rychlejší než algoritmus *K-M-P*. Současně jej vynalezl (či objevil) i R. W. Gosper. Oba tyto algoritmy jsou však bez hlubší analýzy dosti těžko srozumitelné, což ztížilo jejich rozšíření.

V roce 1980 R. M. Karp a M. O. Rabin usoudili, že prohledávání textů se z principu neliší od standardních vyhledávacích metod, a přišli s algoritmem, který sice také funguje v čase úměrném $M+N$, ale myšlenkově se blíží jednoduchému algoritmu *vyhledávání hrubou silou*, s nímž jsme se již seznámili. Kromě toho se jedná o algoritmus, který lze poměrně snadno zobecnit na případ vyhledávání v dvourozměrných polích, takže se potenciálně hodí ke zpracování obrazů.

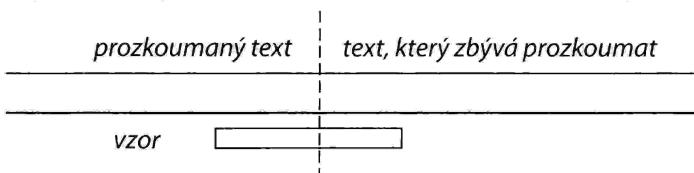
V následujících třech částech kapitoly postupně rozebereme všechny algoritmy, které jsme v předchozím historickém přehledu zmínili.

Algoritmus K-M-P

Nedostatkem algoritmu *vyhledávání hrubou silou* je jeho citlivost na konfiguraci dat: falešné restarty jsou zde velmi nákladné. Při analýze textu se vracíme o celou délku vzoru a přitom zapomínáme vše, co jsme do té doby testovali. Nabízí se myšlenka, že by bylo vhodné nějak využít informace, které jsme již získali – v následujících fázích se přece zčásti opakují stejná porovnání.

V jistých konkrétních případech můžeme algoritmus zlepšit, známe-li strukturu analyzovaného textu. Například: Jestliže s jistotou víme, že první znak vyhledávaného vzoru se v něm již znova nikdy neobjeví⁷, nemusíme v případě restartu vracet ukazatel i o $j-1$ pozic jako v původní verzi (viz výpis `txt-1.cpp`). V tomto případě stačí prostě inkrementovat hodnotu i , protože víme, že případné opakování vyhledávání by již nemohlo nic najít. Jistě se však shodneme, že s texty o takto charakteristických vlastnostech se můžeme setkat jen poměrně zřídka. Předchozí příklad však dokládá, že s vyhledávacími algoritmy můžeme vždy manipulovat – stačí si jen všímat, jaké možnosti se nabízejí. Princip algoritmu *K-M-P* spočívá ve prvotním prozkoumání vzoru, kdy algoritmus zjišťuje, o kolik pozic je potřeba vrátit ukazatel i , pokud zjistí neshodu zkoumaného textu se vzorem. Samozřejmě lze zvolit i přístup s posunováním vzoru vpřed – výsledek přitom bude stejný. Dále budeme vycházet právě z druhé uvedené možnosti. Již víme, že se po zkoumaném textu musíme pohybovat poněkud intelligentněji než v případě algoritmu *vyhledávání hrubou silou*. Jestliže zjistíme neshodu na určité pozici j vzoru⁸, musíme upravit tento index a využít informace, které jsme již získali v prozkoumané „šedé zóně“ shodnosti.

Všechno to jistě zní značně nesrozumitelně. Je tedy načase celou záležitost co nejrychleji objasnit, abychom se vyhnuli možným nedorozuměním. Podívejme se proto na obrázek 8.3.



Obrázek 8.3: Vyhledání optimálního posunutí v algoritmu K-M-P

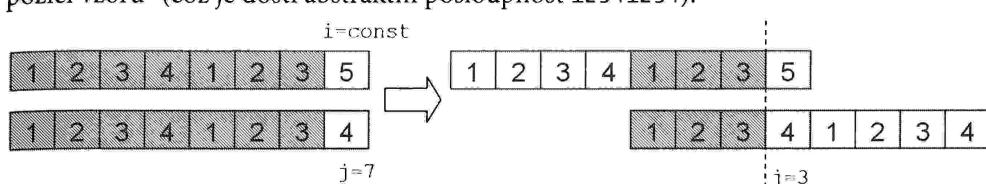
Okamžik neshody je označen přerušovanou svislou čarou. Představme si tedy, že nyní vzor velmi pomalu posunujeme doprava a zároveň se díváme na již prozkoumaný text, abychom sledovali případný překryv v té části vzoru, která se nachází po levé straně přerušované čáry, s textem, který je

⁷ Příklad: „BBBBBBB“ – znak „A“ se vyskytl pouze jednou.

⁸ Nebo i v případě zkoumaného textu.

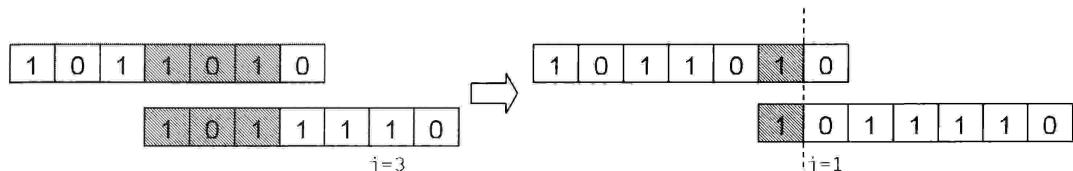
umístěn nad vzorem. V jisté chvíli se může ukázat, že se obě tyto části shodují. Zastavíme tedy posunování a pokračujeme v testování (znak po znaku), zda se rovnají obě části za přerušovanou čarou. Na čem záleží, zda se zkoumané fragmenty textu i vzoru případně budou shodovat? Paradoxně přitom nezáleží na zkoumaném textu – můžeme-li to tak říci. Informace o tom, jaký byl, se skrývá v tvrzení „shodovalo se $j-1$ znaků“. V tom smyslu můžeme na zkoumaný text úplně zapomenout a hledané optimální posunutí najít pouze analýzou vzoru. Právě na tomto postřehu spočívá idea algoritmu *K-M-P*. Ukazuje se, že díky prozkoumání struktury vzoru můžeme spočítat, jak máme upravit index j v případě, kdy zjistíme neshodu textu se vzorem na j -té pozici.

Než se ponoříme do vysvětlování toho, jak se tyto posuny počítají, podívejme se na několik příkladů toho, jak se projevují. Na obrázku 8.4 si můžete všimnout, že jsme zjistili neshodu na sedmé pozici vzoru⁹ (což je dosti abstraktní posloupnost 12341234).



Obrázek 8.4: Posunování vzoru v algoritmu K-M-P (1)

Index i ponecháme beze změny a v prohledávání můžeme bez problému pokračovat, přičemž budeme měnit pouze index j . Jaké posunutí vzoru je optimální? Když budeme vzor pomalu přemisťovat doprava (viz obrázek 8.4), v jistém okamžiku se překryjí posloupnosti 123 před přerušovanou čarou. Celá neshodující se oblast se dostala na pravou stranu a můžeme případně pokračovat v dalším testování. Analogický příklad najdeme i na obrázku 8.5.



Obrázek 8.5: Posunování vzoru v algoritmu K-M-P (2)

V tomto případě se neshoda objevila na pozici $j=3$. Když budeme podobně jako v předchozím příkladu posunovat vzor doprava, zjistíme, že jediný možný překryv znaků se objeví po posunutí o dvě pozice doprava – neboli pro $j=1$. Dále se ukazuje, že se překryly také znaky za přerušovanou čarou, to však algoritmus zjistí teprve při dalším testování shody na pozici i .

Algoritmus *K-M-P* při své činnosti potřebuje pole posunutí `int shift[M]`. Toto pole se používá následujícím způsobem: Pokud se znaky na pozici j neshodují, bude se následující hodnota j rovnat `shift[j]`. Prozatím se nebudeme zabývat tím, jak se toto pole inicializuje (samozřejmě pro každý vzor jinak). Ihned však můžeme předvést algoritmus *K-M-P*, jehož konstrukce téměř přesně kopíruje algoritmus *vyhledávání hrubou silou*:

```
kmp.cpp
int kmp(char *w, char *t)
{
    int i, j, N=strlen(t);
```

⁹ Při tradičním počítání indexu pole od nuly.

KAPITOLA 8 Prohledávání textů

```

for(i=0, j=0; (i<N) && (j<M); i++, j++)
    while((j>=0) && (t[i]!=w[j]))
        j=shift[j];
    if (j==M)
        return i-M;
    else
        return -1;
}

```

Speciální případ nastává, když se neshodují znaky na nulové pozici: z principu zde nemůžeme vzor posunout, abychom dosáhli toho, že se znaky budou překrývat. Proto chceme, aby se index *j* při současném zvyšování indexu *i* neměnil. Dosáhneme toho tak, že položku *shift*[0] inicializujeme smluvní hodnotou -1. Při další iteraci cyklu **for** pak dojde k inkrementaci hodnot *i* i *j*, která hodnotu *j* vynuluje.

Zbývá ještě vysvětlit, jak se vytváří pole *shift*[*M*]. Hodnoty v tomto poli je nutné spočítat ještě před voláním funkce **kmp**. Z toho je zřejmé, že při vícenásobném vyhledávání stejného vzoru již toto pole nemusíme inicializovat znova. Funkce inicializující pole je zajímavá tím, že se prakticky shoduje s funkcí **kmp**. Jediný rozdíl spočívá v tom, že algoritmus kontroluje shodu vzoru... s ním samotným!

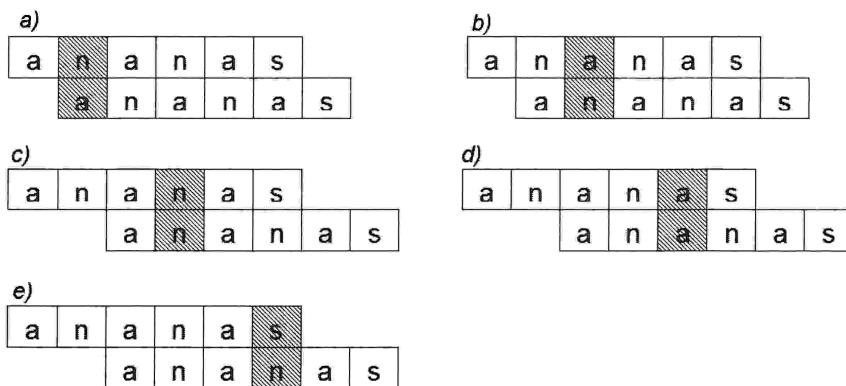
```

int shift[M];
void init_shifts(char *w)
{
    int i, j;
    shift[0]=-1;
    for(i=0, j=-1; i<M-1; i++, j++, shift[i]=j)
        while((j>=0)&&(w[i]!=w[j]))
            j=shift[j];
}

```

Tento algoritmus má následující smysl: Hned po inkrementaci hodnot *i* a *j* víme, že prvních *j* znaků vzoru se shoduje se znaky na pozicích: *p*[*i*-*j*-1] ... *p*[*i*-1] (posledních *j* pozicích v prvních *i* znacích vzoru). Vzhledem k tomu, že se jedná o největší *j* splňující uvedenou podmínu, musíme nastavit *shift*[*i*] na hodnotu *j*, abychom nevynechali potenciální výskyt vzoru v textu.

Podívejme se, jak funkce **init_shifts** zpracuje slovo *ananas* (viz obrázek 8.6). Stínovaná písmena označují místa, kde se vzor neshoduje s textem. V každém případě je vliv posunování vzoru znázorněn graficky – je jasné patrné, které oblasti se před stínovanou zónou překrývají (srovnej s obrázkem 8.5).



Obrázek 8.6: Optimální posunování vzoru „ananas“ v algoritmu K-M-P

Ještě připomeňme, že pole `shift` obsahuje novou hodnotu indexu j , který se přemisťuje po vzoru. Známe-li předem vzory, které budeme vyhledávat, můžeme algoritmus $K\text{-}M\text{-}P$ optimalizovat. Například: Pokud ve svých textech často hledáme slovo *ananas*, můžeme do funkce `kmp` vložit pole posunutí:

```
ananas.cpp
int kmp_ananas(char *t)
{
    int i=-1;
    start: i++;
    et0:
    if (t[i]!='a')
        goto start;
    i++;
    et1:
    if (t[i]!='n')
        goto et0;
    i++;
    et2:
    if (t[i]!='a')
        goto et0;
    i++;
    et3:
    if (t[i]!='n')
        goto et1; i++;
    if (t[i]!='a')
        goto et2; i++;
    if (t[i]!='s')
        goto et3; i++;
    return i-6;
}
```

Abychom mohli správně nastavit návěstí, musíme samozřejmě alespoň jednou spustit funkci `init_shifts` nebo spočítat příslušné hodnoty ručně. V každém případě to stojí za námahu: výše uvedená funkce se vyznačuje značnou úsporností výsledného kódu jazyka assembler, takže funguje velmi rychle. Máte-li k dispozici kompilátor, který umožňuje generovat výsledný kód jako tzv. „assembly output“¹⁰, můžete si snadno ověřit, nakolik se verze `kmp` a `kmp_ananas` liší. Pro příklad mohu uvést, že u zmiňovaného kompilátoru GNU měla klasická verze procedury `kmp` (spolu s funkcí `init_shifts`) velikost kolem 170 řádků assemblerového kódu, zatímco verze `kmp_ananas` si vystačila přibližně se 100 řádky. (Viz: soubory ke stažení s příponou `.s` pro kompilátor GNU nebo `.asm` pro kompilátor Borland C++ 5.5.)

Algoritmus $K\text{-}M\text{-}P$ funguje v čase, který je v nejhorším případě úměrný hodnotě $M+N$. Při jeho použití můžeme zaznamenat nejvyšší zisk u textů s vysokým stupněm vnitřního opakování, jaké se v praxi vyskytují dosti zřídka. U typických textů nám tedy výběr metody $K\text{-}M\text{-}P$ nepřinese patrnější výhody.

10 V případě kompilátorů oblíbené řady Borland C++ je nutné program zkompilovat ručně příkazem `bcc32 -S -Ixxx soubor.cpp`, kde *xxx* označuje složku se soubory typu *H*. Stejnou možnost poskytuje i kompilátor GNU C++, kde je potřeba zadat: `c++ -S soubor.cpp`.

Bez tohoto algoritmu se však neobejdeme v těch aplikacích, kde text prohlížíme lineárně bez ukládání do vyrovnávací paměti. Je zřejmé, že hodnota ukazatele *i* ve funkci *kmp* se nikdy nesnižuje. To znamená, že soubor můžeme procházet od začátku do konce, aniž bychom se vraceli zpět. V některých systémech to může mít značný praktický význam. Uvedme příklad: Máme v úmyslu analyzovat velmi dlouhý textový soubor a charakter prováděných operací neumožňuje, abychom se přitom vraceli zpět (soubor načítáme průběžně).

Boyer-Mooreův algoritmus

Další algoritmus, který budeme analyzovat, je koncepčně značně srozumitelnější než algoritmus *K-M-P*. Oproti metodě *K-M-P* se porovnává poslední znak vzoru. Tento nekonvenční přístup přináší několik podstatných výhod:

- Pokud se při porovnávání ukáže, že aktuálně testovaný znak se ve vzoru vůbec nevyskytuje, můžeme v analýze textu „přeskočit“ vpřed o celou délku vzoru! Zátež algoritmu se tedy z analýzy případných shod přenáší na testování neshod, které se statisticky vyskytují mnohem častěji.
- Délka skoků vzoru je obvykle mnohem větší než 1 – srovnejme s metodou *K-M-P*.

Než přejdeme k podrobnému rozboru kódu, popišme si jeho činnost na příkladu. Podívejme se na obrázek 8.7, který představuje vyhledávání znakového řetězce „*lek*“ v textu „*Z pamiętnika mło- dej lekarki*“¹¹.

V prvních pěti porovnáních se testují písmena: *p*, *i*, *n*, *a* a *ł*, která se ve vzoru nevyskytují (polští písmeno „*ł*“ není totožné s písmenem „*l*“). Pokaždé tedy můžeme v textu přeskočit o tři znaky dopředu (o délku vzoru). Při šestém porovnání se však setkáváme s písmenem *e*, které ve vzoru „*lek*“ existuje. Algoritmus tedy přesune vzor o tolik pozic dopředu, aby byla písmena *e* v zákrytu, a pokračuje v porovnávání.

Poté se ukazuje, že písmeno *j* do vzoru nepatří – smíme se tedy posunout o další tři znaky dopředu. Vzápětí již nalézáme hledané slovo. Stačí k tomu pouze posunutí vzoru o jednu pozici, aby se proti sobě ocitla písmena *k*.

Jak vidíme, algoritmus je srozumitelný, jednoduchý a rychlý. Ani jeho realizace není příliš složitá. Podobně jako u předchozí metody i v tomto případě musíme provést jistou předběžnou komplikaci, abychom vytvořili pole posunutí. V tomto případě však pole bude mít tolik pozic, kolik je znaků v abecedě – přesněji řečeno se jedná o všechny znaky, které se mohou v textu vyskytnout, spolu s mezerou.

Budeme rovněž potřebovat jednoduchou funkci *index*, která bude při výskytu mezery vracet nulu a v ostatních případech pořadí písmena v abecedě. Následující příklad zahrnuje jen několik písmen s diakritikou. Zbývající znaky můžete doplnit samostatně. Číselné označení písmene je samozřejmě zcela libovolné a záleží na programátorovi. Důležité je pouze nevynechat žádné písmeno, které se v textu může vyskytnout. Následuje jedna z možných verzí funkce *index*:

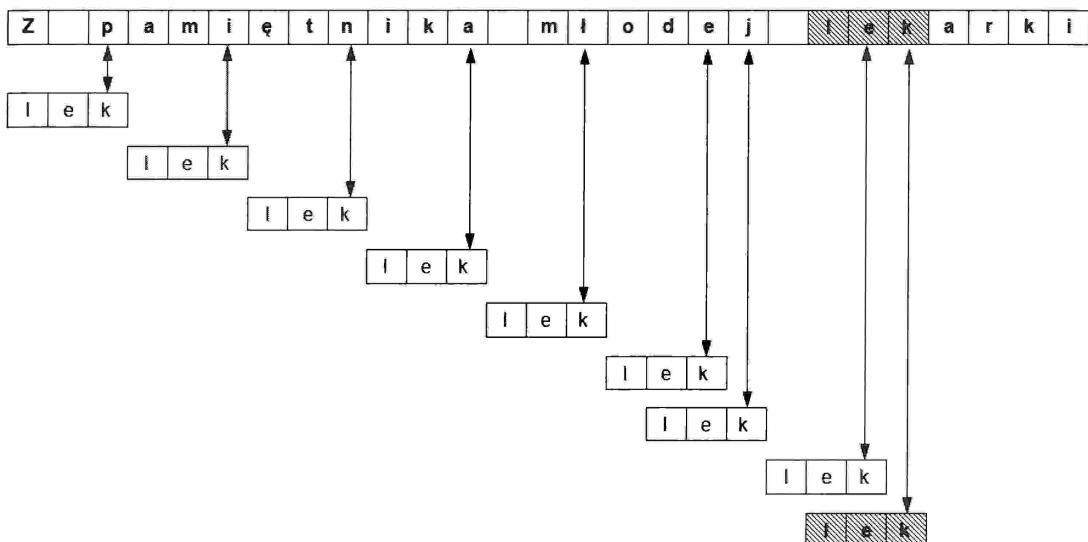
```
bm.cpp
const int K=26*2+2*2+1;           // znaky ASCII + písmena s diakritikou + mezera
int shift[K];
-----
int index(char c)
{
```

¹¹ Název rozhlasového cyklu polské spisovatelky Ewy Szumańské.

```

switch(c)
{
    case ' ':return 0;           // mezera=0
    case 'á':return 53;          // česká písmena
    case 'Á':return 54;
    case 'č':return 55;
    case 'Č':return 56;          // atd. pro zbývající česká písmena
    default:
        if(islower(c))
            return c-'a'+1;
        else
            return c-'A'+27;
}

```



Obrázek 8.7: Prohledávání textu Boyer-Mooreovou metodou

Funkce `index` má pouze pomocnou roli. Umožňuje mj. správně inicializovat pole posunutí. Po seznámení s příkladem na obrázku 8.7 by nás již způsob inicializace neměl příliš překvapit:

```

void init_shifts(char *w)
{
    int M=strlen(w);
    for(int i=0; i<K; i++)
        shift[i]=M;
    for(int i=0; i<M; i++)
        shift[index(w[i])]=M-i-1;
}

```

Nyní se konečně dostáváme k prezentaci kódu samotného algoritmu, který zahrnuje i příklad volání:

```

int bm(char *w, char *t)
{
    init_shifts(w);
    int i, j,N=strlen(t),M=strlen(w);

```

KAPITOLA 8 Prohledávání textů

```
for(i=M-1, j=M-1; j>0; i--, j--)  
    while(t[i]!=w[j])  
    {  
        int x=shift[index(t[i])];  
        if(M-j>x)  
            i+=M-j;  
        else  
            i+=x;  
        if (i>=N)  
            return -1;  
        j=M-1;  
    }  
    return i;  
  
int main()  
{  
    char *t="Zemský ráj to na pohled";  
    cout << "Výsledek vyhledávání=" << bm("hle", t) << endl;  
}
```

Boyer-Mooreův algoritmus patří podobně jako algoritmus K-M-P do třídy M+N, avšak předčí jej natolik, že v případě krátkých vzorů a dlouhé abecedy končí svou činnost již po přibližně M/N porovnáních. Kvůli výpočtu optimálních posunutí¹² autoři algoritmu navrhují, že jej lze zkombinovat s algoritmem K-M-P. Lze ovšem pochybovat o tom, zda je takový postup účelný, protože při optimalizaci samotného algoritmu můžeme velmi snadno zvýšit časovou náročnost vlastního procesu předběžné komplikace vzoru.

Karp-Rabinův algoritmus

Poslední algoritmus prohledávání textů, který budeme analyzovat, vyžaduje znalosti kapitoly 7 a terminologie, s níž jsme se tam seznámili.

Karp-Rabinův algoritmus je založen na dosti neobvyklé myšlence:

- Vzor w (který chceme najít) je *klíčem* (viz terminologii hešování v kapitole 7) s délkou M znaků, který se vyznačuje určitou hodnotou námi vybrané funkce H . Můžeme tedy jednorázově vypočítat $H_w=H(w)$ a výsledek výpočtu používat opakováně.
- Vstupní text t (k prohledání) můžeme odečítat takovým způsobem, abychom měli neustále k dispozici M posledních znaků¹³. Z těchto M znaků průběžně počítáme $H_t=H(t)$.

Budeme-li předpokládat, že je vybraná funkce H jednoznačná, můžeme shodnost vzoru s aktuálně zkoumaným fragmentem textu převést na otázku: Rovnají se čísla H_w a H_t ? Pozorný čtenář asi na tomto místě oprávněně zavrtí hlavou: To přece nemůže fungovat rychle! Na první pohled se zdá, že dodatečný výpočet funkce H pro každé vstupní slovo délky M bude algoritmicky minimálně stejně náročný jako běžné testování textu po jednotlivých znacích (např. pomocí algoritmu *vyhledávání hrubou silou*). Tím spíše, že jsme se zatím vůbec nezmínili o funkci H , kterou přitom potřebujeme. Z předchozí kapitoly si jistě pamatujeme, že výběr této funkce vůbec není jednoduchý.

12 Uvažujme např. opakovaný výskyt stejného písmene ve vzoru.

13 Úplně na začátku to samozřejmě bude M prvních znaků textu.

Popsaný algoritmus však existuje a kromě toho je skutečně rychlý! Aby do sebe všechno logicky zapadlo, určitě se neobejdeme bez nějakého triku. Vtip spočívá ve vhodném výběru funkce H . Karp s Rabinem zvolili takovou funkci, která díky svým vlastnostem umožňuje dynamicky využívat výsledky výpočtů provedených v předchozím kroku, což značně zjednoduší následnou fázi výpočtu. Řekněme, že posloupnost M znaků budeme interpretovat jako jisté celé číslo. Vezmeme-li jako základ systému počet všech možných znaků b , dostaneme:

$$x = t[i]b^{M-1} + t[i+1]b^{M-2} + \dots + t[i+M-1]$$

Přejděme nyní v textu o jednu pozici vpřed a zkontrolujme, jak se změní hodnota x :

$$x' = t[i+1]b^{M-1} + t[i+2]b^{M-2} + \dots + t[i+M]$$

Když se pozorně podíváme na čísla x a x' , zjistíme, že hodnota x' se do značné míry skládá z prvků, které tvoří hodnotu x – kvůli posunutí vynásobených číslem b . Snadno tedy odvodíme, že:

$$x' = (x - t[i]b^{M-1}) + t[i+M]$$

Jako funkci H použijeme funkci $H(x) = x \% p$ dobré známou z předchozí kapitoly, kde p je velké prvočíslo. Předpokládejme, že pro danou pozici i známe hodnotu $H(x)$. Když se v textu posuneme o jednu pozici doprava, musíme vypočítat hodnotu funkce $H(x')$ pro toto „nové“ slovo. Opravdu musíme celý výpočet opakovat? Nedokážeme si jej náhodou usnadnit díky tomu, že mezi čísla x a x' existuje určitá závislost?

Pomůžeme si přitom vlastností funkce `modulo`, která je součástí aritmetického výrazu. Funkci `modulo` lze samozřejmě vypočítat z koncového výsledku, ale to občas není příliš výhodné, protože pracujeme se značně velkým číslem – a kde bychom navíc dosáhli časové úspory? Shodný výsledek však dostaneme, když funkci `modulo` aplikujeme po každé částečné operaci a získanou hodnotu přeneseme do následného částečného výrazu! Jako příklad uvedeme výpočet:

$$(5 \cdot 100 + 6 \cdot 10 + 8) \% 7 = 568 \% 7 = 1$$

Tento výsledek je samozřejmě správný, jak se můžeme snadno přesvědčit na kalkulačce. Totožný výsledek však poskytne i následující sekvence výpočtů:

$$(5 \cdot 100) \% 7 = 3 \quad \boxed{(3 + 6 \cdot 10)} \% 7 = 0 \quad \boxed{(0 + 8)} \% 7 = 1$$

což můžeme rovněž rychle ověřit.

Implementace algoritmu je jednoduchá, i když obsahuje několik instrukcí, které je vhodné vysvělit. Podívejme se na výpis:

```
rk.cpp
const long p=33554393; // velké prvočíslo
const int b=64; // velká + malá písmena + "něco navíc"
//-----

int rk(char w[],char t[])
{
    unsigned long i,bM_1=1, Hw=0, Ht=0, M=strlen(w), N=strlen(t);
    for(i=0; i<M; i++)
    {
        Hw=(Hw*b+index(w[i]))%p; // iniciování funkce H pro vzor
        Ht=(Ht*b+index(t[i]))%p; // iniciování funkce H pro text
    }
    for(i=1; i<M; i++)
```

KAPITOLA 8 Prohledávání textů

```
bM_1=(b*bM_1)%p;
for(i=0; Hw!=Ht; i++)
{
    Ht=(Ht+b*p-index(t[i])*bM_1)%p;      // posun v textu
    Ht=(Ht*b+index(t[i+M]))%p;
    if (i>N-M)
        return -1;                         // vyhledávání nebylo úspěšné
}
return i;
}
```

V první fázi počítáme počáteční hodnoty H_t a H_w . Vzhledem k tomu, že znakové řetězce musíme interpretovat jako čísla, neobejdeme se bez funkce `index`, kterou již dokonale známe (viz stranu 200). Hodnota H_w se nemění a není potřeba ji aktualizovat. To ovšem neplatí pro aktuálně zkoumaný fragment textu – zde se hodnota H_t mění při každé inkrementaci proměnné i . K výpočtu $H(x')$ můžeme využít dříve zmíněnou vlastnost funkce `modulo`, což zajišťuje třetí cyklus `for`. Další vysvětlení si možná zaslouží i řádek označený hvězdičkou (*). Přičtení hodnoty $b*p$ k číslu H_t zaručuje, že se náhodou nedostaneme do záporných čísel. Pokud by k tomu opravdu došlo, hodnota `modulo` přenesená do následujícího aritmetického výrazu by nebyla správná a vedla by k chybnému konečnému výsledku.

Za pozornost dále stojí parametry p a b . Doporučuje se jako p volit velké prvočíslo¹⁴, ale na druhou stranu to nemůžeme přehánět, protože bychom mohli překročit rozsah použitých proměnných. Při výběru velkého čísla p se snižuje pravděpodobnost, že dojde ke kolizi kvůli nejednoznačnosti funkce H . Tato možnost je sice málo pravděpodobná, ale musíme ji mít na paměti. Když funkce `rk` vrátí určitý index i , měl by opatrny programátor provést dodatečný test rovnosti w a $t[i] \dots t[i+M-1]$.

Co se týče základu systému (v programu jej zastupuje proměnná b), je vhodné vybrat spíše hodně velké číslo, které však patří k mocninám dvojkdy. Operaci násobení číslem b pak můžeme implementovat jako bitový posun, který počítač realizuje mnohem rychleji než běžné násobení. Například pro $b = 64$ můžeme násobení $b*p$ zapsat jako $p<<6$.

Pro úplnost můžeme ještě dodat, že pokud nedojde ke kolizi (typický případ), funguje *Karp-Rabinův* algoritmus v čase úměrném součtu $M+N$.

14 V našem případě se jedná o číslo 33 554 393.

KAPITOLA 9

Pokročilé programovací techniky

V této kapitole:

- Programování typu „rozděl a panuj“
- „Hladové“ algoritmy
- Dynamické programování
- Jiné programovací techniky
- Bibliografické poznámky

V předchozích kapitolách¹ jsme představili užitečné programovací postupy. Seznámili jsme se s mnoha zajímavými datovými strukturami a především jsme se naučili uplatňovat rekurzivní techniky, které tvoří základ moderního programování. Speciálně jsme neupozorňovali na zásadní roli rekurze v celém procesu *koncepce* programů, ale soustředili jsme se spíše na důkladný popis technické stránky tohoto mechanizmu.

V této kapitole budeme využítí rekurze více zdůrazňovat. Většina metod, které představíme, totiž vychází právě z této programátorské techniky.

Tematika této kapitoly je poněkud široká a při troše nepozornosti bychom v ní mohli snadno ztratit orientaci. Budeme se totiž zabývat tzv. programovacími *technikami* (jinak též řečeno *metodami*), které mají mimořádně obecný charakter a lze pomocí nich programově řešit téměř libovolné myslitelné problémy. Příslušné algoritmy (či spíše jejich vzory) budeme ilustrovat na velmi odlišných úlohách a zpravidla přitom dosahneme velmi efektních výsledků. Můžeme přitom snadno získat dojem, že jsme dostali univerzální recepty, které *automaticky* umožní vyřešit všechny úkoly, s nimiž jsme si dříve neporadili. Jistě už tušíme, že to tak úplně nebude pravda. Klamný dojem, kterému bychom mohli podlehnout (nebýt tohoto varování), vzbuzuje vhodně vybrané příklady, které se dokonale hodí k právě probírané metodě. Praxe je však obecně mnohem složitější, a pokusíme-li se tyto programovací techniky nasadit jako univerzální „kuchařské recepty“, příliš neuspějeme. Znamená to snad, že jsou tyto metody chybné? To jistě nejsou, ale všechny pokusy o jejich bezmyšlenkovitou aplikaci jsou odsouzeny k nezdaru. Nejdříve je nutné *metodu* *přizpůsobit* konkrétnímu algoritmickému problému.

Měli bychom si uvědomit, že ke každému úkolu je potřeba přistupovat tvůrčím způsobem. Programátor, který disponuje určitými schopnostmi (získanými teoreticky i prakticky), je dokáže vhodně využít, ale přitom si uvědomuje, že univerzální postupy (v zásadě) neexistují. Stejně jako v jiných oblastech se ani v algoritmice nedělí zázraky (chtěli bychom dodat: bohužel...).

¹ Zejména v kapitole 2 o rekurzi a v kapitole 5 věnované datovým strukturám.

Programování typu „rozděl a panuj“

Programování typu „rozděl a panuj“² využívá základní vlastnosti rekurze: nejdříve problém rozložíme na určitý konečný počet podproblémů stejného typu. Poté musíme získaná částečná řešení jistým způsobem zkombinovat a získat tím globální řešení. Označíme-li řešený problém jako Pb a velikost dat písmenem N , můžeme výše uvedený postup symbolicky zapsat takto:

$$Pb(N) \rightarrow Pb(N_1) + Pb(N_2) + \dots + Pb(N_k) + KOMB(N).$$

Problém řádu N jsme rozdělili na k podproblémů.



Upozornění: Funkce $KOMB(N)$ není rekurzivní.

Důležité je, že znak $+$ zde nepoužíváme v aritmetickém smyslu. Pokud však budeme uvažovat s ohledem na čas provedení programu (viz označení z kapitoly 3), můžeme symbol \rightarrow zaměnit znakem *rovnosti* a získaná rovnice bude platit.

Uvedené upozornění má pro probíranou programovací techniku zásadní význam. Problémy totiž v žádném případě nedělíme z estetických důvodů (i když samozřejmě můžeme zohlednit i tento aspekt), ale cílem je zvýšit efektivitu programu. Jinak řečeno: jede nám o *urychlení* algoritmu.

Technika „rozděl a panuj“ v mnoha případech umožňuje změnit třídu algoritmu (např. z $O(n)$ na $O(\log N)$ atd.). Na druhou stranu existuje skupina úloh, které metodou „rozděl a panuj“ nelze výrazněji urychlit. Z kapitoly 3, „Analýza složitosti algoritmů“, víme, jak porovnávat jednotlivé algoritmy. Nejdříve bychom tedy měli vzít do ruky papír a tužku, abychom zjistili, zda má vůbec smysl usednout ke klávesnici.

Následuje formální zápis metody, který využívá programovací pseudojazyk:

```
rozdel_a_panuj(N)
{
    pokud číslo N je dostatečně malé
        vrat Elementarni_Pripad(N);
    v opačném případě
    {
        "Rozděl" Pb(N) na k menších exemplářů:
        Pb(N1), Pb(N2) ... Pb(Nk);
        pro i=1...k
            Vypočítej částečný výsledek wi=rozdel_a_panuj(Ni);
        vrat KOMB(w1, w2, ... wk);
    }
}
```

Přesný význam formulací: „dostatečně malé“ a „elementární případ“ úzce souvisí s řešeným problémem a těžko zde můžeme uvést nějaké zobecnění. Případné nejasnosti by se měly vysvětlit při analýze příkladů, které patří do následujících částí kapitoly.

2 Tento termín, který se rozšířil v anglicky psané literatuře, příliš neodpovídá Machiavelliho myšlence vyjádřené jeho výrokem „Divide ut Regnes“ (ten má nepochybně negativní konotace). Zdá se však, že si toho už téměř nikdo nevšímá.

Vyhledávání minima a maxima v číselném poli

S metodou „rozděl a panuj“ jsme se v této knize již vícekrát nevědomky setkali. Každý čtenář může nyní vyzkoušet svůj postřeh a orientaci v problematice a pokusit se příslušné algoritmy najít (návod se nachází na konci této podkapitoly).

Nejdříve prozkoumáme příklad poměrně jednoduchého problému vyhledávání *největšího a nejménšího* prvku v poli. Problém je vcelku banální, ale má poměrně velký praktický význam. Představme si například, že chceme na obrazovce znázornit průběh funkce $y = f(x)$. Za tímto účelem uložíme do jistého pole vypočítaných N hodnot té funkce z intervalu řekněme $x_d \dots x_g$. Sadu bodů potřebujeme vykreslit tak, aby se celý graf vešel na viditelnou plochu obrazovky. Osa OX nepředstavuje žádný problém – budeme předpokládat, že x_d odpovídá souřadnici 0 a obdobně x_g maximálnímu horizontálnímu rozlišení. Chceme-li však určit měřítko osy OY , musíme znát extrémní hodnoty funkce $f(x)$. Teprve pak si můžeme být jisti, že se graf určitě vejde na viditelnou plochu obrazovky.

Cvičení 1

Odvoďte schéma převodu hodnot (x_i, y_i) na obrazkové souřadnice (x_{obr}, y_{obr}) , známe-li rozlišení grafické obrazovky X_{max} a Y_{max} a zároveň extrémní hodnoty funkce $f(x)$, které označíme jako f_{min} a f_{max} .

Vraťme se nyní k samotnému zadání. Chceme-li realizovat algoritmus vyhledávání minimálních a maximálních hodnot v jistém poli³, nejdříve nás napadne, že bychom je mohli prohledat lineárně:

```
min-max.cpp
const int n=12;
int tab[n]={23,12,1,-5,34,-7,45,2,88,-3,-9,1};
void min_max1(int t[], int& min, int& max)
{
    // používat pouze pro n>=1!
    min=max=t[0];
    for(int i=1; i<n; i++)
    {
        if(max<t[i])          // (*)
            max=t[i];
        if(min>t[i])          // (**)
            min=t[i];
    }
}
```

Předpokládejme, že pole má délku n , tzn. obsahuje prvky od $t[0]$ do $t[n-1]$. Vypočítejme praktickou výpočetní složitost tohoto algoritmu, přičemž za časově nejnáročnější budeme považovat instrukce porovnání. Okamžitě dostáváme výsledek: $T(n)=2(n-1)$, takže program patří do třídy $O(n)$. Algoritmus je jednoduchý a... neefektivní. Když se na proceduru podíváme pozorněji, můžeme si dokonce všimnout, že po úspěšném výsledku porovnání (*) zbytečně provádí porovnání (**). Když hned za řádek (*) doplníme instrukci `else`, v nejhorším případě proběhne $2(n-1)$ porovnání a v nejlepším případě jen $n-1$. Třída algoritmu se tím samozřejmě nezmění, ale program se může poněkud urychlit – samozřejmě v závislosti na konfiguraci vstupních dat.

Vytvořme nyní analogickou proceduru, která bude využívat rekurzivní zjednodušení algoritmu podle zásady „rozděl a panuj“. Vycházíme z tohoto principu:

³ V příkladu se jedná o pole celých čísel, což nijak neomezuje obecnou platnost algoritmu.

KAPITOLA 9 Pokročilé programovací techniky

Obecný případ:

- Pokud má pole délku 1, hodnoty `min` a `max` se rovnají jedinému prvku, který se v poli nachází.
- Pokud pole obsahuje dva prvky, můžeme hodnoty `min` a `max` snadno určit jejich porovnáním.

Obecný případ:

- jestliže má tabulka délku > 2, pak:
 - Rozděl ji na dvě části.
 - Vypočítej hodnoty (`min1, max1`) a (`min2, max2`) obou těchto částí.
 - Jako výsledek vrať `min=min(min1, min2)` a `max=max(max1, max2)`.

Rekurzivní procedura, která odpovídá tomuto popisu, vypadá takto:

```
void min_max2(int left, int right, int t[], int& min, int& max)
{
    if (left==right)
        min=max=t[left]; // jeden prvek
    else
        if (left==right-1) // dva prvky
            if(t[left]<t[right])
            {
                min=t[left];
                max=t[right];
            }
            else
            {
                min=t[right];
                max=t[left];
            }
        else
        {
            int temp_min1, temp_max1, temp_min2, temp_max2, mid;
            mid=(left+right)/2;
            min_max2(left, mid, t, temp_min1, temp_max1);
            min_max2(mid+1, right, t, temp_min2, temp_max2);
            if(temp_min1<temp_min2) // (1)
                min=temp_min1;
            else
                min=temp_min2;
            if(temp_max1>temp_max2) // (2)
                max=temp_max1;
            else
                max=temp_max2;
        }
}
```

Když proceduru porovnáme s obecným schématem metody, můžeme si všimnout, že:

- „Dostatečně malý“ zde znamená rozměr pole 1 či 2.
- Máme dva elementární případy.
- Dělíme pole na dvě stejné části N1 a N2.

- Jako částečné výsledky dostaneme dvojice: (`temp_min1, temp_max1`) a (`temp_min2, temp_max2`).
- Funkce KOMB pouze porovnává částečné výsledky – její roli plní instrukce, které jsou ve výpisu označeny komentáři (1) a (2).

V dalších příkladech již nebudeme procedury rozebírat natolik podrobně. Budeme předpokládat, že to každý čtenář v případě potřeby zvládne sám.

Ani zjištění výpočetní složitost procedury `min_max2` není nijak těžké. Dekompozice problému vypadá následovně:

$$T(n) \rightarrow 2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2 + 2T\left(\frac{n}{2}\right)$$

Je to logaritmický rozklad, jaký už známe. Po jeho vyřešení dostaneme výsledek $T(n) \in O(n)$ (viz kapitolu 3).

Bez problémů vypočítáme také praktickou složitost $T(n)$ tohoto programu. Jestliže vhodně rozšíříme rovnici:

$$T(n) = 2 + 2T\left(\frac{n}{2}\right) = 2 + 2\left\{2 + T\left(\frac{n}{4}\right)\right\} = atd.$$

a budeme předpokládat, že existuje určité k takové, že $n = 2^k$, získáme stejnou rovnici jako v případě procedury `min_max1`. Nemělo by nás to nijak překvapit. Uvědomme si totiž, že druhá verze má úplně stejný úkol jako ta předchozí – i když přitom postupuje jiným způsobem.

Není na předchozím výsledku něco znepokojivého? Zdá se totiž, že nová metoda nezaručuje zvýšení efektivity algoritmu!

Setkáváme se s případem problému, o kterých jsme se již zmínili na úvod, kdy nasazení metody „rozděl a panuj“ zásadním způsobem nemění časové parametry programu. Můžeme se alespoň utěšovat, že je nezhoršuje. V našem případě to bohužel není pravda. Připomeňme si „temné stránky“ rekurze, o kterých jsme se zmínili v příslušné kapitole. Pak bychom si měli jasně uvědomit, že nová verze časové parametry programu skutečně zhorší, protože klade větší nároky na paměť (zálohování rekurzivních volání) a vyžaduje obsluhu nezanedbatelného počtu dodatečných rekurzivních volání.

Uvedený příklad neukazuje rozebíranou metodu právě v nejlepším světle. Uvedli jsme jej však záměrně, aby bychom naznačili potenciální rizika spojená s naivní vírou v zázračné postupy. V některých případech samozřejmě metoda „rozděl a panuj“ dělá opravdové divy (několik ukázek zkrátka představíme). Přesvědčíme se o tom však teprve poté, co určíme výpočetní složitost obou metod. Jestliže skutečně zjistíme, že by rychlosť měla značně vzrůst (například díky změně na lepší třídu algoritmu), je málo pravděpodobné, že převáží livil jistých nevýhodných vlastností rekurze. Získáme-li ovšem výsledek, který svědčí o časové rovnocennosti obou metod, musíme vzít v úvahu rovněž faktory, které sice nesouvisejí s teorií, ale hrají nezanedbatelnou roli v praxi (v reálném světě). Některá upozornění je vhodné připomínat, aby bychom se později nedivili, že počítač nedělá to, co chceme (nebo to dělá hůře, než jsme chtěli).

Násobení matic s rozměry NxN

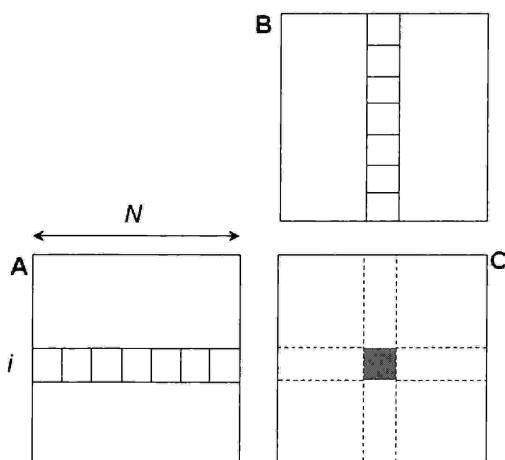
V mnoha problémech numerické povahy je často potřeba násobit matice, což je z principu dosti časově náročná operace. Součin dvou maticí se počítá způsobem, který je symbolicky znázorněn na obrázku 9.1.

KAPITOLA 9 Pokročilé programovací techniky

Pokud matici C budeme považovat za výsledek násobení $A \times B$ (připomeňme, že z pohledu programátora matice odpovídá dvouzměrnému poli), můžeme libovolný prvek $C[i,j]$ získat na základě tohoto schématu:

$$C_{i,j} = \sum_{k=1}^N A_{i,k} B_{k,j}$$

(Násobíme odpovídající prvky řádku i a sloupce j a přitom kumulujeme částečné součty.)



Obrázek 9.1: Násobení matic

Když zohledníme počet prováděných operací násobení (lze oprávněně předpokládat, že jsou zde nejnákladnější), budou se náklady na výpočet jednoho prvku matice C samozřejmě rovnat N . Vzhledem k tomu, že všech prvků je N^2 , budou celkové náklady N^3 , a algoritmus tedy bude patřit do třídy $O(N^3)$.

Algoritmus je velmi náročný, ale dlouho se zdálo, že se tomu nedá vyhnout, a všichni se s tím smířili. V roce 1968 však všechny pořádně překvapil Volker Strassen. Objevil algoritmus založený na principu „rozděl a panuj“, který byl lepší než zdánlivě „nenahraditelný“ algoritmus třídy $O(N^3)$.

Označme prvky matic A , B a C následujícím způsobem:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Lze snadno dokázat, že platí tyto rovnosti:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Zdá se, že metodu „rozděl a panuj“ bychom mohli okamžitě uplatnit přístupem, který je založen na rozdělení obou matic A a B na 4 stejné části (pochopitelně za předpokladu, že N je mocninou čísla 2) a vynásobení matic menšího řádu. Avšak vzhledem k tomu, že toto rozdělení nám nijak neušetří

práci (v další fázi musíme provést naprosto totéž, co dělal iterativní algoritmus), takto efektivnější algoritmus určitě nezískáme. Toto tvrzení není podloženo výpočty, ale nepochybně je pravdivé.

Podívejme se nyní, jak násobení matic optimalizoval V. Strassen. Základní princip jeho metody spočívá v zavedení dodatečných proměnných, což jsou matice řádu $N/2$ označené P, Q, R, S, T, U, V , které umožňují uložit výsledky následujících kalkulací⁴:

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) & * , + , + \\ Q &= (A_{21} + A_{22})B_{11} & + , * \\ R &= A_{11}(B_{12} - B_{22}) & * , - \\ S &= A_{22}(B_{21} - B_{11}) & * , - \\ T &= B_{22}(A_{11} + A_{12}) & * , + \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) & * , + , - \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) & * , + , - \end{aligned}$$

Když získáme tyto částečné výsledky, můžeme matici C sestavit pomocí těchto dosazení:

$$\begin{aligned} C_{11} &= P + S - T + V & + , - , + \\ C_{12} &= R + T & + \\ C_{21} &= Q + S & + \\ C_{22} &= P + R - Q + U & + , - , + \end{aligned}$$

Algoritmus tohoto tvaru vyžaduje 7 operací násobení a dodatečných 18 operací sčítání nebo odečítání.

Všimněme si, že *Strassenův* algoritmus inteligentním způsobem přenáší zátěž výpočtů ze zpravidla nákladné operace násobení⁵ na značně rychlejší sčítání nebo odečítání.

Rekurzivní rozklad ve *Strassenově* algoritmu vypadá takto:

$$T(n) = 7T\left(\frac{N}{2}\right) + cN^2 \in O(N^{2.81})$$

(c je určitá konstanta).

Při podrobnějších praktických testech tohoto algoritmu se ukázalo, že reálný zisk výše uvedené metody lze zaznamenat při násobení matic pro N v řádu desítek, ale v případě opravdu velkých hodnot N (např. nad 100) se již efektivita algoritmu znova přibližuje jeho iterativnímu konkurentu. Tento jev záleží mj. na způsobu správy paměti v daném výpočetním systému. Jestliže se jedná o osobní počítač s velkou „soukromou“ kapacitou paměti, bude se výkon algoritmu blížit teoretickým odhadům. V případě distribuovaných systémů, kde program celou požadovanou paměť „vidí“ jako jeden zdánlivý celek, ale ve skutečnosti pracuje s jejími stránkami, které mu operační systém v případě potřeby poskytuje, však situace může vypadat poněkud méně příznivě. Další podstatnou přičinou poklesu efektivity algoritmu pro velké (prakticky se vyskytující) hodnoty N je kumulace mnoha rekurzivních volání a vzrůstající náročnosti operací sčítání a odečítání. Z výše zmíněných důvodů bychom měli *Strassenův* algoritmus považovat spíše za zajímavý výsledek teoretické práce, který má nepochybnou výukovou hodnotu.

⁴ Vedle rovnic jsou uvedeny aritmetické operace, které jsou potřebné k výpočtu dané rovnosti.

⁵ Zejména pokud jsou prvky matice reálná čísla (typu *float* a *double* v jazyce C++).

Násobení celých čísel

Další příklad patří rovněž do výpočetní kategorie: budeme se zabývat násobením celých čísel.

Násobení dvou celých čísel X a Y , jejichž vnitřní reprezentace má rozměr N bitů, je operace třídy $O(N^2)$. Předpokládáme přitom, že násobení probíhá klasicky, jak jsme se to učili na základní škole (scítáme pod sebou N výsledků částečných součinů, které vesměs patří do třídy $O(N)$).

V případě násobení celých čísel můžeme uplatnit metodu „rozděl a panuj“ na základě následujícího postřehu:

$$\begin{aligned} X &= [A \ B] = A \cdot 2^{\frac{N}{2}} + B \\ Y &= [C \ D] = C \cdot 2^{\frac{N}{2}} + D \end{aligned}$$

A a B spolu s C a D označují poloviny binárních reprezentací čísel X a Y . Součin $X \cdot Y$ lze zapsat jako:

$$X \cdot Y = AC \cdot 2^{\frac{N}{2}} + (AD + BC) \cdot 2^{\frac{N}{2}} + BD$$

Budeme-li předpokládat, že N je mocninou čísla 2 (což v současných počítačích obecně platí), můžeme výpočetní složitost programu vyjádřit tímto způsobem:

$$\begin{aligned} T(1) &= 1 \\ T(N) &= 4T\left(\frac{N}{2}\right) + cN \end{aligned}$$

V těchto rovnicích zohledňujeme vliv čtyř nákladných operací násobení spolu s jistými náklady úměrnými číslu N , které souvisejí se scítáními a bitovými posuny⁶. Při výpočtu třídy tohoto algoritmu lze využít vzory uvedené v kapitole 3. Pokud se však nechceme příliš namáhat, můžeme jednoduše odhadnout, že se jedná o třídu $O(N^2)$.

Jak si můžeme být tak jisti? Vycházíme z pozorování během předchozí analýzy procedury `min_max`: problém jsme rozdělili, ale objem prací se tím nijak nezmenšil. Zázraky se tedy nedějí⁷. Než však metodu „rozděl a panuj“ definitivně zavrhнемe, podívejme se na následující vztah, který vyjadřuje násobení $X \cdot Y$ poněkud jiným způsobem než výše:

$$X \cdot Y = AC \cdot 2^{\frac{N}{2}} + [(A - B)(D - C) + AC + BD]2^{\frac{N}{2}} + BD$$

Kromě poněkud komplikovanější podoby (viz Strassenův algoritmus) jsme omezili počet operací násobení z 4 na 3 (členy AC a BD se vyskytují dvakrát, takže podruhé můžeme využít předchozí výsledek). Rekurzivní vzorec související s tímto rozkladem je shodný s předchozím případem. Stačí jen zaměnit číslo 4 číslem 3. Díky této informaci okamžitě dostáváme, že algoritmus patří do třídy $O(N \log_2 3) = O(N^{1.59})$.

Můžete sami prozkoumat různé příklady a vycházet přitom z různých odhadů ohledně nákladů elementárních operací (+, -, bitový posun), kdy tento algoritmus v porovnání s klasickou metodou může skutečně přinést značné zisky.

⁶ Připomeňme, že násobení čísel mocninou základu číselného systému ($2^1, 2^2, 2^3\dots$) odpovídá přesunu vnitřní reprezentace o tolik míst vlevo, kolik odpovídá exponentu mocniny (1, 2, 3...).

⁷ Pro pochybovače uvedeme matematický důkaz: $T(n) \in O(N \log_2 4) = O(N^2)$.

Jiné známé algoritmy typu „rozděl a panuj“

I když jsme to speciálně nezdůrazňovali, už v předchozích kapitolách jsme se mohli setkat s několika zajímavými algoritmy, které lze zařadit do kategorie „rozděl a panuj“. Mezi nimi nepochybně vyniká slavný algoritmus *Quicksort* (viz kapitolu 4). Dokáže výrazně urychlit třídění a přitom se vyznačuje nebývale jednoduchým zápisem i koncepcí.

Jako dobrý příklad inteligentní dekompozice problému lze uvést i problém *hanojských věží*, který jsme rozebírali v souvislosti s tématem metod odstraňování rekurze (viz kapitolu 6). Verze iterativní i rekurzivní verze sice patří do stejné třídy, ale pro volbu rekurzivní metody mluví hlavně její jednoduchý zápis.

Mezi metody typu „rozděl a panuj“ je možné zařadit i proceduru *binárního vyhledávání*, i když se její princip poněkud liší od schématu, které jsme uvedli v předchozí části této kapitoly. Na zmíněné proceduře lze vhodně ukázat, jak může dobrý algoritmus urychlit řešení problému, pro který známe jednoduchou, ale neefektivní metodu (v tomto případě se jedná o lineární vyhledávání).

„Hladové“ algoritmy

Nová metoda má poněkud divný název, ale v odborné literatuře se metody určité třídy skutečně označují jako „hladové“ či „žravé“ (ang. *greedy*). Tyto algoritmy umožňují najít obecný *postup* řešení daného problému. Příslušné postupy musí zohledňovat jistá východiska (omezení), která mohou například požadovat, aby bylo řešení optimalizované podle určitých kritérií. Při konstrukci těchto postupů vycházíme z řady možností, které tvoří množinu vstupních dat. „Hladové“ algoritmy se vyznačují zejména tím, že v každé fázi hledání řešení vybírají *lokálně* optimální možnost. V závislosti na výběru těchto možností může být optimální i konečné globální řešení, ale není to zaručeno. Popsaná metoda nejvíce vyhovuje určité třídě úloh optimalizační povahy, kdy chceme například najít nejkratší cestu v grafu, určit nevhodnější pořadí jistých počítačových operací atd.

Metoda „hladových“ algoritmů je v souladu s lidskou povahou. Když lidé dostanou nějaký úkol, často se totiž spokojí s řešením, které nutně nemusí být optimální, ale zato je získají rychle.

Algoritmus má následující obecné schéma:

```
hladový(W)
{
    ŘEŠENÍ= Ø ; // prázdná množina
    dokud(ne Nalezeno(ŘEŠENÍ) a W ≠ Ø)
        proved:
        {
            X=Výběr(W);
            jestliže Vyhovuje(X) pak ŘEŠENÍ = ŘEŠENÍ ∪ {X};
        }
        jestliže Nalezeno(ŘEŠENÍ)
            vrat ŘEŠENÍ;
        v opačném případě;
            vrat "řešení neexistuje"
}
```

V popisu metody se používají tyto symboly:

W – množina vstupních dat.

ŘEŠENÍ – množina, na jejímž základě se konstruuje řešení.

x – prvek množiny.

$\text{výběr}(A)$ – funkce, která zajišťuje „optimální“ výběr prvku z množiny A (přičemž prvek odstraní).

$\text{vyhovuje}(X)$ – lze výběrem prvku X zkompletovat částečné řešení tak, aby bylo možné najít alespoň jedno globální řešení?

$\text{nalezeno}(R)$ – je R řešením úlohy?

Z uvedeného zápisu je jasné, jak metoda získala svůj název: v každé fázi vybírá nejlepší kousek a přitom se příliš nestará o budoucnost. Podívejme se na několik příkladů využití nové metody.

Problém batohu aneb těžký život pěšího turisty

Vzijme se nyní do role turisty, který se chystá na dlouhý pěší pochod po horách. Aby nás příklad nebyl příliš obecný, řekněme, že máme za úkol dojít na vrchol nějaké hory v Tatrách. Jedná se o smluvně místo setkání s přáteli, s nimiž chceme uspořádat piknik. Do cílového místa směruje celkem pět osob. Každá z nich slíbila, že přinese hodně jídla, aby byl plánovaný piknik co nejbohatší. Nebudeme zabíhat do přílišných podrobností a pokoušet se odhadnout, co nesou zbývající čtyři známí. Zaměříme se jen na svůj osobní úkol, který řešíme během přípravy na cestu. Předpokládejme, že nás přátelé pověřili tím, abychom přinesli několik druhů chutných sýrů. Nějak nás ovšem nenapadá, jak je naskládat do volné části batohu.

Podle údajů výrobce má nás batoh zaručený objem 60 litrů, z čehož nám na potraviny zůstalo $M = 20$ litrů. Většinu místa už zabírají předměty, které potřebujeme k přežití v horách. Zbývá nám tedy vyřešit dilema, jak vyplnit zbytek batohu. Chceme vzít celkem tři druhy sýra (s_1, s_2 a s_3). V domácí ledničce máme těchto sýrů w_1, w_2 a w_3 litrů. Každý ze sýrů chutná skvěle, nicméně můžeme jim přiřadit orientační ceny c_1, c_2 a c_3 , což nám umožní seřadit jednotlivé druhy podle jakosti.

Naším cílem je vzít od každého druhu sýra takové množství:

$$0 \leq x_1, x_2, x_3 \leq 1,$$

abychom v součtu nepřekročili maximální objem části batohu, která je pro sýry určena:

$$\sum_{i=1}^3 w_i x_i \leq M$$

a zároveň zabalili co nejhodnotnější náklad, tzn. maximalizovali funkci

$$\sum_{i=1}^3 c_i x_i$$

Abychom nezůstali u teoretických úvah, podívejme se na konkrétní příklad konfigurace dat:

$$w_1 = 16, \quad w_2 = 12, \quad w_3 = 10,$$

$$c_1 = 80, \quad c_2 = 70, \quad c_3 = 60.$$

Několik příkladů uvedených v tabulce 9.1 ilustruje, jak se liší potenciální řešení v případě netričná konfigurace vstupních dat (s takovou se setkáme, když je součet w_i menší než M – řešení takové situace může každý čtenář snadno odhadnout samostatně).

Ze třech ad hoc vymyšlených řešení je momentálně optimální to druhé, ale nic nám nezaručuje, že neexistují lepší konfigurace parametrů x_i .

Na tomto místě je nutné zdůraznit, že základní princip, z nějž procedura hladový vychází, nám nezaručuje optimální výsledek. Právě naopak: v typickém případě získáme řešení⁸, které bude jen *skoro optimální*!

Tabulka 9.1: Příklady řešení problému batohu

Č.	(x_1, x_2, x_3)	$\sum_{i=1}^3 c_i x_i$	$\sum_{i=1}^3 w_i x_i$
1	$\left(\frac{1}{1}, \frac{1}{4}, \frac{1}{10}\right)$	103,5	20
2	$\left(\frac{2}{3}, \frac{1}{2}, \frac{1}{3}\right)$	108,3	20
3	$\left(\frac{2}{3}, \frac{1}{5}, \frac{2}{3}\right)$	107,3	19,7

Když věci postavíme takovým způsobem, může nás popisovaná metoda rychle znechutit. Vzpořeme si však na to, co jsme uvedli na úvod této části kapitoly: při řešení problémů je někdy potřeba, abychom popisované algoritmy nejdříve přizpůsobili. Pokud se je pokusíme aplikovat bezmyšlenkovitě, pravděpodobně neuspějeme.

Lépe to ukážeme při analýze několika možných strategií řešení problému batohu pomocí „hladového“ algoritmu. První, zdánlivě optimální řešení spočívá v tom, že se pokusíme batoh vyplnit nejdražším sýrem (s_1) – pokud se jeho celkový objem vejde do volného prostoru, vezmeme celý kus. V opačném případě ukrojíme takový kus, abychom nepřekročili objem M a využili z tohoto sýra co největší část. Poté analogickým způsobem naložíme s druhým sýrem podle cenového pořadí atd.

Stačí ovšem „ručně“ vyzkoušet několik konfigurací, které pomocí této metody dostaneme, abychom se přesvědčili, že neposkytují nejlepší výsledky. Je to zcela zřejmé z analýzy tabulky 9.1, zejména pozic 1 a 2.

Řešení není optimální z poměrně prostého důvodu: koncový efekt (funkce, kterou chceme maximalizovat) nezávisí jen na aktuální hodnotě vkládaných sýrů, ale i na jejich objemu. Možná bychom se tedy místo parametru c_i měli nejdříve zaměřit na parametr w_i ?

Když ale s tužkou v ruce uděláme několik pokusů tohoto typu, nezískáme příliš působivé výsledky ani v tomto případě. A opět můžeme začít pochybovat, zda má celá metoda vůbec smysl...

Pokud k optimálnímu řešení nevede ani jedna z obou analyzovaných krajností, nezbývá nám nic jiného než změnit strategii a postupovat tak, abychom objektivně zohlednili oba parametry (w_i, c_i) zároveň. Ukazuje se, že pokud předběžně uspořádáme vstupní data takovým způsobem, abychom pro libovolné i zachovali vztah:

$$\frac{c_{i+1}}{w_{i+1}} \leq \frac{c_i}{w_i}$$

pak bude „hladový“ algoritmus směřovat k optimálnímu řešení. Abychom tuto příručku příliš nezatežovali matematikou, důkaz výše uvedeného tvrzení si odustíme, protože pro nás není podstatný.

Podívejme se na program v jazyce C++, který řeší naše batohové dilema:

8 Samozřejmě pokud existuje.

```

greedy.cpp
const int n=3;
void greedy(double M, double W[n], double C[n], double X[n])
{
    int i;
    double Z=M; // zbývá zaplnit
    for(i=0; i<n; i++)
    {
        if(W[i]>Z)
            break;
        X[i]=1;
        Z=Z-W[i];
    }
    if(i<n)
        X[i]=Z/W[i];
}
int main()
{
    double W[n]={10,12,16}, C[n]={60,70,80}, X[n]={0,0,0};
    greedy(20,W,C,X);
    double p=0;
    for(int i=0; i<n; p+=X[i]*C[i], i++)
        cout << i << "\t" << W[i] << "\t" << C[i] << "\t" << X[i] << endl;
    cout << "Dohromady:" << p << endl;
}

```

Ukazuje se, že optimálním řešením je vektor

$$X = \left(1, \frac{5}{6}, 0 \right)$$

– v takovém pořadí dat, v jakém jsou uvedena ve výpisu, kde jsme je již předběžně upravili podle výše popsaného vzoru.

Z analýzy problému batohu bychom si měli odnést následující poučení: Než začneme psát program ve svém oblíbeném programovacím jazyce (nemusí to být nutně C++), měli bychom věnovat několik minut analýze, která může výrazně přispět k jakosti konečného řešení.

Dynamické programování

Výhody rekurzivního programování se projevují tím, že umožňuje formulovat řešení snadno a přirozeně. Rekurze má bohužel i svou odvrácenou tvář, na kterou můžeme snadno zapomenout, budeme-li o ní uvažovat pouze v matematických kategoriích. Jde samozřejmě o to, jak rekurzivní postup skutečně proběhne v počítači, jaké budou výpočetní náklady na realizaci rekurzivních volání, návraty z těchto volání, kombinaci částečných výsledků atd.

Může se tedy ukázat, že formálně rychlý rekurzivní algoritmus (s ohledem na kategorii třídy O) bude mnohem pomalejší, než by vyplývalo z teoretických výpočtů.

S tímto problémem se můžeme vypořádat různými způsoby (viz např. kapitolu 6), mezi něž patří... psaní výhradně iterativních procedur.

Programátorská revoluce, která by zavedla všeobecný zákaz využití rekurze, však rozhodně nepatří k cílům této knihy. Postavme tedy otázku jiným způsobem: *Můžeme čerpat z výhod, které přináší rekurzivní formulace řešení, ale rekurzi přitom nepoužívat?*

I když to na první pohled může vypadat absurdně, je to možné. Právě na tomto postulátu – který zdánlivě nelze realizovat – je založena technika *dynamického programování*. Její objevitel Richard E. Bellman byl roku 1979 (více než 20 let po samotném objevu) vyznamenán cenou IEEE Medal of Honor.

Tato technika se mimořádně hodí k řešení problémů numerického typu:

- k počítání nejkratší cesty v grafu (na grafy se zaměříme v kapitole 10),
- k počítání jisté komplikované hodnoty vyjádřené pomocí rekurzivního vztahu.

Ve své původní verzi (jak ji publikoval R. Bellman) vychází metoda dynamického programování ze zásady zvané princip optimálnosti. Abychom celý výklad nekomplikovali, nebudeme zde tento princip vysvětlovat a uvedeme pouze jeho algoritmickou interpretaci.

Konstrukci programu, který využívá zásadu dynamického programování, lze formulovat ve třech fázích:

- **Koncepcie:**
 - Pro daný problém P vytvoříme rekurzivní model jeho řešení (spolu s jednoznačným vymezením elementárních případů).
 - Vytvoříme pole, které umožní ukládat řešení elementárních případů i řešení podproblémů, které na jejich základě vypočítáme.
- **Iniciace:**
 - Do pole zapíšeme číselné hodnoty, které odpovídají elementárním případům.
- **Progrese:**
 - Na základě číselných hodnot uložených v poli můžeme pomocí rekurzivního vzorce vypočítat řešení problému vyššího rádu a zapsat jej do pole.
 - Tímto způsobem postupujeme, dokud nedosáhneme požadované hodnoty.

Uvedený postup může působit poněkud záhadně. Jak ale ukážeme na příkladech, metoda skutečně není složitá. Než však přejdeme k ukázkám této programovací techniky, porovnejme ji s metodou „rozděl a panuj“, kterou již známe.

„*rozděl a panuj*“

- Problém rádu N rozkládáme na podproblémy nižšího rádu a řešíme je.
- Spojujeme řešení podproblémů, abychom získali globální řešení.

„*dynamické programování*“

- Na základě informací o řešení elementárního problému vypočítáme řešení problému vyššího rádu.
- Pokračujeme ve výpočtech, dokud nezískáme řešení rádu N .

Nová technika vzbuzuje dojem, že se jedná o optimální přístup: jednou nalezené řešení jistého podproblému zaregistrujeme v poli a v případě potřeby jej můžeme později využít. U metody „rozděl a panuj“ oproti tomu docházelo k tomu, že se stejné hodnoty počítaly opakováně.

Nově poznanou metodu ilustrujeme na několika příkladech různého stupně obtížnosti. Začneme přitom od problému, který již dokonale známe – od výpočtu prvků Fibonacciho posloupnosti (viz kapitolu 2).

Fibonacciho posloupnost

Znovu připomeňme, jak se Fibonacciho posloupnost definuje:

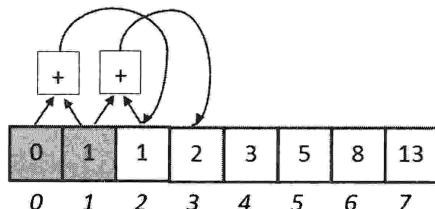
$$\begin{aligned}fib(0) &= 0, \\fib(1) &= 1, \\fib(n) &= fib(n - 1) + fib(n - 2), \text{ kde } n \geq 2\end{aligned}$$

Rekurzivní řešení jsme testovali už několikrát. Nyní se pokusíme rekurzivní proceduru výpočtu této posloupnosti přizpůsobit výše uvedeným zásadám konstrukce programu, který využívá dynamické programování:

- *Koncepcie* – rekurzivní vzor již máme, takže stačí pouze deklarovat pole $fib[n]$ k ukládání vyvypočítaných hodnot.
- *Iniciace* – prvními hodnotami v poli fib budou samozřejmě počáteční podmínky: $fib[0] = 0$ a $fib[1] = 1$.
- *Progrese algoritmu* – tento bod úzce souvisí s rekurzivním vzorem, který implementujeme pomocí pole. V našem případě se hodnota $fib[i]$ v poli (pro $i \geq 2$) rovná součtu dvou dříve vypočítaných hodnot: $fib[i-1]$ a $fib[i-2]$. Obě tyto hodnoty jsou uloženy v poli analogicky s rekurzivním programem, který je uchovává... v zásobníku rekurzivních volání.

Všimněme si však, že tuto analogii nemůžeme příliš rozvíjet, protože při dynamickém programování se využívá sekvence elementárních instrukcí bez dodatečných procedurálních volání, s nimiž pracuje každý rekurzivní program.

Uvedený postup ilustruje obrázek 9.2.



Obrázek 9.2: Výpočet hodnoty prvků Fibonacciho posloupnosti

Již zcela pro formálnost uvedeme proceduru, která realizuje výše uvedené výpočty:

```
fib-dyn.cpp
void fib_dyn(int x, int f[])
{
    f[0]=0;
    f[1]=1;
    for(int i=2; i<x; i++)
        f[i]=f[i-1]+f[i-2];
}
```

Rovnice s více proměnnými

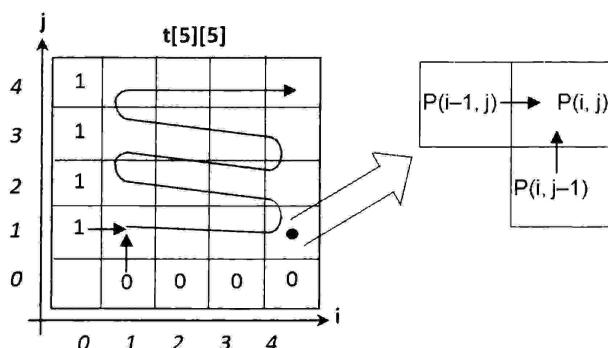
Poněkud složitější situace nastává v případě rekurzivních rovnic, které obsahují více než jednu proměnnou. Podívejme se na následující vzor:

$$P_{(i,j)} = \begin{cases} 1 & \text{pro } i = 0 \text{ a } j > 0 \\ 0 & \text{pro } i > 0 \text{ a } j = 0 \\ \frac{P(i-1,j) + P(i,j-1)}{2} & \text{pro } i > 0 \text{ a } j > 0 \end{cases}$$

Máme zde dvě proměnné: i a j . Chceme vypočítat hodnotu parametru P . Výše uvedený vzor je již na první pohled dosti nepřijemný. Kromě toho lze dokázat, že je velmi nákladný i z hlediska času výpočtů. Máme tedy dokonalý příklad toho, že bychom rekurzi raději vůbec neměli používat... samozřejmě za podmínky, že máme k dispozici jiné způsoby řešení.

Jednu cestu nám nabízí technika dynamického programování. Jestliže nás napadne použít dvourozměrné pole, jehož vodorovná a svislá souřadnice budou odpovídat proměnným i a j , bude počítání rekurzivního vzoru triviální. Podívejme se na obrázek 9.3, který představuje obecnou myšlenku programu na výpočet hodnoty $P(i, j)$.

Vzhledem k charakteru problému bude výhodné pole iniciovat počátečními hodnotami (nuly a jedničky na příslušných místech) hned na úvod, ačkoli v optimalizované verzi můžeme tuto část kódu vložit do hlavního cyklu programu. K výpočtu hodnoty $P(i, j)$ potřebujeme znát obsah dvou sousedních buněk: dolní – $P(i, j-1)$ a té na levé straně – $P(i-1, j)$. Z této poznámky můžeme odvodit, že přirozený způsob výpočtu hodnoty $P(i, j)$ využívá postup podle klikaté čáry (viz obrázek 9.3).



Obrázek 9.3: Dvourozměrný rekurzivní vzor

Na základě všech uvedených informací již samotný program dokážeme napsat obratem:

```
dyn.cpp
const int n=5;
void dynam(double P[n][n])
{
    int i,j;
    for(i=1;i<n;i++) {P[i][0]=0;P[0][i]=1;}
    for(j=1;j<n;j++)
        for(i=1;i<n;i++)
            P[i][j]=(P[i-1][j]+P[i][j-1])/2.0;
}
```

Snadno si všimneme, že uvedený program dokonale odráží rekurzivní vzor. Stačilo pouze najít *regulérní* způsob, jak vyplnit pole. Slovo regulérní je potřeba zdůraznit. V případě dvou- a více-rozměrných problémů (můžeme-li si dovolit takové označení vzhledem k rozměrům pole) se totiž můžeme velmi snadno dopustit chyby spočívající v tom, že se pokusíme využít hodnotu z pole,

které v dané fázi ještě není vypočítané. Klopýtnutí tohoto typu lze často velmi těžko odhalit, takže musíme dbát značné opatrnosti.

Nejdelší společný podřetězec

Dynamické programování umožňuje s výhodou řešit algoritmické problémy, které souvisejí s prohledáváním znakových posloupností. Pochopitelně: klasické reprezentace znakových posloupností předpokládají, že se jedná o pole vyplněná kódy ASCII (nebo libovolnými jinými kódy).

Porovnání sekvencí kódů se uplatní v mnoha oblastech. Jedná se např. o:

- porovnávání souborů a textů;
- určování „vzdálenosti“ mezi znakovými posloupnostmi (kolik musíme provést změn, abychom jednu posloupnost převedli na jinou – jedná se o tzv. Levenshteinovu vzdálenost);
- schopnost klasifikace a porovnávání se hodí např. při analýze kódů DNA (vyhledávání podobností v řetězcích genetické informace).

Pusťme se nyní do klasického problému vyhledávání nejdelšího společného podřetězce (ang. *longest common subsequence*) dvou posloupností X a Y . Algoritmus, který se pokusíme společně „vynalezť“, bychom neměli zaměňovat s klasickými algoritmy popsanými v předchozí kapitole. Sekvence, o které nám nyní jde, totiž nemusí být souvislé.

Svůj problém budeme modelovat rekurzivně. Cílem cvičení je najít určitou (zatím) neznámou funkci $LCS(X_i, Y_j)$, která vrátí nejdelší společný podřetězec posloupností X a Y . Jako příklad vezměme posloupnosti *PKOYTEK* a *MKJAIEOTI*. Jsou dosti krátké, takže hledaný výsledek dokážeme identifikovat už na první pohled: *KOT*.

Jak funkci LCS najít co nejjednodušej? Ve všech problémech rekurzivní povahy bychom měli vyhledat elementární případ, který nám naznačí další dekompozici algoritmu. V případě funkce LCS přitom budeme předpokládat, že dvě posloupnosti končí stejným znakem x_i (nebo y_i), který vzhledem k tomu tvoří jejich LCS. Když z řetězců odstraníme tento znak, musíme prozkoumat LCS zkrácených posloupností (provádíme dekompozici) a k výsledku přidáme dříve nalezený společný prvek.

Toto pravidlo lze rekurzivně zapsat následujícím způsobem:

$$LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + x_i$$

Operátor + používáme ve významu funkce, která spojuje řetězce (znakové posloupnosti).

Poněkud složitější případ nastává, když se poslední znaky obou posloupností *vzájemně liší*. Naše hledání se dělí na dvě větve:

- Odstraňujeme poslední znak z posloupnosti Y a porovnáváme s posloupností X , tzn. zkoumáme $LCS(X_i, Y_{j-1})$.
- Odstraňujeme poslední znak z posloupnosti X a porovnáváme s posloupností Y , tj. testujeme $LCS(X_{i-1}, Y_j)$.

Zajímá nás samozřejmě delší z obou získaných výsledků.

Když to shrneme, lze rekurzivní funkci LCS formulovat jako:

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & i = 0 \text{ nebo } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + x_i & x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & x_i \neq y_j \end{cases}$$

První řádek se týká situace, kdy znakovou posloupnost porovnáváme s prázdným řetězcem. Je jasné, že funkce LCS musí v tom případě vrátit rovněž prázdnou posloupnost (označenou symbolem φ). Zbývající dvě podmínky se vztahují na oba případy rekurzivní dekompozice, které jsme identifikovali výše.

Dále představíme program, který výše uvedený vzorec doslovně převádí do kódu jazyka C++. Funkce, která je součástí programu, vrací hledanou posloupnost LCS. Kvůli ukázkovému programu dostupnému ke stažení, který je propracovanější než přetištěná verze, jsou klíčové proměnné (názvy, pole) deklarovány jako *globální*.

```
lcs.cpp
const int M=7, N=9;

string X="PKOYTEK";           // M
string Y="MKJAJIEOTI";        // N

int i, j;
string S[M+1][N+1];          // pole posloupnosti LCS
string LCS_posl (string X, string Y)
{
    for (i=0; i<M; i++) S[i][0] = ""; // inicializace
    for (i=0; i<N; i++) S[0][i] = "";
    for (i=1; i<M+1; i++)
        for (j=1; j<N+1; j++)
            if (X[i-1] == Y[j-1])
                S[i][j]=S[i-1][j-1] + X[i-1];
            else
                if ( S[i][j-1].length() > S[i-1][j].length() )
                    S[i][j]=S[i][j-1];
                else
                    S[i][j]=S[i-1][j];
    return S[M][N];
}
```

V „seriozním“ programu je samozřejmě vhodné postarat se o větší parametrizaci funkcí a hermetizaci kódu.

Použití dvourozměrného pole, které uchovává podřetězce LCS proměnné délky vypočítané v jednotlivých krocích algoritmu, je v praxi časově nákladné (při operacích se znakovými řetězci je nutné pracovat s knihovními funkcemi, které jsou dosti složité), a program navíc může spotřebovat i hodně paměti. Vzhledem k témtu nevýhodám se občas uplatňuje přístup, kdy se místo posloupností LCS uchovávají jen jejich *délky*. Rekurzivní vzor zůstává téměř stejný, pouze místo řetězení znakových posloupností se výsledky funkce (délky posloupností) sčítají:

$$C(X_i, Y_j) = \begin{cases} 0 & i = 0 \text{ nebo } j = 0 \\ C(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \\ \max(C(X_i, Y_{j-1}), C(X_{i-1}, Y_j)) & x_i \neq y_j \end{cases}$$

Klíčový elementární případ, který odpovídá druhému řádku, je v kódu C++ označen komentářem (*).

KAPITOLA 9 Pokročilé programovací techniky

Vlastní řetězec LCS lze rekonstruovat zpětným rekurzivním procházením jednotlivých rozhodnutí, která algoritmus zvolil:

```
int C[M+1][N+1]; // pole délky LCS

int LCS_délka (string X, string Y)
{// C - pole délky posloupnosti LCS
    for (i=0; i<M; i++) C[i][0] = 0; // inicializace

    for (i=0; i<N; i++) C[0][i] = 0;

    for (i=1; i<M+1; i++)

        for (j=1; j<N+1; j++)
            if (X[i-1] == Y[j-1])
                C[i][j]=C[i-1][j-1] + 1; // (*)
            else
                C[i][j]=max( C[i][j-1], C[i-1][j] );
    return C[M][N];
}
// Tato funkce rekonstruuje posloupnost LCS:
string LCS_vypis (int i, int j)
{
    if ( (i==0) || (j==0) )
        return "";
    if (X[i-1] == Y[j-1])
        return (LCS_vypis(i-1, j-1) + X[i-1]);
    else
        if (C[i][j-1] > C[i-1][j])
            return LCS_vypis(i, j-1);
        else
            return LCS_vypis(i-1, j);
}
int main()
{
    cout << "LCS:" << LCS_pos1(X,Y)      << endl;
    cout << "Délka LCS:" << LCS_délka(X,Y) << endl;
    cout << "LCS:" << LCS_vypis(M,N)      << endl;
}
```

(Uvedený kód se nachází ve stejném zdrojovém souboru jako předchozí kód.)

Zdrojový soubor navíc obsahuje funkce, které umožňují vypsat obsah polí C a S. Zkuste je v rámci cvičení rozepsat na papír a pozorujte, jaké výsledky algoritmus poskytne pro různé posloupnosti.

Jiné programovací techniky

Nezasvěceným osobám informatika často připomíná magii, protože elektronická zařízení nějakým zázrakem dokáží řešit problémy, v počítačových hrách vzbuzují dojem myslících protihráčů a pomáhají s únavnými každodenními činnostmi. Ukazuje se však, že v souvislost svého oboru s kouzelnictvím někdy začínají věrit i samotní informatici. K takovým úvahám může vést studium oblasti umělé inteligence (kapitola 12) nebo právě pokročilé algoritmické techniky, zvláště ty, které se označují jako heuristické.

Co je to heuristika? V logice tento termín označuje schopnost nacházet nová fakta a souvislosti mezi nimi, díky čemuž lze získat nové vědomosti⁹. V informatice se jedná o metodu hledání přibližných, ale uspokojivých řešení, zvláště v situacích, kdy ideální řešení nemůžeme najít kvůli technickým omezením, případně nám na něm nezáleží. Heuristické vyhledávací algoritmy jsou přizpůsobené konkrétnímu nasazení: znalosti zkoumané úlohy (např. pravidel určité hry, postupů protivníka v průběhu partie, fyzikálních zákonů, způsobu šíření počítačových virů...) lze zabudovat do programu, aby dokázal postupovat k výsledku.

Pomocí heuristických metod je možné předběžně zúžit oblast možných řešení, abychom nakonec mohli uplatnit klasický algoritmus, který vypočítá věrohodný koncový výsledek. Pokud při prohledávání oboru řešení postupujeme klasickým způsobem, algoritmus se často dostává do slepých uliček a při jejím zkoumání jen plýtvá časem. Můžeme tedy počítači pomoci, aby se nezdržoval hledáním řešení na „cestách“, které nevedou k úspěchu nebo nejsou z hlediska očekávaného výsledku příliš zajímavé.

S určitým zjednodušením můžeme říci, že schéma heuristického algoritmu předpokládá existenci modelu řešeného problému, který obsahuje:

- *Stavy* – popisují aktuální konfigurace dat nebo rozhodnutí.
- *Operátory* – převádějí stavy na jiné, pokud možno ve směru hledaného řešení.

Stavy se obvykle umisťují do praktické datové struktury – např. stromu, seznamu či grafu (těmi se budeme zabývat v další kapitole), ale samozřejmě je musíme doplnit o celou algoritmickou obsluhu, která aktivně pracuje s mnoha pomocnými funkcemi. Jedná se např. o funkce na výpočet aktuálního řešení a heuristické funkce, které algoritmus řídí. Během transformací může heuristický algoritmus pomocí svých heuristických funkcí (ty je potřeba najít či vymyslet) počítat i určité dodatečné údaje, které usnadňují výběr přechodů mezi jednotlivými stavami, aby se algoritmus přiblížoval optimálnímu řešení. Heuristika může v závislosti na typu řešené úlohy nabývat konkrétní matematickou formu (funkce počítá jisté parametry kontrolující algoritmus) nebo v obecnější podobě se může jednat o určitou metodu, jak urychlit hledání řešení.

Příklad složitého algoritmu, který využívá heuristiku:

- Chceme postavit K dam na šachovnici s rozměry $N \times N$ tak, aby se dámy vzájemně neohrožovaly. Příklad na obrázku 9.4 využívá 8 dam i klasickou šachovnici. Pro jednu z dam je znázorněna oblast, kterou ohrožuje.
- „Stavem“ bude v tomto případě samozřejmě aktuální pozice na šachovnici, přičemž začínáme od prázdné šachovnice.
- Když hledáme optimální usporádání, umisťujeme figury např. do řádků následujících po sobě a přitom konstruujeme strom výběrů. V závislosti na zvolených rozhodnutích se budou v následujících fázích dostupné možnosti zužovat, protože dříve postavené dámy budou ohrožovat stále větší oblast šachovnice¹⁰.

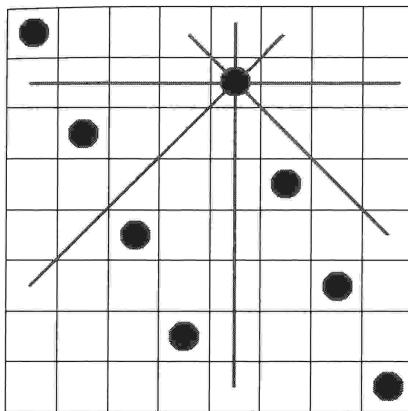
Při mechanickém postupu založeném na hrubé síle bychom generovali všechny možné konfigurace a testovali bychom, zda splňují podmínu, že se dámy nesmí vzájemně ohrožovat. Můžeme si domyslet, že řešení tohoto typu obvykle bývá příliš časově náročné, takže se nevyplatí je zkoušet (v některých úlohách je zcela nemožné). Zkusme se zamyslet nad nějakým lepším postupem.

⁹ Podle zjednodušené slovníkové definice.

¹⁰ Připomeňme pravidla hry: Dáma může útočit v každém směru (svisele, vodorovně i šikmo) a může přitom přejít libovolný počet polí.

KAPITOLA 9 Pokročilé programovací techniky

Jak provést optimální volbu? Můžeme si vzít na pomoc heuristiku neboli svou schopnost ohodnotit aktuální postavení na šachovnici. Jestliže při generování dalších pozic, kam můžeme postavit novou dámou, zároveň vypočítáme jejich „hodnotu“ pomocí jisté (záhadné) heuristické funkce f , můžeme výsledky setřídit a vybrat třeba takový, který je z hlediska použité heuristiky optimální.



Obrázek 9.4: Problém 8 dam

Jak definovat funkci f ? Musíme se nad tím samozřejmě dobře zamyslet, protože na kvalitě této funkce závisí výsledky činnosti algoritmu. V tomto případě můžeme jako kritérium přijmout například počet polí v následujících řádcích, které nejsou ohroženy útokem, abychom příliš neomezili možnosti v dalších krocích.

Nedostatkem heuristických postupů je jejich praktický aspekt: místo jasného matematického modelu (který lze převést na čitelný kód) založíme algoritmus na určitých pravidlech, která vycházejí z intuice a zkušenosti. Kvalitu heuristiky pak prakticky zkoumáme tak, že analyzujeme... výsledky činnosti algoritmů.

Klasickými heuristickými metodami samozřejmě řada možných programovacích technik nekončí. Stojí za to, abychom se seznámili i s jinými zajímavými metodami hledání výsledků, které narušují zavedené zvyklosti a konvence:

- *Losujeme* množinu možných stavů (voleb) z prostoru přípustných řešení a počítáme pro ně dříve definovanou cílovou funkci. Vybiráme nejlepší řešení a... označíme je za konečné řešení problému.
- Využijeme algoritmy zvané *genetické*, které způsobem svého fungování napodobují produkty biologické evoluce. Genetické (nebo obecněji evoluční) algoritmy tvoří bohatou oblast algoritmiky, do které se kvůli nedostatku místa nebudeme pouštět. Metoda genetických algoritmů často u nepoučených osob vzbuzuje pochybnosti, což často vyplývá z trochu zavádějící biologické analogie. Ve skutečnosti se jedná o určitý způsob modelování řešení optimalizačních úloh, kde parametry problému kódujeme v datové struktuře označované jako chromozóm, což může být... binární posloupnost. Opět se tedy dostáváme na pevnou půdu informatiky.

Čtenář se může oprávněně zeptat: Proč se uvedeným tématem nezabýváme podrobněji, ačkoliv vypadá velmi zajímavě? Důvodem je samozřejmě značná šíře celé problematiky. O popsaných (nebo přesněji řečeno zmíněných) technikách vyšly obsáhlé specializované publikace. Kdybychom se stejně optimalizační problémy pokusili rozebrat v této příručce, její rozsah by neúnosně vzrostl. Je ale dobré vědět o tom, že takové programovací techniky existují. Studenti informatiky se s nimi v průběhu výuky nepochybňě setkají.

Bibliografické poznámky

V této kapitole jsme měli možnost seznámit se s několika jednoduchými programovacími technikami, jejichž efektivní použití může programátorem při algoritmickém řešení problémů značně pomoci. Samozřejmě nejde výhradně o metaalgoritmy, se kterými bychom se mohli setkat v odborné literatuře. Volba padla na techniky, jejichž princip není příliš komplikovaný a které nevyžadují náročnější informatické studium.

Pokud se o programovací techniky zajímáte hlouběji, doporučuji sáhnout po knize [HS78]. Je psaná velmi prostým jazykem (jistě nevadí, že anglickým) a obsahuje velmi podrobné popisy mnoha různorodých programovacích strategií a technik. Osoby, které by při řešení algoritmických problémů chtěly využívat stromové struktury, si mohou spolu s touto knihou přečíst také publikaci [Nil82]. Pokud se vám původní Nilssonova práci nepodaří sehnat, naleznete hodně cenných informací i v polském titulu [BC89]. Nakonec mohu doporučit francouzskou práci [CP84], která však může být dosti obtížně dostupná – jedná se o skriptum, takže jej spíše než na francouzském knižním trhu naleznete v tamních univerzitních knihovnách.

KAPITOLA 10

Prvky algoritmiky grafů

V této kapitole:

- Základní definice a pojmy
- Cykly v grafech
- Způsoby reprezentace grafů
- Základní operace s grafy
- Roy-Warshallův algoritmus
- Floyd-Warshallův algoritmus
- Dijkstrův algoritmus
- Bellman-Fordův algoritmus
- Minimální rozpínavý strom
- Prohledávání grafů
- Problém vhodného výběru
- Shrnutí
- Úlohy

Grafy nejsou nic jiného než *datová struktura*, a některé čtenáře proto může překvapit, že jim věnujeme samostatnou kapitolu. Je to však způsobeno značným významem grafů v algoritmice. Nijak nepřeháníme, když prohlásíme, že bez této datové struktury by se mnohé problémy vůbec nedaly vyřešit.

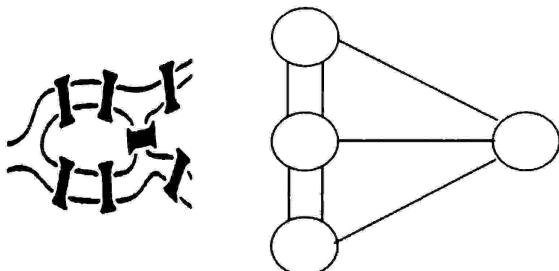
Grafy mají dosti složité teoretické základy (jejich studium představuje samostatnou matematickou disciplínu), ale v našem výkladu se pokusíme přílišnému formalizmu vyhýbat.

Výhradně kvůli přesnému vymezení rozebíraných problémů představíme některé teoretické principy, ale omezíme se přitom jen na nezbytné minimum.

Čtenářům, kteří se o teorii grafů více zajímají, lze v zásadě doporučit libovolnou příručku algoritmiky, protože oborová literatura obvykle této datové struktuře věnuje dostatek prostoru. Zajímavý přístup založený na kombinaci matematiky a informatiky nabízí [Hel86]. Bohužel nevím, zda je tento titul již dostupný v knižním vydání, nebo zůstal pouze ve formě školního skripta. Z dostupnějších publikací doporučuji [CLR02] nebo [Sed02].

V této kapitole se pokusíme představit základní informace (jedná se totiž o mimořádně rozsáhlou tematiku) týkající se grafů a způsobů jejich reprezentace v programech. Seznámíme se s potřebnou terminologií, která se týká této datové struktury, a ukážeme rovněž několik typických algoritmů, které s ní pracují.

Z historické perspektivy se grafy objevily roku 1736 díky švýcarskému matematikovi L. Eulerovi. Díky nim dokázal vyřešit problém, se kterým přišli obyvatelé města Königsberg (česky Královec), a to: jak postupně přejít všech sedm mostů, které v tomto městě stojí, ale přitom žádný most nepřekročit dvakrát. Obrázek 10.1 znázorňuje část historické mapy s centrálním ostrovem Kneiphof na řece Pregel (Pregola), která se za ostrovem rozděluje do dvou ramen. Vpravo od plánu je znázorněna smluvní grafová reprezentace, která předpokládá, že každý most je reprezentován hranou grafu a uzly grafu pak představují fragmenty souše (ostrovy). Při té příležitosti se můžeme seznámit s typickou konvencí, která se vztahuje na vrcholy a hrany: vrcholy souvisejí s jistými objekty (např. města, území) a hrany zase s relacemi, které mezi objekty existují (např. cesty, mosty, letecká spojení)...



Obrázek 10.1: Grafový model problému mostů v Královci

Euler ve své práci dokázal, že úloha nemá řešení. Rozhoduje o tom konfigurace mostů a vodních ploch (podrobnější informace na tomto tématu se nachází v části „Cykly v grafech“ na straně 231).

Tento historický příběh zároveň dokonale ukazuje, k čemu mohou být grafy v praxi užitečné: dovolují ideálně modelovat všechny algoritmické úlohy, které spočívají v hledání (optimálních) cest. Tím uplatnění grafů samozřejmě nekončí – skvěle se hodí i k řešení mnoha jiných problémů, k nimž patří:

- analýza bezpečnosti sítě (zajímá nás, zda poškození jednoho z komunikačních uzlů ovlivní možnost předávat informace mezi body A a B),
- analýza elektronických obvodů (např. zda v dané konfiguraci nedojde ke zkratu),
- vyhledávání originálních a převzatých informací v hypertextových dokumentech (tj. v dokumentech s hypertextovými odkazy),
- problém optimálního výběru (např. středoškoláci podávají přihlášky na několik univerzit na jednou, aby zvýšili svou šanci na přijetí, a vysoké školy zase omezují počet dostupných míst v jednotlivých oborech a uplatňují vlastní kritéria výběru nejlepších zájemců).
- kompilátory (a programy na spojování kódu, tzv. linkery) pomocí grafů vytvářejí sítě závislosti funkcí a modulů.

Programátor, který se dobře seznámí s možnostmi nasazení grafů a porozumí jím, v praxi zdvojnásobí své schopnosti účinného *modelování* řešených problémů. Mnoho algoritmických úloh se vyznačuje tou zvláštní vlastností, že vytvoříme-li dobrý celkový model, dokážeme je vyřešit téměř mimochodem. Právě grafy, které rozšiřují stromové struktury, se ideálně hodí pro mnoho problémů abstraktní i praktické povahy.

V následujících odstavcích se budeme věnovat základní terminologii grafů a poté přejdeme ke způsobům reprezentace grafů v počítačových programech.

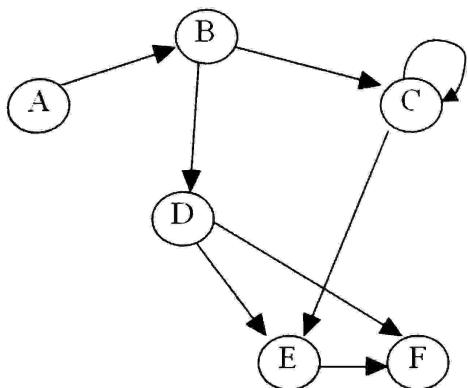
Základní definice a pojmy

Nepříliš rozsáhlé grafy můžeme ideálně předvést ve formě názorných schémat, jaké vidíme např. na obrázku 10.2.

Grafem G nazýváme dvojici (X, Γ) , kde X označuje množinu tzv. *uzlů* (nebo *vrcholů*) a $\Gamma(x, y) \in X^2$ představuje množinu *hran*.

Graf je *rovinný* (též *planární*), pokud jej můžeme nakreslit v ploše takovým způsobem, aby se jeho hrany neprotínaly. Otázka: Je graf z obrázku 10.2 rovinný? Schematická znázornění jednoduchých grafov v algoritmických knihách mohou vést k malému nedorozumění, které spočívá v tom, že čtenáři obrázkům grafů přikládají přílišný význam. V praxi jsou však grafy se svými předlohami

vizuálně podobné jen v případě, kdy modelují fyzické struktury (např. mapy či elektronické obvody). Podle potřeby totiž můžeme vzdálenosti uzelů i délky hran volit tak, aby datové struktura – která je z principu abstraktní – nějak odrážela svůj vzor z reálného světa.



Obrázek 10.2: Příklad grafu

Chromatické číslo grafu je nejmenší počet barev, kterými lze obarvit vrcholy grafu takovým způsobem, aby každé dva vrcholy spojené hranou měly různé barvy.

Graf je *orientovaný* (ang. *directed graph*, občas též označovaný jako *digraph*¹), pokud je hranám připsán nějaký směr (na obrázcích se symbolizuje šipkou). Orientovaný graf se občas definuje se znamenem hran $X \rightarrow Y$. Šipka v této notaci odpovídá šipce na hraně mezi příslušnými vrcholky grafu. Budeme-li uvažovat dva uzly grafu X a Y spojené hranou, pak X je *počáteční uzel* a Y *příslušný koncový uzel*.

Vstupní stupeň vrcholu grafu je počet hran, které do vrcholu vstupují. Analogicky *výstupní stupeň* určuje počet hran, které z daného vrcholu vystupují.

Jako *regulární* nazýváme graf, jehož všechny vrcholy mají stejný stupeň.

Graf na obrázku 10.2 zahrnuje 6 uzelů: A, B, C, D, E a F , přičemž některé z nich jsou spojeny hranami: $(A, B), (B, C), (B, D), (D, F), (D, E), (E, F)$ a (E, C) . Uzel C má specifický charakter, protože z něj vychází hrana, která... se následně vrací do svého počátečního uzlu! V některých algoritmických úlohách budeme potřebovat i takové podivné hranы, protože pomocí nich dokážeme modelovat více situací než s použitím běžných uzelů a hran.

Posloupnost vrcholů, kde každé dva následné vrcholy jsou spojeny hranou, se označuje jako *sled grafu*. Tah v grafu je sled, ve kterém se neopakují hranы, a cesta je sled, ve kterém se neopakují vrcholy.

Graf, ve kterém existuje cesta mezi každými dvěma vrcholy, nazýváme *souvislý*.

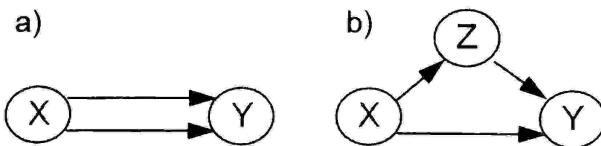
Podgraf grafu je graf, jehož vrcholy tvoří podmnožinu množiny vrcholů grafu G a hraný podmnožinu množiny hran grafu G . V algoritmických popisech se podgrafený někdy označují jako *redukované grafy*.

Čísla uzelů (nebo také symbolicky: písmenové etikety) slouží v zásadě jen k rozlišení uzelů, kterým přitom nepřipisují nějaké pevné pořadí. Programátor však v případě potřeby může číslům uzelů přiřadit nějaký dodatečný význam (např. v teorii her to může být záznam s popisem stavu hry).

¹ Tento termín uvádíme jen pro úplnost, protože se bez tohoto kalku z angličtiny snadno obejdeme.

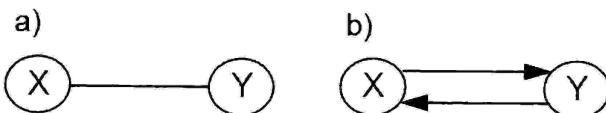
KAPITOLA 10 Prvky algoritmiky grafů

Podle původní definice může mezi dvěma uzly existovat pouze jedna hrana, ale od grafu, který této definici neodpovídá (viz obrázek 10.3a), můžeme velmi snadno přejít ke standardnímu grafu tak, že přidáme dodatečné vrcholy (obrázek 10.3b).



Obrázek 10.3: „Normalizování“ grafu (1)

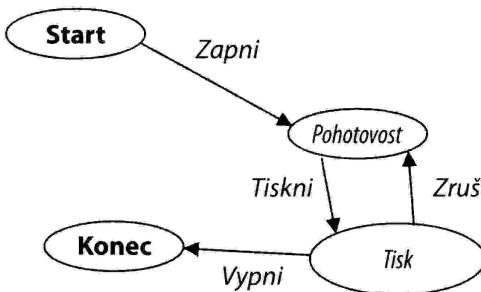
Orientovaný graf je z principu nejobecnější, protože graf *neorientovaný* (viz obrázek 10.4a) lze velmi snadno převést na *orientovaný* (obrázek 10.4b).



Obrázek 10.4: „Normalizování“ grafu (2)

V určitých aplikacích je potřeba hranám grafu připsat konkrétní hodnoty (nejčastěji číselné, ale mohou to být rovněž etikety jiného typu). Tím se zároveň změní definice grafu, protože místo dvojice (X, Γ) dostáváme (X, Γ, V) . Třetí parametr V označuje množinu hodnot odpovídající daným hranám. Může se jednat o nějakou smluvní etiketu, „váhu“, náklady, řídicí příkaz...

Obrázek 10.5 znázorňuje, jak snadno lze tzv. konečné automaty modelovat pomocí grafů. Aniž bychom zabíhali do teorie automatů, můžeme přijmout konvenci, že „stav“ (uzel grafu) reprezentuje určitou informaci týkající se systému (zařízení, mechanizmu). Přechody mezi stavůmi jsou reakcemi na určité události (viz popis hrany)².



Obrázek 10.5: Graf přechodů stroje konečných stavů

V teorii grafů se můžeme setkat ještě s mnoha dalšími definicemi a pojmy, ale dohodněme se, že budeme-li potřebovat nové termíny, seznámíme se s nimi postupně v dalším výkladu.

V následující části kapitoly přejdeme k popisu několika typických metod, které umožňují reprezentovat grafy v paměti počítače. Díky tomu dokážeme snáze prezentovat samotné grafové algoritmy. Výklad můžeme začít popisem zajímavého jevu, který se v grafech vyskytuje – *cyklů*.

² Osoby, které se vážněji zajímají o informatiku či elektroniku, se při studiu nepochyběně setkají s teorií automatů a pojmem konečného automatu (Mooreova stroje).

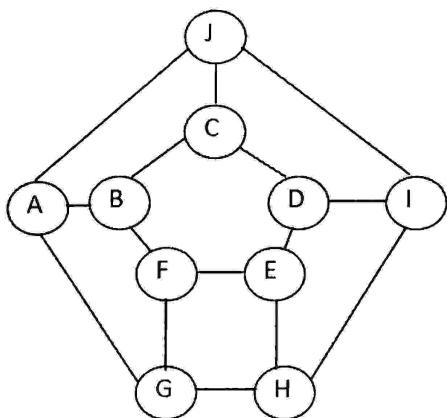
Cykly v grafech

Cesta v grafu, která končí ve stejném vrcholku, kde začala, se nazývá *cyklus* (kružnice).

Jako *orientovaný cyklus* (který se z definice vyskytuje v orientovaném grafu) označujeme cyklus, ve kterém jsou všechny dvojice sousedních vrcholů uspořádány podle směru hran.

Orientovaný graf, který neobsahuje ani jeden orientovaný cyklus, se nazývá *acyklický orientovaný graf* (ang. *directed acyclic graph*, občas označovaný jako DAG³).

Jako hamiltonovský cyklus označujeme cestu, která prochází všemi vrcholy a vrací se do počátečního vrcholu⁴ (viz obrázek 10.6, kde můžeme abecedně projít cestu od uzlu A k uzlu J).



Obrázek 10.6: Hamiltonovský cyklus

Předpokládejme nyní, že náš graf pomocí uzlů reprezentuje města a hrany jsou označeny svou „váhou“, která může označovat např. vzdálenosti nebo dopravní náklady daného úseku cesty (viz tabulku 10.1).

Tabulka 10.1: Problém obchodního cestujícího

	Brno	Ostrava	Plzeň	Praha
Brno	0	165	296	202
Ostrava	165	0	456	362
Plzeň	296	456	0	94
Praha	202	362	94	0

Pokud v takovém grafu najdeme hamiltonovský cyklus s minimální sumou vah jeho hran, vyřešíme tím slavný *problém obchodního cestujícího* (ang. *traveling salesman problem*), který chce začít cestu v jednom městě, navštívit s co nejmenšími náklady všechna zbyvající města a vrátit se do výchozího bodu. Problém obchodního cestujícího patří do třídy NP-úplných problémů⁵.

³ Další podivný jazykový kalk, který uvádíme jen z formálních důvodů.

⁴ Sir William Rowan Hamilton byl známý irský matematik, který žil v letech 1805–1865. Hamiltonovy a Eulerový práce vzbudily zájem o teorii grafů a široké možnosti jejího uplatnění.

⁵ V této knize nebudeme otevírat téma Turingova stroje, bez kterého nelze přesně definovat pojmem třídy NP-úplných algoritmů. Pro jednoduchost berme na vědomí, že optimalizační problém je NP-úplný. To znamená, že při rostoucím objemu dat (např. zvyšování počtu uzlů grafu) nedokážeme najít algoritmy, které by poskytovaly přesné řešení problému v polynomickém čase, který by závisel na velikosti dat.

KAPITOLA 10 Prvky algoritmiky grafů

V praxi se při řešení problému obchodního cestujícího uplatňují genetické algoritmy, které poskytují dobré, i když přibližné výsledky.

Hamiltonovský cyklus je potřeba rozlišovat od *eulerovského tahu*, který se týká hran – v tom případě můžeme stejný vrchol navštívit vícekrát. Na základě Eulerovy práce zmíněné na začátku kapitoly se dodnes používá pojem *eulerovského grafu*, tedy takového grafu, v němž existuje uzavřený eulerovský tah – cesta, která umožňuje jedenkrát přejít všechny hrany grafu.

Jiným známým algoritmickým problémem z kategorie vyhledávání optimálních cest je problém čínského listonoše, kdy hledáme optimální cestu za předpokladu, že musíme každou hranu grafu (ulici) navštívit alespoň jednou. Název problému pochází z jeho prvního popisu, který byl roku 1962 publikován v čínském matematickém časopise. Samotnou úlohu pak pomocí zajímavého názvu popularizoval Alan Goldman.

Vraťme se ještě k eulerovskému tahu.

Stojí za to zapamatovat si jeho následující vlastnost, která vyplývá z Eulerových prací:

Neorientovaný graf obsahuje eulerovský tah, pokud je souvislý a má vrcholy sudého stupně. Orientovaný graf, který obsahuje eulerovský tah, se musí kromě souvislosti vyznačovat i stejným počtem vstupních a výstupních hran v každém vrcholu.

Eulerovo tvrzení dovoluje zkontoľovat, zda v grafu eulerovský tah existuje, ale při jeho hledání nám bohužel nepomůže. Naštěstí máme k dispozici jednoduchý a elegantní algoritmus, jehož autorem je Henry Fleury⁶ a který umožňuje příslušný tah najít velmi intuitivním způsobem:

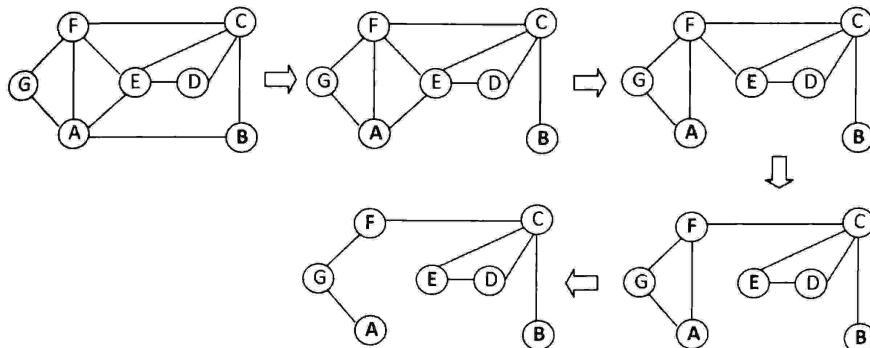
1. Vybereme libovolný vrchol.
2. Vybereme jednu z výstupních hran vrcholu, kde se aktuálně nacházíme. Volíme hranu, po které jsme zatím nešli, a navíc se snažíme, aby vybraná hrana nebyla dělící hranou grafu (s výjimkou situace, kdy to jinak nejde).
3. Zapamatujeme vybranou hranu a odstraníme ji z grafu.
4. Po vybrané hraně přejdeme na následující vrchol.
5. Opakujeme kroky 2, 3 a 4 až do chvíle, kdy přejdeme všechny hrany a vrátíme se do počátečního vrcholu.

V každém kroku Fleuryho algoritmu vzniká redukovaný graf, tzn. výchozí graf bez odstraněných vrcholů. Zbývající vrcholy musí samozřejmě tvořit jeden souvislý graf. Jinak řečeno nesmíme graf rozdělit na dvě části. Když dodržíme pravidla algoritmu a podaří se nám přitom vrátit se do vrcholu, z nějž jsme vyšli, a navštívit přitom všechny hrany, nalezli jsme eulerovský tah.

Několik prvních kroků Fleuryho algoritmu je znázorněno na obrázku 10.7.

Podle instrukcí vybíráme libovolný vrchol, řekněme *B*. Ze dvou jeho sousedů, které můžeme vybrat (*A* a *C*) se náhodně pustíme ve směru *A* a přitom „mažeme“ hranu *B-A*. Podobným způsobem postupujeme např. od *A* k *E* a poté od *E* k *F* (pravá dolní část obrázku). V tomto okamžiku máme u uzlu *F* na výběr: *C*, *A* a *G*. Můžeme zvolit uzel *C*? Ne, protože v tom případě by se redukovaný graf rozpadl na dvě samostatné části. Vybíráme tedy např. uzly *A*, *G*, *F* atd., dokud se nedostaneme na konec neboli k uzlu *B*.

⁶ Tento algoritmus byl popsán v roce 1883. Zajímavá analýza historických zdrojů, které se týkají publikace i samotného autora, je k dispozici na fóru <http://mathforum.org>.

**Obrázek 10.7:** Příklad Fleuryho algoritmu

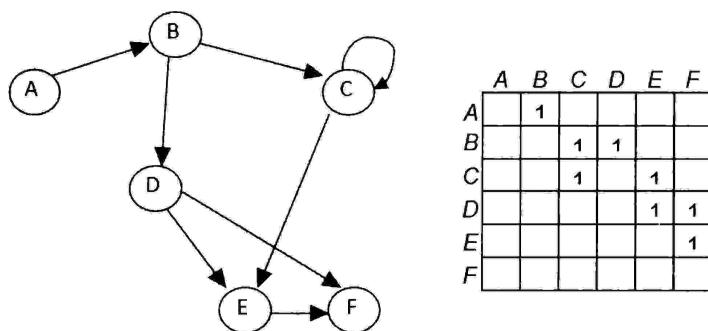
Fleuryho algoritmus dokonale ilustruje typický způsob, jakým lze popisovat grafové algoritmy. Chceme-li jej implementovat pomocí počítače, musíme však disponovat několika nástroji, které zatím postrádáme. Mimo jiné se musíme naučit modelovat grafy jako datové struktury a seznámit se s trohou teorie. V další části kapitoly tyto nedostatky napravíme. Nejdříve předvedeme, jakými způsoby lze grafy reprezentovat v paměti počítače.

Způsoby reprezentace grafů

K reprezentování grafů se dobře hodí datové struktury, které již známe, např. pole, seznamy a stromy. V praxi se však grafy obvykle realizují jedním ze dvou způsobů: pomocí dvourozměrného pole (tzv. matice sousedství) a tzv. *slovníku uzlů* (tzv. seznam sousedství). Občas se využívá i řešení, které je založeno na zapsání seznamu hran formou *množiny*.

Reprezentace pomocí pole

Své úvahy začneme od nejjednoduššího případu, tedy od *dvourozměrného pole*. Když se dohodneme, že řádky budou označovat počáteční uzly hran grafy a sloupce jejich koncové uzly, můžeme graf z obrázku 10.2 převést na pole z obrázku 10.8.

**Obrázek 10.8:** Reprezentace orientovaného grafu pomocí pole

Jednička na pozici (x, y) označuje, že mezi uzly x i y existuje hrana, která je orientována směrem k y . Ve všech ostatních případech bude pole obsahovat například nulu. Hodnoty 0 či 1 lze samozřejmě nahradit logickými hodnotami (*true a false*).

KAPITOLA 10 Prvky algoritmiky grafů

Díky reprezentaci založené na nulách a jedničkách lze jednoduše zapísovat orientované grafy pomocí běžných dvourozměrných polí. Změníme-li typ pole z číselného např. na znakový, můžeme v buňkách pole ukládat také názvy etiket nebo jiné hodnoty. Ve svých algoritmech samozřejmě musíme zajistit, aby obsah takových polí správně interpretovaly. Chybějící hodnota může například označovat, že neexistuje příslušná hrana, a znak „q“ pak může informovat o tom, že hrana existuje, a zároveň sloužit jako její etiketa.

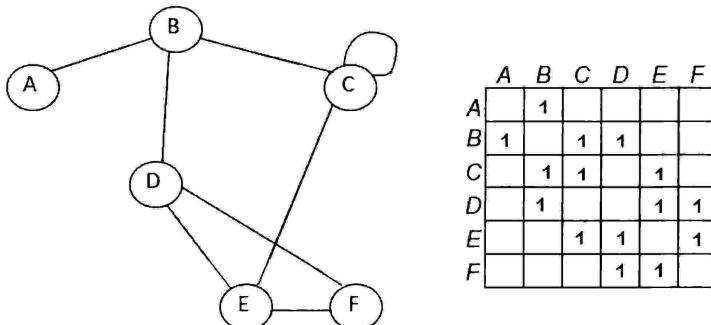
Všimněme si, že reprezentace pomocí pole má jednu podstatnou výhodu: velmi snadno se implementuje prakticky v libovolném programovacím jazyce a navíc se jednoduše používá. Vada: Formou polí je možné reprezentovat pouze grafy, které mají předem určený počet uzlů.

Chceme-li implementovat grafy pomocí polí v jazyce C++, musíme do kódu uvést deklaraci následujícího typu:

```
const int n=5;  
int G[n][n];
```

Když přijmeme konvenci, že indexy $(x, y) \in (0, 1, 2, 3 \text{ a } 4)$ označují uzly grafu A, B, C, D a E, můžeme zapsáním jisté hodnoty do buňky $G[x][y]$ zakódovat existenci hrany x-y, která spojuje dva vrcholy. Hodnota v poli může znamenat délku úseku či jeho náklady – záleží pouze na programové interpretaci.

Jak můžeme pomocí pole implementovat neorientovaný graf? Můžeme postupovat tak, že dublujeme zápis y týkající se dané hrany (např. A-B a B-A). Na příklad se můžeme podívat na obrázku 10.9.



Obrázek 10.9: Reprezentace orientovaného grafu pomocí pole

Když pevně deklarujeme maximální velikost pole, spotřebujeme bohužel hodně paměti, což může při spuštění příslušného programu způsobit potíže.



Upozornění: Pokus o spuštění programu, který pracuje s velmi rozsáhlým dynamickým polem, může skončit přeplněním zásobníku (dobře známou zprávou *Stack overflow!*).

I v případech, kdy velikost pole nepředstavuje problém, zůstává nevýhoda málo srozumitelného přidávání a odstraňování uzlů (s hranami samozřejmě žádné potíže nejsou). Když investujeme určité programátorské úsilí, jistě můžeme v jazyce C++ vytvořit třídu, která bude účinně simuloval graf s proměnným počtem uzlů a bude nadále založena na poli. V praxi se však bude jednat o těžká a neefektivní řešení. Navíc pokud graf obsahuje malý počet hran, bude takové pole využito velmi málo a většina jeho buněk zůstane prázdná.

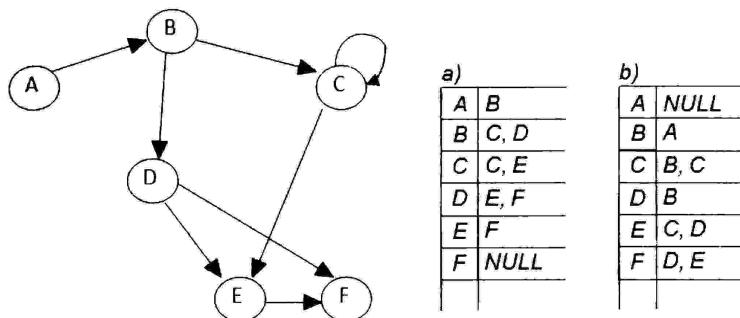


Poznámka: Mimořádnou výhodou reprezentace pomocí pole je bezprostřední a levný (z hlediska algoritrické složitosti) přístup k informacím. Chceme-li zkontrolovat, zda jsou dva uzly propojeny hranou, stačí přečíst příslušnou buňku pole (čas přístupu je přitom stálý a nezávisí na poloze uzlu ani rozměru grafu).

Slovníky uzlů

Pokud potřebujeme implementovat graf, u kterého se během činnosti programu může měnit počet uzlů, je vhodnější použít tzv. *slovník uzlů* (neboli seznam sousedství, ačkoli tento název je poněkud matoucí).

Slovník uzlů se může týkat dvou typů uzlů: *následníků* (výstupních uzlů) nebo *předchůdců* (vstupních uzlů) daného uzlu. Koncepci znázorňuje obrázek 10.10.



Obrázek 10.10: Reprezentace grafů pomocí slovníku uzlů

Pojem „slovník“ se používá vzhledem k charakteru vyhledávání: *klíčem* je uzel, který odkazuje na *data* neboli přidružené přiléhající uzly (stejně pravidlo se vztahuje jak na orientované, tak na běžné grafy).

Slovník může být běžný seznam (v nejhorším případě pole) ukazatelů na seznam uzlů (možná to vypadá komplikovaně, ale opravdu se není čeho bát). Seznam přitom může zahrnovat *výstupní* (obrázek 10.10a) i *vstupní* (obrázek 10.10b) uzly daného uzlu z hlediska hran. Některé grafové algoritmy potřebují právě informace tohoto typu a v těchto případech je uvedená reprezentace výhodná. Vezmeme-li v úvahu, že slovník uzlů lze snadno realizovat pomocí seznamu seznamů, automaticky odpadá problém, s nímž jsme se setkali v reprezentaci pomocí pole – graf může mít v zásadě neomezený počet uzlů.



Upozornění: Nevýhoda reprezentace založené na seznamech je *nepřímý* přístup k informaci o tom, zda jsou dva uzly spojené hranou: musíme prohledat seznam (množinu) přiléhajících uzlů. Příslušná operace má proto lineární složitost.

Seznamy kontra množiny

Zajímavá varianta implementace nabízí řešení, které je založeno na zapsání seznamu hran pomocí *množiny*. Za předpokladu, že prvky množiny budou dvojice hran (např. orientované), lze graf z obrázku 10.10 zapsat jako množinu dvojic hran:

$\{A, B\}, \{B, C\}, \{B, D\}, \{C, C\}, \{C, E\}, \{D, E\}, \{D, F\}, \{E, F\}, \{F, \text{NULL}\}$.

KAPITOLA 10 Prvky algoritmiky grafů

Zbývá nám samozřejmě implementovat základní operace v datové struktuře, která spočívá na této množině:

- přidávání a odstraňování dvojic vrcholů,
- prohledávání množiny (iterátory a funkce, které vyhledávají dvojice splňující určená kritéria).

Vzhledem k tomu, že s poli, seznamy a množinami jsme se již seznámili (v kapitole 5), nepochybnejme vytvořit i příslušnou datovou strukturu (např. třídu *Graf*) v jazyce C++. V této knize se popisem takové třídy nebudeme zabývat, protože bychom tím jen zhoršili srozumitelnost dále prezentovaných algoritmů. Kdybychom zvolili důsledně objektový přístup, museli bychom uvažovat o několika třídách, které by implementovaly uzly, seznamy ukazatelů na seznamy uzel a seznamy výstupních (vstupních) uzel. Jak jsme již uvedli, místo seznamu uzel je možné použít množinu uzel. Záleží na tom, zda potřebujeme nějakým způsobem uspořádat uzly, které přiléhají k danému vrcholu.

Díky velkému počtu možných implementací mají programátoři značnou volnost, aby datovou strukturu přizpůsobili konkrétnímu řešenému problému. Známe-li maximální počet uzel a model založený na poli je přijatelný z hlediska obsazené paměti, není důvod váhat – v tomto případě rozhoduje argument jednoduchého použití.

Základní operace s grafy

Mnoho grafových algoritmů lze snadno vyjádřit pomocí speciální matematické notace, kterou představíme v této části kapitoly.

Jsou-li dány dva grafy: $G_1 = (X, \Gamma_1)$ a $G_2 = (X, \Gamma_2)$, můžeme na nich definovat následující operace:

Součet grafů

$$G_3 = G_1 + G_2 = (X, \Gamma_1 + \Gamma_2).$$

Výsledný graf G_3 obsahuje všechny hrany grafů G_1 a G_2 .

Kompozice grafů

$$G_3 = G_1 \circ G_2 = (X, \Gamma_3) = \{(x, y) \mid \exists z \in X : (x, z) \in \Gamma_1 \text{ a } (z, y) \in \Gamma_2\}$$

Hrany (x, y) výsledného grafu G_3 splňují podmínku, že existuje jistý uzel z takový, že (x, z) patří do G_1 a (z, y) patří do G_2 .

Kompozici grafů lze programově realizovat poměrně snadno, např. takovým způsobem, jaký používá následující výpis (pro jednoduchost se omezíme jen na reprezentaci pomocí pole):

```
kompoz.cpp
void kompozice(int g1[n][n], int g2[n][n], int g3[n][n])
{
    int z;
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
    {
        z=0;
        while(1)      // nekonečný cyklus
    {
```

```

if(z==n)
    break; // opuštění cyklu
if( (g1[x][z]==1) && (g2[z][y]==1) )
    g3[x][y]=1;
z++;
}
}
}

```

(Plná verze programu obsahuje propracovaný příklad ve funkci `main`.)

Mocnina grafu

Mocnina G^p se definuje rekurzivním způsobem:

$$\begin{aligned} G^0 &= D, \\ \forall p \geq 1, G^p &= G^{p-1} \circ G = G \circ G^{p-1}. \end{aligned}$$

Symbol D označuje tzv. *diagonální graf* neboli takový graf, ve kterém existují výhradně hrany typu (x, x) . S mocninou grafu souvisí poměrně zajímavé tvrzení: (x, y) patří do G^p tehdy a právě tehdy, jestliže v G existuje cesta délky p , která vede od uzlu x k uzlu y .

Graf představuje dosti zajímavou matematickou koncepci, protože již svou vlastní strukturou umožňuje zcela přirozeně vyjadřovat *binární relace*, které jsou definovány na množině jeho vrcholů X . Jako elementární příklad můžeme uvést pojem *symetrie*: pokud existence hrany (x, y) implikuje existenci hrany (y, x) , můžeme o grafu prohlásit, že je *symetrický*. Podobným způsobem lze definovat hodně jiných binárních relací, jejichž většina... nemá žádný praktický význam. Výjimku představuje relace tranzitivity, která označuje, že každá cesta grafu G s délkou větší nebo rovnou 1 je „podepřena“ nějakou hranou.

Proč je relace tranzitivity tak důležitá? Tranzitivita totiž – paradoxně – nic neznamená. Jedná se pouze o výhodný prostředek, jak definovat tzv. *tranzitivní uzávěr grafu*, který se typicky označuje jako $G^+ = (X, \Gamma^+)$, kde:

$$G^+ = \{(x, y) \mid \text{existuje cesta od } x \text{ do } y \text{ v grafu } G\}.$$

Dokážeme-li provést tranzitivní uzávěr grafu, umíme odpovědět na důležitou otázku, zda lze po hránach grafu přejít z jednoho vrcholu do druhého. Všimněme si, že tranzitivní uzávěr neposkytuje návod, jak přejít od daného vrcholu k vrcholu y : pouze se dozvímme, zda je to možné.

Tranzitivní uzávěr grafu lze kromě jiných způsobů vypočítat i následovně:

$$G^+ = G \cup G^2 + \dots + G^n.$$

(n označuje počet vrcholů grafu neboli poněkud formálněji $n = |X|$).

Výhodou uvedeného algoritmu je jednoduchý zápis. Jak si snadno domyslíme, jeho nevýhoda spočívá v tom, že se složitě realizuje a výsledný kód je poměrně náročný.

Pokud má někdo hodně volného času, jistě snadno vymyslí alespoň jeden algoritmus, který zajistí tranzitivní uzávěr podle výše uvedeného návodu. Měli bychom však předem upozornit, že toto úsilí nebude mít přílišný smysl. Existuje totiž jiný algoritmus, který překonává všechny varianty algoritmů založených na mocninách grafů. Jedná se o slavný *Roy-Warshallův algoritmus*, kterému se budeme věnovat v další části kapitoly.

Roy-Warshallův algoritmus

Algoritmus popsaný v této části se vyznačuje několika vlastnostmi, díky nimž při výpočtu tranzitivního uzávěru grafu v zásadě nemá konkurenci. Především nepoužívá žádné pomocné grafy (v případě mocninných algoritmů se tomu nedá vyhnout). Navíc umožňuje poměrně snadno zjistit cestu, která vede po hranách od jednoho vrcholu k druhému.

Algoritmus je založen na operaci Θ , která je pro graf $G = (X, \Gamma)$ definována následujícím způsobem:

$$\Theta_k = \Gamma \cup (y, z) \mid (x, k) \in \Gamma \text{ a zároveň } (k, y) \in \Gamma.$$

Uvedený zápis znamená, že pro daný vrchol k do množiny hran Γ přidáváme hrany, které spojují předchůdce a následníky tohoto vrcholu.

Lze dokázat, že tranzitivní uzávěr grafu můžeme vypočítat pomocí následujících kompozic operací Θ , tzn. pro graf s vrcholy $1, \dots, n$:

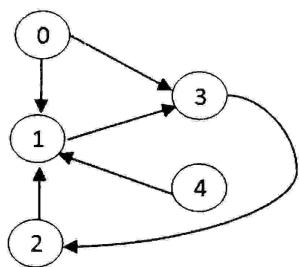
$$\Gamma^+ = \Theta_n (\Theta_{n-1} \dots (\Theta_1) \dots).$$

Algoritmus je možné v jazyce C++ zapsat velmi jednoduše:

warshall.cpp

```
void warshall(int g[n][n])
{
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
            for(int z=0; z<n; z++)
                if(g[y][z]==0)
                    g[y][z]=g[y][x]*g[x][z];
}
```

Abychom tomuto programu dobře porozuměli, budeme sledovat, jak zpracuje jednoduchý graf s pěti uzly, který je znázorněn na obrázku 10.11.



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

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

Výsledek provedení Roy-Warshallova algoritmu

Obrázek 10.11: Příklad provedení Roy-Warshallova algoritmu

(Místo obvyklých jedniček se na obrázku používají znaky x.)

Z pole znázorněného na obrázku 10.11 lze výčist mj. následující informace:

- Nelze se dostat k uzlům s čísly 0 a 4.
- Od uzlu s číslem 1 lze dojít k uzlům 2, 3 a ... 1 (dostali jsme se do tzv. uzavřeného okruhu).

Dokonce i na tomto jednoduchém příkladu se ukazují mimořádné možnosti, jaké Roy-Warshallův algoritmus poskytuje.

Vyznačuje se výjimečně prostou koncepcí i zápisem, takže jej můžeme zařadit do skupiny algoritmů, které lze souhrnně nazvat elegantní (J. Bentley je označuje výrazem programátorská perla).

Roy-Warshallův algoritmus je možné poměrně snadno upravit tak, aby nepodával pouze informaci o tom, zde existuje cesta z vrcholu x do vrcholu y , ale kromě toho poskytl také návod, jak se lze do cílového vrcholu dostat.

Abychom mohli najít cestu (samozřejmě v případě, že vůbec existuje), přiřadíme matici grafu k tzv. *matici navrhovaných cest* (ang. *routing matrix*) R . Definována je následujícím způsobem:

- $R[x, y] = 0$, pokud neexistuje cesta vedoucí z x do y .
- $R[x, y] = z$, kde z označuje následující vrchol na cestě z x do y .

Díky své konstrukci matice umožňuje přirozeným způsobem rekonstruovat cestu, která vede z daného vrcholu k jinému:

```
route.cpp
void pis(int x, int y, int R[n][n])
{
    int k;
    if(R[x][y]==0)
        cout << "Cesta neexistuje\n";
    else
    {
        cout << x << endl;
        k=x;
        while(k!=y)
        {
            k=R[k][y];
            cout << k << endl;
        }
    }
}
```

Už víme, jak na základě matice R vypsat cestu. Je tedy načase podívat se na algoritmus, který správnou cestu pro daný graf určí.

Dále představená procedura *route* předpokládá, že matice R předaná jako parametr již byla inicializována tímto způsobem:

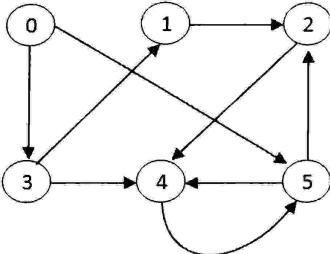
- $R[x, y] = 0$, pokud neexistuje hrana (x, y) .
- $R[x, y] = y$ v opačném případě.

Zápis procedury je triviální:

```
void route(int R[n][n])
{
    for(int x=0; x<n; x++)
        for(int y=0; y<n; y++)
            if(R[y][x]!=0) // víme, jak dojít z 'y' do 'x'
                for(int z=0;z<n;z++)
                    if( (R[y][z]==0) && (R[x][z]!=0) )
                        R[y][z]=R[y][x];
}
```

KAPITOLA 10 Prvky algoritmičnosti grafů

Algoritmus samozřejmě vychází z algoritmu, který jsme již představili, a stejně jako původní verze působí velmi prostě. Matematický důkaz toho, že funguje správně, však vůbec není jednoduchý. Podívejme se na výsledek provedení procedury route pro graf znázorněný na obrázku 10.12 (několik příkladů).



Obrázek 10.12: Vyhledávání cesty v grafu

```

Cesta z 0 do 2: 0 3 1 2
Cesta z 1 do 0: Cesta neexistuje
Cesta z 1 do 5: 1 2 4 5
Cesta z 2 do 0: Cesta neexistuje
Cesta z 4 do 2: 4 5 2
Cesta z 5 do 3: Cesta neexistuje

```

V dalším textu se budeme zabývat dalším problémem: nalezením optimální cesty v grafu z hlediska nákladů.

Floyd-Warshallův algoritmus

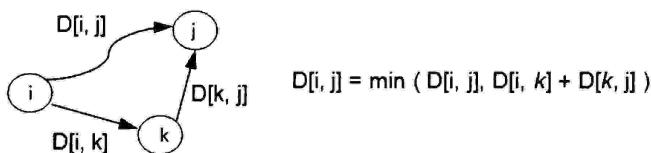
V této části se zaměříme na algoritmus, který umožňuje najít nejkratší cesty mezi všemi dvojicemi vrcholů v grafu. Aby mohl algoritmus fungovat, musíme hranám grafu připsat konkrétní hodnoty. Předpokládejme tedy, že máme matici W , která obsahuje hodnoty připsané hranám grafu:

- $W[i, i] = 0$;
- $W[i, j] = \text{hodnota připsaná hraně nebo } \infty$ (jinak: hodně velké číslo).

Princip algoritmu lze srozumitelně vysvětlit na následujícím příkladu:

Řekněme, že hledáme optimální cestu z i do j . Přitom procházíme graf a zkoušíme najít jiný mezipřesun k , který po svém vložení do cesty umožní získat lepší výsledek oproti předtím vypočítané hodnotě $D[i, j]$. Najdeme jisté k a ptáme se: Zlepší se výsledek přechodem přes vrchol k , nebo nikoli?

Obrázek 10.13 ilustruje odpověď na předchozí otázku poněkud názornější formou, než to dokáže strohý matematický vzorec (uvedený vedle).



Obrázek 10.13: Floyd-Warshallův algoritmus (1)

Je zřejmé, že v případě většího počtu vhodných mezilehlých vrcholků je potřeba zvolit nejlepší z nich. Algoritmus lze tedy rekurzivně zapsat tímto schématem:

$$D_{i,j}^{(k)} = \begin{cases} w_{i,j} & k = 0 \\ \min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) & k \geq 1 \end{cases}$$

Elementární případ se týká situace, kdy na cestě mezi i a j nejsou žádné mezilehlé vrcholy (hrana). Níže uvedený program odpovídá nejjednodušší formě *Floydova* algoritmu, která hodnotu optimální cesty pouze počítá, ale neukládá ji. Program v jazyce C++ využívá princip dynamického programování (částečné výsledky se uplatňují v pozdějších výpočtech):

```
floyd.cpp
const int n=7;
int G[n][n];

void floyd(int g[n][n])
{
    for(int k=0;k<n;k++)
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                g[i][j]=min( g[i][j], g[i][k]+g[k][j]);
}

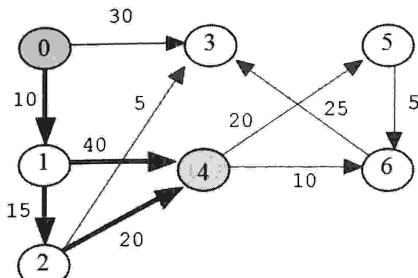
int main()
{
    for(int i=0;i<n;i++) // inicializace grafu
        for(int j=0;j<n;j++)
            G[i][j]=10000; // 10000 = smluvní "nekonečno"
    G[0][3]=30; G[0][1]=10; // graf jako na obrázku v knize
    G[1][2]=15; G[1][4]=40;
    G[2][3]=5; G[2][4]=20; G[4][5]=20;
    G[4][6]=10; G[5][6]=5; G[6][3]=25;

    floyd(G); // Vyvolání algoritmu a kontrola výsledku:
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
    {
        if(G[i][j]==10000)
            cout << i << " --> " << j << "[cesta neexistuje]\n";
        else
            if(i!=j)
                cout << i << " --> " << j << "=" << G[i][j] << endl;
    }
}
```

Podívejme se na obrázek 10.14, který znázorňuje příklad výběru optimální cesty pomocí *Floydova* algoritmu.

Předpokládejme, že nás zajímá optimální cesta z vrcholu číslo 0 do vrcholu číslo 4. Vzhledem k dosti jednoduché struktury grafu je jasné, že máme na výběr dvě cesty: 0-1-4 a o něco delší 0-1-2-4.

KAPITOLA 10 Prvky algoritmiky grafů



Obrázek 10.14: Floyd-Warshallův algoritmus (2)

Elementární výpočet ukazuje, že efektivnější je druhá trasa (náklady: 45) než první (náklady: 50). Dosti podstatnou vadou je to, že nelze zjistit optimální cestu. V případě malých grafů ji sice můžeme vyčít sami, ale u velmi rozsáhlých grafů je to prakticky nemožný úkol.

Potřebujeme tedy *Floydův* algoritmus nějak jednoduše upravit, aby jeho základní princip zůstal stejný, ale aby si přitom dokázal cestu zapamatovat.

Ukazuje se, že řešení není těžké. Původní algoritmus (viz výše uvedený výpis) je nutné změnit takto:

floyd2.cpp

```

...
if( g[i][k]+g[k][j]<g[i][j])
{
    g[i][j]=g[i][k]+g[k][j];
    R[i][j]=k;
}
  
```

Optimální cesta bude uložena do matice navrhovaných cest R. Proceduru, která bude rekonstruovat cestu podle údajů v této matici, dokážeme napsat snadno. Předpokládejme, že na začátku je matice R vynulovaná. Chceme-li přečíst optimální cestu z vrcholu i do vrcholu j, sledujeme hodnotu R[i][j]. Pokud se rovná nule, jedná se o elementární případ neboli o hranu, kterou je potřeba přejít. V opačném případě vede cesta z i do R[i][j] a poté z R[i][j] do j. Výše uvedené části cesty nemusí být elementární. Z toho je jasné patrné, že procedura má rekurzivní charakter:

```

void cesta(int i, int j)
{
    int k = R[i][j];
    if (k != 0)
    {
        cesta(i,k);
        cout << k << " ";
        cesta(k,j);
    }
}
  
```

Na závěr se podívejme na příklad vyvolání procedur, které jsme pečlivě sestavili:

```

int main()
{
    initG(); // nulování grafu, všude G[i][j] = ∞
    initR(); // nulování matice navrhovaných cest, všude R[i][j] = 0
  
```

```

G[0][3]=30; G[0][1]=10; G[1][2]=15; G[1][4]=40; G[2][3]=5;
G[2][4]=20; G[4][5]=20; G[4][6]=10; G[5][6]=5; G[6][3]=25;

floyd(G); // vyvolání algoritmu a kontrola jeho výsledků:
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
    {
        if(G[i][j]==10000)
            cout << i << " --> " << j << "[cesta neexistuje]\n";
        else
            if(i!=j)
            {
                cout << i << " --> " << j << "=" << G[i][j] << ", cesta přes:";
                cesta(i,j);
                cout << endl;
            }
    }
}

```

Některé výsledky:

```

0 --> 4=45, cesta přes: 1 2
0 --> 5=65, cesta přes: 1 2 4
0 --> 6=55, cesta přes: 1 2 4
1 --> 0[cesta neexistuje]
1 --> 4=35, cesta přes: 2
1 --> 5=55, cesta přes: 2 4
1 --> 6=45, cesta přes: 2 4
2 --> 5=40, cesta přes: 4

```

Ze struktury algoritmu je na první pohled zřejmé, že náleží do třídy $O(N^3)$. Stojí za pozornost, že třída algoritmů vyhledávajících všechny cesty v grafu je velmi podobná třídě algoritmů, které pracují s jedinou cestou (viz následující část kapitoly).

Dijkstrův algoritmus

Chceme-li zjistit nejmenší vzdálenost od *jednoho stanoveného* vrcholu grafu s (orientovaného či nikoli) ke všem zbyvajícím vrcholům, můžeme využít také klasický *Dijkstrův* algoritmus. Jedná se o nejznámější algoritmus, který umožňuje vyhledat nejkratší cestu v grafu. Jediné omezení tohoto algoritmu spočívá v tom, že dokáže zpracovat pouze graf s kladnými vahami hran, ale v oblastech, jako je kartografie, to příliš nevadí.

V popisu algoritmu se používají tyto symboly:

S – množina vrcholů, pro které již byla nalezena nejkratší cesta od počátečního uzlu,

s – počáteční uzel (tzv. zdroj),

t – cílový uzel (tzv. ústí),

$V-S$ – zbyvající vrcholy,

D – pole nejlepších odhadů vzdáleností mezi zdrojem a každým vrcholem v v grafu G ,

$w[u, v]$ – váha hrany $u-v$ v grafu,

Pr – pole předchůdců.

KAPITOLA 10 Prvky algoritmiky grafů

Algoritmus představíme v programovacím pseudojazyce, aby byl popis stručnější (ačkoli si tím samozřejmě ztěžíme případnou implementaci v konkrétním programovacím jazyce):

```
Iniciace:  
Pro každý uzel  $v$  proved'  
{ $D[v] = \infty$ ;  $Pr[v] = \text{NULL}$ ;}  
 $D[s] = 0$  // počáteční uzel!  
 $S = \text{NULL}$   
 $Q = \{\text{vlož všechny vrcholy grafu do fronty}\}$   
while not Empty( $Q$ )  
     $u = \text{vyjmí z fronty } Q \text{ prvek s nejmenším } D[u] \text{ // prioritní fronta!}$   
     $S = S \cup u$   
    pro každý vrchol  $v$  ze seznamu následníků vrcholu  $u$  proved'  
    {  
        // tzv. relaxace: kontrola, zda lze aktuální odhad  
        // nejkratší cesty do  $V$  (tzn.  $D[v]$ ) zlepšit díky  
        // přechodu přes  $u$  ( $u$  se stane předchůdcem  $v$ ):  
        if  $D[v] > D[u] + w[u,v]$  then  
        {  
             $D[v] = D[u] + w[u,v]$   
             $Pr[v] = u$   
        }  
    }  
}
```

Algoritmus sice není příliš čitelný, ale jeho princip je intuitivně jednoduchý:

- Klíčový význam má pole odhadů D . Všechny vrcholy kromě počátečního (s), o kterém víme, že má vzdálenost 0, mají v tomto poli zpočátku dočasný odhad ∞ .
- Vrcholky spojené s vrcholem s hranou získají v dalších krocích dočasný odhad, který se rovná vzdálenosti od s (hodnota hrany). Vybereme z nich vrchol v s nejmenší hodnotou odhadu $D[v]$ a sledujeme jeho následníky. Pokud od vrcholu s vede k některému z těchto následníků cesta, která prochází přes vrchol v a je kratší než dosavadní odhad, pak tento odhad podle ní zmenšíme (relaxace).
- Vyhledáme vrchol s nejmenším dočasným odhadem a pokračujeme v provádění algoritmu, dokud se nedostaneme do vrcholu t .

Pro úplnost ještě uvedeme, jak přečíst nejkratší cestu z s do t (pseudokód):

```
path ( $s, t$ )  
{  
     $S = \text{NULL}$  // prázdná sekvence  
     $u = t$   
    while(true)  
    {  
         $S = u + S$  // vlož  $u$  na začátek  $S$   
        if ( $u == s$ ) then break;  
         $u = Pr[u]$   
    }  
}
```

Dijkstrův algoritmus v nejhorším případě funguje v čase úměrném V^2 , ale pokud jeho frontu implementujeme pomocí haldy, můžeme dosáhnout výsledku $O(E \log V)$.

Bellman-Fordův algoritmus

Dijkstrův algoritmu bohužel nelze aplikovat na grafy, jejichž cesty mají záporné váhy. Algoritmus totiž předpokládá, že přidáním dalších hran se cesta může jedině prodloužit. Záporné váhy by přitom toto pravidlo narušovaly.

Richard E. Bellman a Lester Randolph Ford Jr. navrhli modifikaci Dijkstrova algoritmu. Spočívá v tom, že místo relaxace založené na vybraném uzlu, který zajišťuje minimální váhu („hladový“ přístup), provádíme relaxaci všech uzlů $|V|-1$ krát. Díky tomu se minimální vzdálenosti správně propagují po celém grafu ($|V|$ označuje počet vrcholů).

Zápis algoritmu v pseudokódu hodně připomíná Dijkstrův algoritmus:

```

Iniciace:
Pro každý uzel v proved'
    {D[v] = ∞; Pr[v] = NULL;}
    D[s] = 0           // počáteční uzel!

    // relaxace:
for i=1 to |V[G]| - 1 // počet hran minus 1
    Pro každou hranu (u,v) proved'
        if D[v] > D[u] + w(u,v) then
        {
            D[v] = D[u] + w(u,v)
            Pr[v] = u
        }
    // obsahuje graf záporné cykly?

    Pro každou hranu (u,v) proved'
        if D[v]> D[u]+ w(u,v) then
            Piš: "Graf obsahuje záporný cyklus"

```

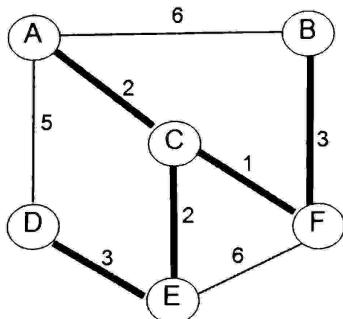
Pokud v grafu existuje cyklus, který se skládá z hran o záporných vahách, Bellman-Fordův algoritmus v takovém grafu nedokáže hledanou cestu najít (ale může tuto situaci zjistit). Složitost algoritmu se rovná $O(|V|\cdot|E|)$.

Minimální rozpínavý strom

V teorii grafů se můžeme setkat s pojmem tzv. minimálního rozpínavého stromu. Je to množina hran neorientovaného grafu, které spojují všechny vrcholy tak, aby součet všech vah (nebo délek) hran této množiny byl co nejmenší. Definici můžeme vysvětlit na příkladu grafu a jeho minimálního rozpínavého stromu z obrázku 10.15.

Zdánlivě se jedná o akademické cvičení, ale ve skutečnosti tento princip nachází řadu praktických aplikací:

- Při návrhu elektronických obvodů můžeme zjistit, jak rozmístit kontakty, aby bylo možné přenáset signály s nejmenšími ztrátami.
- Projektujeme rozvody kabelové televize po všech bytech (vrcholy) pomocí instalačních šachet (hrany).
- Potřebujeme založit telefonní síť ve vesnici, jejíž domy jsou těžko přístupné (v hornatém terénu s kopci a potoky).



Obrázek 10.15: Minimální rozpínavý strom

Nejznámější algoritmy, které řeší úlohu hledání rozpínavého stromu v grafu, se nazývají *Kruskalův*⁷ a *Primův*⁸ – oba patří do kategorie „hladových“: do stromu, který se při hledání zvětšuje, přidávají výhradně hrany s nejmenší vahou.

Kruskalův algoritmus

- Z vrcholů grafu vytvoříme les (množinu stromů) L a na začátku považujeme každý vrchol za samostatný strom.
- Ze všech hran grafu vytvoříme množinu S a setřídíme ji podle váhy hran v neklesajícím pořadí.
- Při činnosti algoritmu z množiny S vybíráme a odstraňujeme hranu s nejnižší vahou (proto je vhodné hrany nejdříve setřídit).
- Zjišťujeme, zda vybraná hrana patří do dvou různých stromů. Pokud ano, je potřeba stromy spojit. V opačném případě *hranu odmítáme*.
- V provádění algoritmu pokračujeme, dokud všechny vrcholy nejsou propojeny jedním cílovým stromem (řešením).

Zkusme sledovat činnost Kruskalova algoritmu na ukázkovém grafu z obrázku 10.15. Postupně vybírané (a odmítané) vrcholy jsou uvedeny v tabulce 10.2. Vybranou hranu je vhodné označit tučně. Díky tomu lze snadno určit, zda se následná hrana „dotýká“ dříve vytvořeného stromu na dvou místech (cyklus), tedy zda opravdu tvoří jeho část. Popis algoritmu jistě nezní příliš srozumitelně, ale stačí vzít do ruky tužku a papír a můžeme jej úspěšně vyzkoušet.

Tabulka 10.2: Kroky Kruskalova algoritmu

Vybíraná hrana (označena tučně)	Poznámka
1	Přijímáme
1, 2	Přijímáme
1, 2, 2	Přijímáme
1, 2, 3, 3	Přijímáme
1, 2, 2, 3, 3	Přijímáme
1, 2, 2, 3, 3, 5	Odmítáme (stejný strom)

⁷ Joseph Kruskal jej navrhl v roce 1956.

⁸ Robert C. Prim jej navrhl v roce 1957.

Vybíraná hrana (označena tučně)	Poznámka
1, 2, 2, 3, 3, 5, 6	Odmítáme (stejný strom)
1, 2, 2, 3, 3, 5, 6	Odmítáme (stejný strom)

Primův algoritmus

- Máme-li jistý vstupní graf $G(V,E)$, vybereme z něj libovolný vrchol a začneme vytvářet strom.
- Pro každý další vrchol, který přidáváme ke stromu, opakujeme následující kroky:
 - Přidáme ke stromu hranu s nejmenší váhou, která je přístupná z *libovolného* vrcholu do-sud vytvořeného stromu (jestliže existuje několik hran se stejnou vahou, můžeme zvolit libovolnou z nich). Kvůli rozhodování, kterou hranu vybrat, je nevhodnější setřídit při-lehlé hrany podle jejich váhy.
 - Přidáme ke stromu vybranou hranu a nový vrchol, který je přístupný po přechodu této hrany.

Sledujme činnost Primova algoritmu pro náš ukázkový graf (obrázek 10.15). Kroky algoritmu i zpracovávané objekty shrnuje tabulka 10.3 (vycházíme např. od vrcholu C).

Tabulka 10.3: Kroky Primova algoritmu

Vrcholy	Váhy dosažených hran	Vybraný vrchol
C	1, 2, 2	F
C, F	2, 2, 3, 6	E
C, E, F	2, 3, 3, 6	A
A, C, E, F	3, 3, 5, 6, 6	D
A, C, D, E, F	3, 5, 6, 6	B

Začneme-li od vrcholu C, můžeme postupovat ve třech směrech přes hrany s vahami 1, 2 a 2 (od-povídající vrcholy: F, A a E). Samozřejmě vybíráme vrchol F a v této fázi se ze dvou dostupných vrcholů C a F můžeme dostat na vrcholy s vahami 2, 2, 3 a 6 (vrchol C-F nepočítáme). Vzhledem k nejnižší váze volíme vrchol A a pokračujeme podle algoritmu, dokud nedosáhneme všech vrcholů.

Prohledávání grafů

V mnoha zajímavých algoritmických úkolech, kde se určité situace modelují pomocí grafů, je nutné grafy systematicky prohledávat – buď naslepo, nebo podle určitých praktických pravidel (tzv. *heuristik*). Prohledávání grafů se konkrétně uplatní ve všech úlohách, které souvisejí s tzv. teorií her. K této tematice se znova vrátíme v kapitole 12. Nyní se soustředíme na dvě nejjednodušší techniky procházení grafů: strategii „do hloubky“ (ang. *depth-first search*, často se označuje zkratkou DFS) a strategii „do šírky“ (ang. *breadth-first search* neboli zkráceně BFS). Při analýze příkladů se zaměříme na vlastní proces prohledávání a nebudeme uvažovat o tom, čemu slouží. Pamatujme však, že prohledávání grafu má mít nějaký cíl: nalezení optimální strategie hry, řešení hlavolamu nebo konkrétního technického problému, který je vyjádřen pomocí grafu. Algoritmy by se tedy neměly omezovat na samotný modul prohledávání, ale měly by zahrnovat také rozhodovací logiku (funkce zajišťující účel).



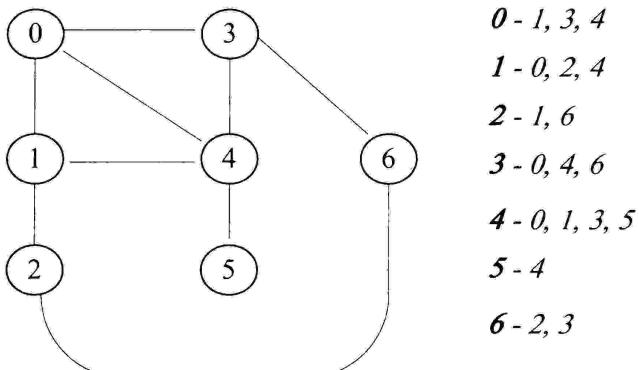
Poznámka: V dálé uvedených příkladech se setkáme výhradně s reprezentací grafů pomocí polí. Prezentované příklady budou díky tomu jednodušší, ale neměli bychom zapomínat, že tato reprezentace není jediná. V případě vyhledávacích algoritmů, které pracují s velmi rozsáhlými grafy, se pole vůbec nedají používat. Jediné východisko z takové situace nabízí reprezentace, která je založena např. na slovníku uzlů. Přitom je potřeba uvedené algoritmy modifikovat. Kvůli usnadnění je tedy opět uvedeme v pseudokódu, abychom je dokázali přepsat s ohledem na konkrétní datovou strukturu.

Strategie „do hloubky“ (sestupné prohledávání)

Název této techniky procházení grafů souvisí s topologickým tvarem cest, po kterých při analýze grafu postupujeme. Algoritmus prohledávání „do hloubky“ zkoumá danou cestu, dokud ji zcela nevyčerpá. Je pro něj typické, že nejdříve *zcela prozkoumá dříve vybranou cestu a teprve poté případně vybírá další*. Strategie „do hloubky“ provádí tzv. expanzi uzlů, což znamená, že generuje seznam potomků daného uzlu a pracuje s nimi.

Obrázek 10.16 představuje nevelký graf, na kterém můžeme problematiku vysvětlit.

Seznam přilehlých vrcholů:



Obrázek 10.16: Prohledávání grafu „do hloubky“

Vedle grafu jsou pro názornost uvedeny seznamy vrcholů *přilehlých* k danému vrcholu⁹.

Algoritmus prohledávání „do hloubky“ lze v jazyce C++ zapsat poměrně snadno:

```
depthf.cpp
const int n=7;

int G[n][n], V[n];           // G - graf nxn, V - uchovává informace,
                             // zda byl daný vrchol již testován (1) či nikoli (0)
void navstiv(int G[n][n], int V[n], int i)
{
    V[i]=1;                  // označení vrcholu jako "otestovaného"
    cout << "Zkoumání vrcholu " << i << endl;
    for(int k=0;k<n;k++)
        if(G[i][k]!=0)      // přechod existuje
```

⁹ Pořadí prvků v tomto seznamu souvisí s použitím reprezentace založené na poli, kde je pořadí uzlů do jisté míry předem určeno podle jejich indexů (které slouží jako čísla uzlů).

```

    if(V[k]==0) navstiv(G,V,k);
}

void hledej(int G[n][n], int V[n])
{
    int i;
    for(i=0;i<n;i++) V[i]=0; // vrchol zatím nebyl testován
    for(i=0;i<n;i++)
        if(V[i]==0)
            navstiv(G,V,i);
}

```

Jak je zřejmé, kód obsahuje dvě procedury: `hledej`, která inicializuje samotný proces prohledávání, a `navstiv`, která prohledávání směruje takovým způsobem, aby postupovalo opravdu „do hloubky“. Procedura `navstiv` prohledává seznam vrcholů přilehlých k vrcholu i . Její skutečný obsah (v pseudokódu) lze tedy vyjádřit takto:

```

navstiv(i)
{
    označ vrchol i jako "otestovaný";
    pro každý vrchol k přilehlý k vrcholu i
        jestliže vrchol k zatím nebyl otestován
            navstiv(k)
}

```

U této procedury má zásadní význam označení navštíveného vrcholu jako „otestovaného“, aby nedocházelo k vícenásobnému volání pro stejný vrchol.

Poznámka: V závislosti na informatické implementaci grafu (pole, seznamy nebo množiny soudních uzlů) mohou být uzly procházeny v poněkud odlišném pořadí. Základní charakter této strategie se tím však nemění.

Po spuštění programu (neboli provedení instrukce `hledej(G,V)`) se dozvíme, že vrcholy budou prohledávány v tomto pořadí: 0, 1, 2, 6, 3, 4 a 5.

Vedle grafu jsou pro názornost uvedeny seznamy vrcholů přilehlých k danému vrcholu. Zamysleme se nad tím, zda se opravdu jedná o prohledávání „do hloubky“. Podle cyklu `for` obsaženého v proceduře `hledej` se bude nejdříve prohledávat vrchol 0 a první ze všech bude také označen jako otestovaný (1). K němu přiléhají tři vrcholy: 1, 3 a 4, pro které bude znova vyvolána procedura `navstiv` (tentokrát rekurzivně). Vrchol 1 bude označen jako „otestovaný“ a poté algoritmus prozkoumá seznam vrcholů, které k němu přiléhají (0, 2 a 4). Vzhledem k tomu, že vrchol 0 byl otestován již dříve, na řadě je vrchol 2, pro který bude znova vyvolána procedura `navstiv`. (Než k tomu dojde, vrchol bude samozřejmě označen jako otestovaný.) K vrcholu 2 přiléhají vrcholy 1 a 6, ale protože vrchol 1 byl již prozkoumán, procedura `navstiv` bude vyvolána pouze pro vrchol 6 atd.

Budeme-li tuto cestu sledovat dále, můžeme rekonstruovat způsob, jakým funguje algoritmus prohledávání „do hloubky“ pro celý graf. Máme-li konkrétní zadání, musíme svůj prohledávací algoritmus samozřejmě doplnit nějakou porovnávací funkcí (podle účelu programu).

Strategie „do šírky“

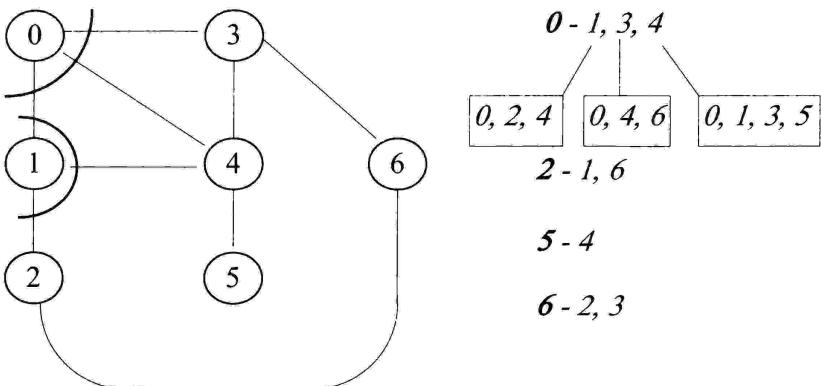
Strategie prohledávání „do šírky“ postupně zkoumá jednotlivé úrovně grafu, které mají stejnou hloubku.

KAPITOLA 10 Prvky algoritmiky grafů

Prohledávání „do šírky“ budeme analyzovat na stejném grafu jako v předchozím případě. Obrázek je však doplněn o prvky, díky nimž lze novou koncepci prohledávání snáze pochopit.

Začít můžeme například od vrcholu 0. Na seznamu přilehlých vrcholů se bude nacházet řada vrcholů 1, 3 a 4. Právě tyto vrcholy vyhledávací algoritmus prozkoumá jako první. Teprve poté zohlední seznamy přilehlých vrcholů k vrcholům, které již testoval: (0, 2, 4), (0, 4, 6) a (0, 1, 3, 5). Z toho vyplývá, že graf z obrázku 10.17 bude prohledán v tomto pořadí: 0, 1, 3, 4, 2, 6 a 5.

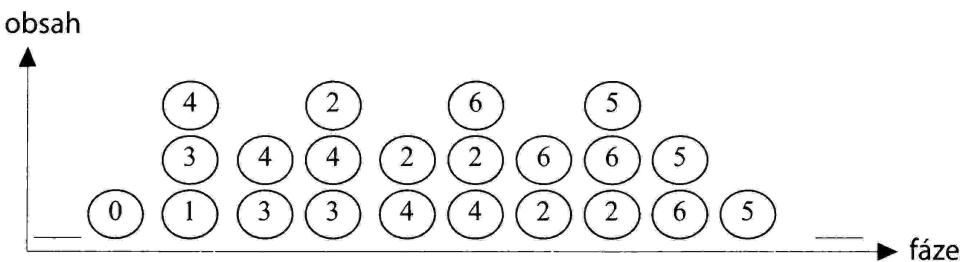
Seznam přilehlých vrcholů:



Obrázek 10.17: Prohledávání grafu „do šírky“

Jak si však při prohledávání daného vrcholu i můžeme zapamatovat, že na případné prozkoumání čekají ještě další vrcholy? Ukazuje se, že se k tomu nejlépe hodí běžná *fronta*¹⁰, která spravedlivě obslouží všechny vrcholy v pořadí, v jakém byly do fronty zařazeny.

V našem příkladu bude obsah fronty vypadat tak jako na obrázku 10.18.



Obrázek 10.18: Obsah fronty při prohledávání grafu „do šírky“

Algoritmus prohledávání „do šírky“ má v jazyce C++ následující podobu:

```

breadth.cpp
int G[n][n], V[n];
// G - graf nxn , V - uchovává informaci, zda daný vrchol
// byl již testován (1) či nikoli (0)
#include "kolejka.h"

void hledej(int G[n][n], int V[n], int i)
// začátek ve vrcholu 'i'

```

¹⁰ Viz kapitolu 5 a příklad v souboru kolejka.h.

```

{
    FIFO<int> fronta(n);
    fronta.vloz(i);

    int s;

    while(!fronta.prazdne())
    {
        fronta.obsluz(s); // vyjmutí určitého vrcholu 's' z fronty
        V[s]=1;           // označení vrcholu 's' jako "otestovaného"

        for(int k=0; k<n; k++)
            if(G[s][k]!=0) // přechod existuje
                if(V[k]==0) // vrchol 'k' zatím nebyl testován
                {
                    V[k]=1; // označení vrcholu 'k' jako "otestovaného"
                    fronta.vloz(k);
                }
    }
}

```

Smysl algoritmu lze mnohem srozumitelněji vyjádřit pomocí pseudokódu:

```

hledej(i)
{
    vlož i do fronty;
    dokud fronta není prázdná proved:
    {
        vyjmi z fronty určitý vrchol s;
        označ vrchol s jako "otestovaný";
        pro každý vrchol k přilehlý k vrcholu s
            jestliže vrchol k zatím nebyl otestován
            {
                označ vrchol k jako "otestovaný";
                vlož k do fronty;
            }
    }
}

```

V archivu ke stažení se nachází příklad, který ukazuje funkci algoritmu na konkrétním grafu. Kvůli lepší srozumitelnosti obsahuje i kontrolní instrukce (program vypisuje informace o vkládaných a vyjmávaných prvcích).

Jiné strategie prohledávání

Uzly grafu lze samozřejmě navštívit více způsoby, než kolik jich nabízejí klasické strategie prohledávání grafů, které jsme popsali v předchozích částech kapitoly. Tyto strategie však slouží jako šablony, které můžeme modifikovat. Uvedeme několik metod, o kterých jsme se zatím nezmínili:

- Strategie s návraty

V této části rozbereme variantu metody prohledávání „do hloubky“, ve které místo generování všech potomků zkoumaného uzlu generujeme pouze jednoho „potomka“. Pokud nový uzel nevyhovuje cílovému či koncovému kritériu, postupujeme dále. Když v určitém okamžiku

získaný uzel splňuje konečné kritérium prohledávání grafu nebo nedovoluje generovat nového potomka, následuje návrat k nejbližšímu předkovi, který generování potomka umožňuje. Strategie s návraty eliminuje riziko zbytečného generování uzlů. V klasické metodě „do hloubky“ nemusí být část uzlů získaných v následných krocích činnosti algoritmu vůbec testována.

■ **Strategie A***

Cílem strategie A* je vyznačit nejlevnější cestu v grafu mezi počátečním a cílovým vrcholem (nebo vrcholy). Součástí strategie je heuristická funkce $f(x) = h(x) + g(x)$, která je součtem heuristického odhadu $h(x)$ nákladů na cestu spojující uzel x s cílovým uzlem a $g(x)$ – nákladů na cestu spojující počáteční uzel s uzlem x . Algoritmus A* roku 1968 popsali Peter Hart, Nils Nilsson a Bertram Raphael.

■ **Metoda šplhání (ang. hill-climbing)**

Po expanzi uzlů z nich k další expanzi vybíráme ten nejslibnější. Při přesunech během prohledávání využíváme *lokální* optimalizaci a nepovolujeme postup opačným směrem. Strategie je proto nevratná.

Problém vhodného výběru

Na závěr pojednání o grafech představíme velmi zajímavý a komplikovaný *problém výběru* (nebo jinými slovy *minimalizace konfliktů*). Opět se přesvědčíme, že díky dobrému modelu lze snáze najít vhodné řešení.

Vzhledem k tomu, že čistě matematická formulace úlohy je dosti nesrozumitelná, rozeberme její princip na praktickém příkladu.

Představme si následující situaci:

- Máme N studentů a N témat diplomových prací.
- Ke každé diplomové práci patří jeden školitel (učitel dané fakulty). V obou směrech tedy musíme zohlednit lidský faktor.
- Všichni studenti mají vlastní názor na téma jednotlivých prací a některým z nich nepochybňě dávají přednost.
- Obdobně každý pedagog má svůj žebříček oblíbených studentů a jistě by radši spolupracoval se studentem X , kterého dobře zná, než s nesympatickým studentem Y , který pravidelně opouštěl jeho přednášky.

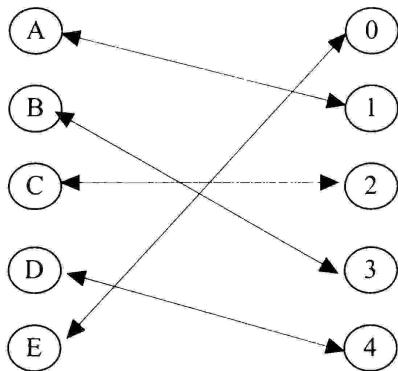
Problém výběru se samozřejmě neomezuje na akademickou oblast a můžeme se s ním v rozmanitých formách setkat v nejrůznějších oblastech. Proč jej řešíme pomocí grafů? Asi nejlépe to vysvětlí obrázek 10.19.

Obrázek představuje jedno z možných řešení problému výběru pro $N = 5$ studentů a prací. Zadání má podobu speciálního grafu, jehož uzly jsou seskupeny podle kategorií a uspořádány v řadách. Měli bychom si však uvědomit, že tato vizualizace je užitečná výhradně pro člověka. Počítač totiž nevidí žádný rozdíl mezi „hezkým“ a „ošklivým“ uspořádáním (grafická struktura konkrétního výběru je prostě graf, který obsahuje jistý počet *dvojic uzlů*). Jsou-li uzly i a j vzájemně propojené, znamená to, že byly *vybrány* (nezáleží přitom na tom, zda dobře či špatně). Z toho vyplývá, že uzel nelze použít vícekrát.

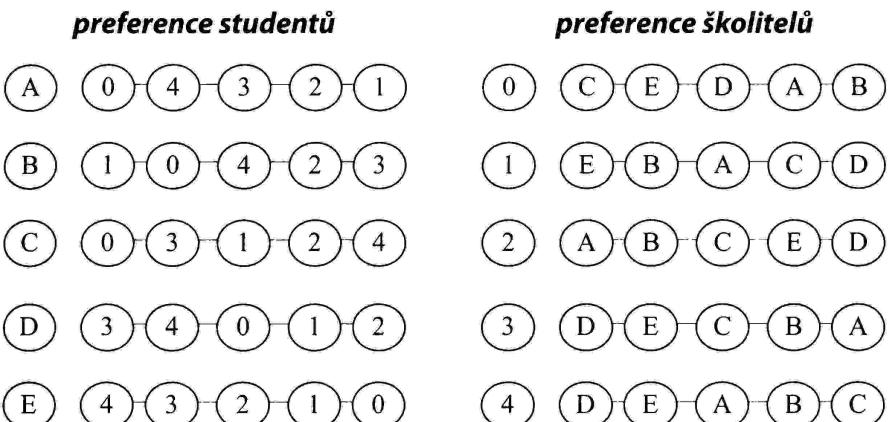
Při analýze problému výběru se nevyhnutelně dostaneme k otázce *vyjadřování preferencí*. Každý student musí mít názor na danou práci a jejího školitele a zároveň každý školitel musí jasně vyjádřit své preference týkající se příslušných studentů. Jak se ukazuje, přirozenou metodu nabízejí tzv. *se-*

znamy hodnocení: názor studenta X na diplomovou práci Y se projeví její pozicí na příslušném seznamu hodnocení diplomových prací. Do podobných seznamů pak musí školitelé seřadit studenty. Popsanou situaci představuje obrázek 10.20.

studenti **témata prací (školitelé)**



Obrázek 10.19: Problém výběru



Obrázek 10.20: Seznamy hodnocení v problému výběru

Je zřejmé, že N libovolných dvojic $\{student, práce\}$ lze sice vybrat snadno, ale vyhovět přitom velmi rozdílným požadavkům tolika osob je už složitější.

Uvažujme následující návrh: $D-0$, $E-1$, $A-2$, $B-3$, $C-4$. Nepochybň se jedná o platné řešení problému výběru (žádný uzel totiž nevyužíváme více než jednou). Je však toto řešení dobré? Student D dostal téma 0, které se na jeho seznamu nacházelo teprve na třetím místě. Vzhledem ke svým požadavkům by dal určitě přednost tématu 3. Téma 3 však připadlo studentovi B . Školitel, který odpovídá za téma 3, na svém preferenčním seznamu umístil velmi vysoko studenta D , avšak místo něj dostal studenta A ! Máme tedy dosti absurdní situaci:

$D-0$

student D dává přednost tématu 3 před 0

$B-3$

školitel tématu 3 dává přednost studentovi D před B

KAPITOLA 10 Prvky algoritmiky grafů

Výše navržené řešení se označuje jako *nestabilní*, protože vede k potenciálním osobním konfliktům. Chtěli bychom najít takový algoritmus, který by navrhl co možná nejstabilnější výběr a přitom v největší možné míře zohlednil vstupní seznamy hodnocení. Vezmeme-li v úvahu tzv. lidský faktor, jistě pochopíme, proč se toto zadání nedá jednoduše vyřešit: seznamy hodnocení totiž budou mít velmi nerovnoměrné rozložení. Určitá téma se budou líbit převážné většině studentů a jiná se zpravidla ocitnou na samém konci. Samotný výběr N dvojic z programátorského hlediska nevypadá příliš složitě, ale mnohem komplikovaněji působí kontrola, zda je výběr stabilní. Potíže vznikají proto, že potenciálních řešení, u nichž bychom měli zkонтrolovat stabilitu, je příliš mnoho. Algoritmus typu *hrubou silou*, který nejdříve generuje všechna potenciální řešení (jejichž počet je přece konečný) a poté testuje jejich stabilitu, by byl proto mimořádně neefektivní.

Problém výběru byl široce studován a zdá se, že nalezené řešení se v porovnání s tupým algoritmem typu *hrubou silou* vyznačuje jistou inteligencí. Myšlenka tohoto algoritmu spočívá v tom, že systematicky opakuje schéma *částečného výběru*:

- Student i navrhujete téma j , které se na jeho seznamu hodnocení nachází nejvýše:
 - Jestliže školitel j zatím nevybral žádného studenta, pak „spojení“ (i, j) je *dočasně* přijato.
 - Jestliže školitel j již dočasně akceptoval studenta k , pak může být spojení (k, j) přerušeno ve prospěch studenta j za podmínky, že školitel dává přednost studentovi j před dříve vybraným studentem k . V důsledku toho je student k opět volný a v jedné z následujících fází bude muset ze svého seznamu hodnocení navrhnut téma, které následuje po dříve odmítnutém.

Vyjdeme-li z údajů na obrázku 10.20, algoritmus by mohl postupovat podle fází z tabulky 10.4.

Je načase vysvětlit kód C++, který zajistí řešení problému vhodného výběru. Kód vypadá jednoduše, protože využívá pouze pole celých čísel. Tím se veškeré operace s daty mimořádně usnadňují¹¹.

Tabulka 10.4: Příklad problému výběru

Návrh	Aktuální výběry	Reakce
Student A navrhujete téma 0		Téma 0 je volné a školitel přijímá studenta A
Student B navrhujete téma 1	(A, 0)	Téma 1 je volné a školitel přijímá studenta B
C navrhujete 0	(B, 1)	Téma 0 je obsazené, ale protože jeho školitel dává přednost studentovi C před A, spojení (A, 0) je přerušeno ve prospěch (C, 0)
atd.	(A, 0) (B, 1) (C, 0)	atd.

dobor.cpp

```
#include <iostream>
using namespace std;

int dalsi[5]={-1,-1,-1,-1,-1}; // uložení poslední volby
// úplně na začátku platí dalsi[-1 + 1] = 0, později dochází k posunu
// o 1 pozici dále během dané fáze výběru

int vyber[5]={-1,-1,-1,-1,-1}; // řešení úlohy

#define TEST
```

¹¹ Všechna číselná data odpovídají obrázku 10.17.

```

int vybira[5][5]={           // preference studentů
{0,4,3,2,1}, /* A */
{1,0,4,2,3}, /* B */
{0,3,1,2,4}, /* C */
{3,4,0,1,2}, /* D */
{4,3,2,1,0}}; /* E */

// preference školitelů: rad[i][0] = číslo A na seznamu 'i',
// rad[i][1] = číslo B na seznamu 'i' atd.

int rad[5][5]={ /* A B C D E */
{3,4,0,2,1},
{2,1,3,4,0},
{0,1,2,4,3},
{4,3,2,0,1},
{2,3,4,0,1}};

```

Algoritmus výběru lze zahrnout do rozšířené funkce main:

```

int main()
{
    int student,vybirajici,skolitel,odmitnuty;

    for(student=0;student<5; student++)
    {
        vybirajici=student;
        while(vybirajici!=-1)
        {
            dalsi[vybirajici]++;
            skolitel=vybira[vybirajici][dalsi[vybirajici]];
            if(vyber[skolitel]==-1) // školitel (se svým tématem) je volný
            {
                vyber[skolitel]=vybirajici;
                vybirajici=-1;
            }
            else
                if(rad[skolitel][vybirajici]<
                    rad[skolitel][vyber[skolitel]])
                {
                    odmitnuty=vyber[skolitel];
                    vyber[skolitel]=vybirajici;
                    vybirajici=odmitnuty;
                }
        }

        for(int i=0;i<5;i++)
            cout << "(Školitel " << i << ", student " << (char)(vyber[i]+'A')
              << ")\n";
    }
}

```

Zkusme analyzovat činnost programu a ukažme přitom, jak vybírájí jednotliví studenti. Uvedme zároveň přerušená spojení:

KAPITOLA 10 Prvky algoritmiky grafů

- Začíná vybírat student A a navrhuje téma (školitele) 0.
- Téma 0 je volné a dostává je student A.
- Začíná vybírat student B a navrhuje téma (školitele) 1.
- Téma 1 je volné a dostává je student B.
- Začíná vybírat student C a navrhuje téma (školitele) 0.
- Školitel 0 zavrhuje svůj aktuální výběr studenta A ve prospěch studenta C.
- Začíná vybírat odmítnutý student A a navrhuje téma (školitele) 4.
- Téma 4 je volné a dostává je student A.
- Začíná vybírat student D a navrhuje téma (školitele) 3.
- Téma 3 je volné a dostává je student D.
- Začíná vybírat student E a navrhuje téma (školitele) 4.
- Školitel 4 zavrhuje svůj aktuální výběr studenta A ve prospěch studenta E.
- Začíná vybírat student A a:
 - Navrhuje téma (školitele) 3.
 - Navrhuje téma (školitele) 2.
 - Téma 2 je volné a dostává je student A.

Konečné výsledky:

(Školitel 0, student C)
(Školitel 1, student B)
(Školitel 2, student A)
(Školitel 3, student D)
(Školitel 4, student E)

Popsaný algoritmus výběru není ideální. Jak se však snadno můžeme prakticky přesvědčit, lineární charakter cyklu **for**, kvůli němuž jsou aktivními účastníky výhradně studenti (oni totiž navrhují a školitelé pouze pasivně čekají na jejich nabídky), nemá vliv na spravedlivost závěrečného výsledku. V komplikovanějších verzích uvedeného algoritmu si pasivní a aktivní účastníci v jednotlivých fázích vyměňují role. Záměrně jsme však zvolili jednodušší verzi algoritmu, která přesto umožňuje předvést zajímavou techniku řešení zdánlivě obtížných problémů.

Shrnutí

Tím můžeme své krátké setkání s grafy zakončit. Jak jsme již zmínili v úvodu kapitoly, omezili jsme se pouze na některé části teorie grafů. Uvedené informace byly vzhledem k obsáhlosti tematiky značně stručné, což jim však neubírá na užitečnosti. Možná čtenáře dokonce motivují k tomu, aby sáhli po publikacích citovaných na začátku kapitoly.

Úlohy

Vybrat reprezentativní sadu rekurzivních úloh nebylo vůbec snadné. Tato oblast je značně rozsáhlá a svým způsobem je zajímavé prakticky vše, co zde najdeme. Stejně jako na jiných místech knihy nakonec rozhodl praktické aspekty a jednoduchost.

Úloha 1

Zamysleme se nad tím, jak pomocí grafů modelovat strukturu Internetu. Co může být uzlem a co hranou? Pro jednoduchost pomyňme lokální síť spolu s koncovými uživateli a bezdrátové spoje i optická vlákna považujme za stejné transportní médium.

Úloha 2

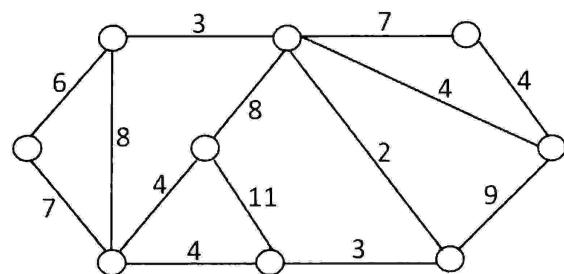
Jak pomocí slovníku uzlů (seznamu sousedství) modelovat *neorientovaný* graf?

Úloha 3

Pro jednoduchý graf (může to být jeden z grafů znázorněných na obrázcích v této kapitole) spočítejte, kolik místa obsadí v paměti počítače jeho reprezentace založená na poli, resp. na seznamu. Přitom je potřeba přijmout určité předpoklady, např. že jednotkami mohou být uzly i hrany, a zohlednit případné ukazatele a všechny pomocné struktury, budou-li v daném případě potřebné.

Úloha 4

Na graf z obrázku 10.21 aplikujte Kruskalův a Primův algoritmus.



Obrázek 10.21: Rozpínavý strom – cvičení

Numerické algoritmy

Desítky let se počítače používaly primárně a hlavně k urychlení výpočtů (mnoho lidí dodnes „počítač“ považuje za výkonnější „kalkulačku“). Oblast těchto aplikací je stále aktuální. Měli bychom si však uvědomit, že opakované vymýšlení stejných řešení má jen minimální praktický význam. V uplynulých letech se objevila celá řada hotových programů, které dokáží řešit typické matematické problémy (např. hledat řešení soustav rovnic, interpolovat a approximovat, integrovat a derivovat, provádět symbolické úpravy atd.). Těm, kdo potřebují používat pokročilé matematické funkce, lze tedy doporučit nákup vhodného nástroje, k nimž patří mj. Matlab (<http://www.mathworks.com>), Mathcad (<http://www.mathsoft.com>) nebo Mathematica (<http://www.wolfram.com>). Tyto softwarové balíky, které občas mívají i bezplatné verze, se neomezují jen na automatizaci výpočtů, ale navíc spolupracují s jinými programovacími jazyky a poskytují grafickou prezentaci výsledků.

V této kapitole představíme několik užitečných metod z oblasti numerických algoritmů, které lze potenciálně uplatnit v rámci větších programátorských projektů. Nebudeme příliš zabíhat do matematických důkazů uvedených poznatků, ale pokusíme se ukázat, jakým způsobem je možné numerický algoritmus převést na spustitelný kód jazyka C++. Podoba algoritmů popsaných v této kapitole vychází hlavně z těchto prací: skripta [Kla87] dostupného v Polsku a klasického díla [Knu10]. Čtenářům by však neměly činit potíže ani jiné příručky, které pojednávají o tématu numerických algoritmů, protože jich v posledních letech vyšlo poměrně hodně. Všechny programy přetisklé v této kapitole je nutné brát jako „výukové“ implementace, které poskytují představu o využití jazyka C++ při řešení výpočetních úloh. Méně nároční uživatelé je však mohou převzít jako hotové „kuchařské recepty“.

Vyhledávání nulových bodů funkcí

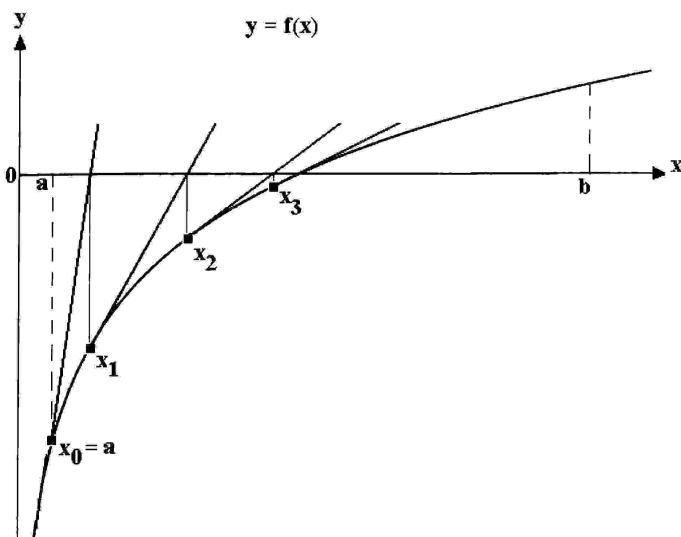
Matematici dosti často stojí před úkolem vyhledat nulové body funkce. Existuje mnoho numerických metod, které umožňují tento úkol vyřešit pomocí počítače. V této kapitole se omezíme na jednu z jednodušších – tzv. *Newtonovu metodu*. Princip této metody zjednodušeně spočívá v systematickém přibližování k nulovému bodu pomocí tečen ke křivce, jak je patrné na obrázku 11.1.

V této kapitole:

- Vyhledávání nulových bodů funkcí
- Iterativní výpočet hodnot funkce
- Interpolace funkcí Lagrangeovou metodou
- Derivování funkcí
- Integrování funkcí Simpsonovou metodou
- Řešení soustav lineárních rovnic Gaussovou metodou
- Závěrečné poznámky

KAPITOLA 11 Numerické algoritmy

Newtonova metoda klade na zkoumanou funkci $y=f(x)$ řadu omezení. V analyzovaném intervalu $[a, b]$ se například nachází právě jeden kořen, funkce na okrajích intervalu nabývá hodnot lišících se znaménkem a její první a druhá derivace v tomto intervalu nemění své znaménko.



Obrázek 11.1: Newtonův algoritmus vyhledávání nulových bodů

Z hlediska programátora lze *Newtonův* algoritmus vyjádřit jako iterativní opakování následujících operací (i označuje fázi iterace):

- $$z_i = z_{i-1} - \frac{f(z_{i-1})}{f'(z_{i-1})}$$

- stop, jestliže $|f(z_i)| < \epsilon$

Symbol ϵ označuje určitou kladnou konstantu (např. 0,00001), která zajišťuje, že se algoritmus zastaví. Úplně na začátku samozřejmě inicializujeme proměnnou z_0 jistou počáteční hodnotou. Navíc potrebujeme znát explicitní rovnice f a f' (funkci a její první derivaci)¹.

Navrhne rekurzivní² verzi algoritmu, která jako parametry přijímá mj. ukazatele na funkce reprezentující f a f' . Podívejme se na příklad, jak lze pomocí *Newtonovy* metody vypočítat nulové body funkce $f(x) = 3x^2 - 2$. Procedura *nula* přesně odráží schéma, které jsme uvedli na začátku:

```
newton.cpp
const double epsilon=0.0001;
double f(double x)          // funkce f(x)=3x^2-2
{
    return 3*x*x-2;
}
double fp(double x)         // derivace f'(x)=(3x^2-2)'=6x
{
    return 6*x;
}
```

1 Musíme je do kódu programu v C++ zapsat „natvrdo“.

2 Všimněme si, že se jedná o příklad tzv. koncové rekurze, kterou lze přirozeným způsobem převést na algoriticky rovnocennou iterativní verzi.

```

double nula(double x0, double(*f)(double), double(*fp)(double) )
{
    if(fabs( f(x0) ) < epsilon)
        return x0;
    else
        return nula(x0-f(x0)/fp(x0), f, fp);
}
int main()
{
    cout << "Nulový bod funkce 3x*x-2 se rovná "<<nula(1,f,fp)<<endl;
    // výsledek 0,816497
}

```

Díky ukazatelům na funkci je procedura `nula` univerzálnější, ale samozřejmě nic nebrání tomu, abychom tyto funkce používali přímo.

Iterativní výpočet hodnot funkce

S efektivním postupem výpočtu hodnoty mnohočlenů se seznámíme v kapitole 13, kde popíšeme tzv. *Hornerovo schéma*. Nyní se budeme zabývat algoritmem na iterativní výpočet hodnoty funkce, který se sice v praxi nepoužívá příliš často, ale občas může být užitečný.

Předpokládejme, že nás zajímá jistá funkce $y = f(x)$. Převedme ji do tzv. implicitní formy:

$$F(x, y) = 0.$$

Označme parciální derivaci, která se počítá podle proměnné y , jako $F'_y(x, y)$, kde $F'_y(x, y) \neq 0$ v každé iteraci.

S jistým zjednodušením můžeme pomocí Newtonovy metody vypočítat hodnotu pro určité x iterativním způsobem:

- $y_{n+1} = y_n - \frac{F(x, y_n)}{\frac{\partial F}{\partial y}(x, y_n)}$

- stop, jestliže $|y_{n+1} - y_n| < \epsilon$

Počáteční hodnota y_0 by měla být co nejbližší hledané hodnotě y .

Problém výpočtu hodnoty funkce $f(x)$ jsme v popsané metodě převedli na hledání nulového bodu funkce $F(x, y)$.

Výhody *Newtonovy metody* se projevují zvláště u některých funkcí, jejichž kvocient se může (ale nemusí) značně zjednodušit.

Příklad: Pro $y = 1/x$ máme $F(x, y) = x - 1/y$ a $F'_y(x, y) = 1/y^2$. Z podmínky $F(x, y) = 0$ dostáváme iterativní vzorec $y_{n+1} = 2y_n - x(y_n)^2$.

Výše uvedené vzorce lze vyjádřit následujícím programem v jazyce C++:

```

wartf.cpp
const double epsilon=0.00000001;

double hodn(double x, double yn)
{
    double yn1=2*yn-x*yn*yn;
    if(fabs(yn-yn1)<epsilon) // fabs = absolutní hodnota
        return yn1;
}

```

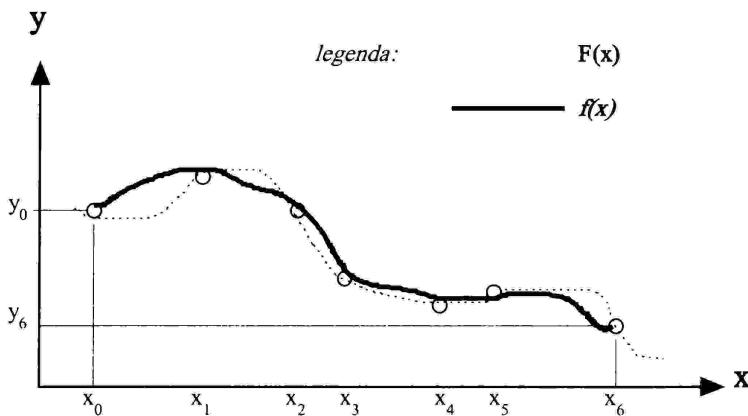
```

    else
        return hodn(x,yn1);
}
int main()
{
    cout << "Funkce y=1/x má pro x=7 hodnotu "<<hodn(7,0.1);
    // výsledek: 14,2857
}

```

Interpolace funkcí Lagrangeovou metodou

V předchozích částech této kapitoly jsme často využívali explicitní vzorce funkce a její derivace. Jak ovšem postupovat, disponujeme-li částí grafu funkce (tzn. známe její hodnoty pro konečnou množinu argumentů), případě by výpočet funkce podle vzorce byl vzhledem k jeho složitosti velmi časově náročný? V obou případech si můžeme pomocí metodami tzv. *interpolace funkce*. Přitom se k funkci přibližujeme pomocí jednodušší funkce (např. mnohočlenu určeného stupně) tak, aby interpolaci funkce procházela známými body grafu původní funkce (viz obrázek 11.2).



Obrázek 11.2: Interpolace funkce $f(x)$ pomocí mnohočlenu $F(x)$

V příkladu znázorněném na obrázku máme k dispozici 7 dvojic $(x_0, y_0), \dots, (x_6, y_6)$ a na jejich základě dokážeme vypočítat mnohočlen $F(x)$, díky němuž lze hodnoty $f(x)$ určovat mnohem snáze (ačkoli občas mohou být výsledky daleko od pravdy).

Interpolaci mnohočlen lze konstruovat pomocí výpočetně náročného *Vandermondova determinantu*, který umožňuje najít koeficienty hledaného mnohočlenu³. Pokud nás ovšem zajímá pouze hodnota funkce v určitém bodě z , existuje prostší a efektivnější *Lagrangeova metoda*:

$$F(z) = (z - x_0)(z - x_1) \dots (z - x_n) \sum_{j=0}^n \frac{y_j}{(z - x_j) \prod_{i=0, i \neq j}^n (x_j - x_i)}$$

Navzdory dosti odstrašující formě lze výše uvedený vzorec snadno převést na kód jazyka C++, který obsahuje dva vnořené cykly `for`.

³ Viz např. https://cs.wikipedia.org/wiki/Vandermondova_matici.

U následujícího výpisu však hrozí riziko dělení nulou (pokud se hodnota z rovná některému z uzlů). Na procičení můžete uvedenou funkci vylepšit a rozšířit ji o kontrolu vypočítaných hodnot a příslušnou signalizaci chyb.

```
interpol.cpp
const int n=3; // stupeň interpolacního mnohočlenu

// pole hodnot funkce ( $y[i]=f(x[i])$ )
double x[n+1]={3.0, 5.0, 6.0, 7.0};
double y[n+1]={1.732, 2.236, 2.449, 2.646};
// (ve skutečnosti funkce na výpočet kořene z hodnoty 'x'
double interpol(double z, double x[n], double y[n])
{ // vrácení hodnoty funkce v bodě 'z'
    double wnz=0, om=1, w;
    for(int i=0; i<=n; i++)
    {
        om=om*(z-x[i]);
        w=1.0;
        for(int j=0; j<=n; j++)
            if(i!=j) w=w*(x[i]-x[j]);
        wnz=wnz+y[i]/(w*(z-x[i]));
    }
    return wnz=wnz*om;
}
int main()
{
    double z=4.5;
    cout << "Funkce sqrt(x) má v bodě " << z << " hodnotu "
        << interpol(z,x,y) << endl;
}
```

Derivování funkcí

V předchozích částech této kapitoly jsme často využívali vzorce funkce a její derivace, které byly přímo zapsány do kódu C++. Výpočet derivace však občas bývá náročný a pracný. Hodí se proto metody, které tento problém dokáží vyřešit a obejdou se přitom bez explicitního vzoru funkce.

Mezi oblíbené metody numerického derivování patří tzv. *Stirlingův* vzorec. Popis jeho odvození přesahuje rámec této publikace. Předvedeme proto pouze praktické výsledky a do matematických důkazů se nebudeme pouštět.

Stirlingův vzorec umožňuje jednoduše vypočítat derivace f' a f'' v bodě x_0 pro jistou funkci $f(x)$, jejíž hodnoty známe v tabulkové formě:

$$\dots(x_0-2h, f(x_0-2h)), (x_0-h, f(x_0-h)), (x_0, f(x_0)), (x_0+h, f(x_0+h)), (x_0+2h, f(x_0+2h))\dots$$

Parametr h je jistý stálý krok v oboru hodnot x .

Stirlingova metoda využívá tzv. *pole centrálních rozdílů*, jehož konstrukce je znázorněna na obrázku 11.3. Rozdíly δ se počítají shodným způsobem v celém poli, např.:

$$\delta f(x_0 - 3/2 h) = f(x_0 - h) - f(x_0 - 2h) \text{ atd.}$$

KAPITOLA 11 Numerické algoritmy

Přijmeme-li zjednodušený předpoklad, že budeme vždy počítat derivace pro centrální bod $x = x_0$, nabývají Stirlingovy vzorce následující podoby:

$$f'(x) = \frac{1}{h} \left(\frac{\delta f(x - \frac{1}{2}h) + \delta f(x + \frac{1}{2}h)}{2} - \frac{1}{6} \frac{\delta^3 f(x - \frac{1}{2}h) + \delta^3 f(x + \frac{1}{2}h)}{2} + \frac{1}{30} \frac{\delta^5 f(x - \frac{1}{2}h) + \delta^5 f(x + \frac{1}{2}h)}{2} + \dots \right)$$

$$f''(x) = \frac{1}{h^2} \left(\delta^2 f(x) + \frac{1}{12} \delta^4 f(x) - \frac{1}{90} \delta^6 f(x) + \dots \right)$$

x	$f(x)$	$\delta f(x)$	$\delta^2 f(x)$	$\delta^3 f(x)$	$\delta^4 f(x)$
$x_0 - 2h$	$f(x_0 - 2h)$				
$-$		$\delta f = (x_0 - 3/2 h)$			
$x_0 - h$	$f(x_0 - h)$		$\delta^2 f(x_0 - h)$		
		$\delta f = (x_0 - 1/2 h)$		$\delta^3 f = (x_0 - 1/2 h)$	
x_0	$f(x_0)$		$\delta^2 f(x_0)$		$\delta^4 f(x_0)$
		$\delta f = (x_0 + 1/2 h)$		$\delta^3 f = (x_0 + 1/2 h)$	
$x_0 + h$	$f(x_0 + h)$		$\delta^2 f(x_0 + h)$		
		$\delta f = (x_0 + 3/2 h)$			
$x_0 + 2h$	$f(x_0 + 2h)$				

Obrázek 11.3: Pole centrálních rozdílů v Stirlingově metodě

Kontrolních bodů funkce může být samozřejmě mnohem více než 5. Zde se zaměříme na velmi jednoduchý příklad s pěti hodnotami funkce, který vede k poli centrálních rozdílů nízkého rádu. Vzorový program v C++, který počítá derivace určité funkce $f(x)$, může vypadat takto:

```
pochodna.cpp
const int n=5; // řád počítaných centrálních rozdílů činí n-1

double t[n][n+1]=
{
    {0.8, 4.80}, // dvojice (x[i], y[i]) pro y=5x*x+2*x
    {0.9, 5.85}, // (zapsané jsou dva první sloupce, a nikoli řádky)
    {1, 7.00},
    {1.1, 8.25},
    {1.2, 9.60}
};

struct DERIVACE{double f1,f2;};

DERIVACE stirling(double t[n][n+1])
// funkce vrací hodnoty f'(z) a f''(z), kde z je centrálním
// prvkem: zde t[2][0]; pole 't' je nutné předtím centrálně
```

```
// inicializovat, jeho správnost se nekontroluje
{
    DERIVACE res;
    double h=(t[4][0]-t[0][0])/((double)(n-1)); // krok argumentu 'x'
    for(int j=2;j<=n;j++)
        for(int i=0;i<=n-j;i++)
            t[i][j]=t[i+1][j-1]-t[i][j-1];
    res.f1=((t[1][2]+t[2][2])/2.0-(t[0][4]+t[1][4])/12.0)/h;
    res.f2=(t[1][3]-t[0][5]/12.0)/(h*h);
    return res;
}

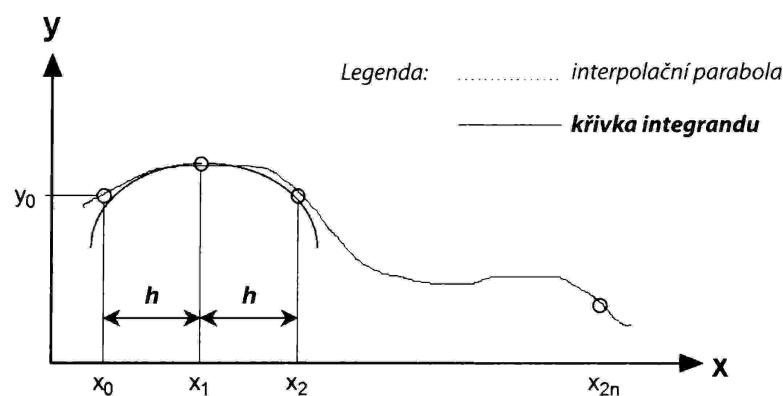
int main()
{
    DERIVACE res=stirling(t);
    cout << "f'=" << res.f1 << ", f''=" << res.f2 << endl;
}
```

Když se už zabýváme numerickým derivováním, měli bychom zdůraznit, že je poměrně nepřesné. Čím je hodnota parametru h menší, tím větší vliv na výsledek mají chyby zaokrouhlení. Zvětšování parametru h však odporuje ideji Stirlingovy metody (která má přece napodobovat skutečné derivování!). Stirlingova metoda se nehodí k derivování na okrajích intervalů proměnlivosti argumentu funkce. Pokud vás toto téma zajímá, můžete najít další informace v příslušné literatuře. Téma je totiž širší, než se může zdát.

Integrování funkcí Simpsonovou metodou

Některé funkce se těžko integrují, protože výpočet jejich integrálu je dosti symbolicky náročný. Abychom dostali požadovaný výsledek, musíme občas provést hodně obtížných transformací (např. substituce, rozklady na posloupnosti atp.).

Můžeme si však pomocí interpolačními metodami (které komplikovanou funkci nahradí přibližnou a výpočetně jednodušší formou). Princip numerického integrování je znázorněn na obrázku 11.4.



Obrázek 11.4: Přibližné integrování funkce

KAPITOLA 11 Numerické algoritmy

V dané fázi i jsou tři po sobě následující body integrované funkce proloženy parabolou, což zajišťuje poměrně dobrou přesnost integrování (pro některé křivky můžeme získat úplně shodné výsledky jako při ruční integraci). Dílčí integrál analyzovaného fragmentu bude určen vztahem:

$$\int_{x_0}^{x_2} f(x)dx = \frac{f(x_0) + 4f(x_1) + f(x_2)}{3h}$$

Uvedený vzorec, který se označuje jako *Simpsonův*, stačí aplikovat na každý interval integrované oblasti, který zahrnuje tři následné body křivky $f(x)$. Metoda vyžaduje pouze to, aby byly vybrané intervaly h stejně dlouhé. Když tedy budeme předpokládat hranice integrování od a do b , při rozdělení na $2n$ částí dostaneme rovnici $h = (b - a)/2n$. Globální integrál se bude samozřejmě rovnat součtu dílčích integrálů, které lze vypočítat následujícím způsobem:

```
simpson.cpp
const int n=4; // počet bodů = 2n+1

// funkce x*x-3*x+1 v intervalu [-5,3]
double f[2*n+1]={41, 29, 19, 11, 5, 1, -1, -1, 1};

double simpson(double f[2*n+1], double a, double b)
// funkce vrací integrál funkce f(x) v intervalu [a,b],
// ježíž hodnoty jsou uvedeny tabulkově v 2n+1 bodech
{
    double s=0,h=(b-a)/(2.0*n);
    for(int i=0;i<2*n;i+=2) // skok co dva body
        s+=h*(f[i]+4*f[i+1]+f[i+2])/3.0;
    return s;
}
```

Integrování Simpsonovou metodou lze samozřejmě využít i při integrování funkce, kterou známe nejen v tabulkové, ale i v analytické podobě. Stačí pouze předat příslušný ukazatel na funkci, která je vyjádřena v analytické verzi:

```
double fun(double x)
{
    return x*x-3*x+1;
}
double simpson_f(double(*f)(double), double a, double b, int N)
// funkce vrací integrál funkce f(x) známé v podobě vzorce
// na intervalu [a, b], N - počet částí
{
    double s=0,h=(b-a)/(double)N;
    for(int i=1;i<=N;i++)
        s+=h*(f(a+(i-1)*h)+4*f(a-h/2.0+i*h)+f(a+i*h))/6.0;
    return s;
}
```

Dále je uvedena forma volání obou variant funkce „simpson“:

```
int main()
{
```

```

cout << "Hodnota integrálu =" << simpson(f,-5,3) << endl;      // 82.667
cout << "Hodnota integrálu =" << simpson_f(fun,-5,3,8) << endl; // 82.667
}

```

Řešení soustav lineárních rovnic Gaussovou metodou

S úkolem vyřešit soustavu lineárních rovnic se setkáváme v mnoha oblastech, zejména technických. Vzhledem k tomu, že na samotném řešení soustav rovnic není nic objevného (učili jsme se to přece již na základní škole), s výhodou využijeme počítačovou proceduru, která tuto únavnou práci vykoná za nás.

Aby mohl počítač vyřešit danou soustavu rovnic, musíme ji nejdříve zapsat v rozšířeném tvaru, tzn. ponecháváme koeficienty rovné nule a píšeme proměnné v pevném pořadí. Díky tomu lze správně sestavit rozšířenou matici sestavy rovnic.

Soustavu rovnic:

$$5x+z = 9$$

$$x-z+y = 6$$

$$2x-y+z = 0$$

je tedy nutné vyjádřit ve tvaru:

$$5x+0y+1z = 9$$

$$1x+1y-1z = 6$$

$$2x-1y+1z = 0$$

což nám umožní kompletně ji zapsat v maticové formě:

$$\begin{pmatrix} 5 & 0 & 1 \\ 1 & 1 & -1 \\ 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 9 \\ 6 \\ 0 \end{pmatrix}$$

Po vynásobení těchto matic bychom se měli dostat zpět ke klasickému čitelnému zápisu.

Výhoda maticové reprezentace spočívá v tom, že umožňuje umístit všechny číselné koeficienty do jednoho pole $N \times (N+1)$ a při řešení soustavy s těmito koeficienty počítat. Operace s touto maticí odpovídají transformacím, které provádíme se samotnými rovnicemi (např. eliminaci proměnných nebo sčítání stran rovnic).

Vzhledem ke snadné programové implementaci se při řešení soustav lineárních rovnic velmi často uplatňuje tzv. *Gaussova eliminace*. Zahrnuje dvě hlavní etapy: převod matice soustavy rovnic na tzv. *trojúhelníkovou matici*, která pod úhlopříčkou obsahuje nuly, a *zpětnou redukci*, kdy usilujeme o výpočet hodnoty hledaných proměnných:

- Nejdříve eliminujeme proměnnou x ze všech řádků kromě prvního (využíváme přitom klasické přičítání aktuálního řádku, který vynásobíme vhodným koeficientem).
- Následně stejným způsobem pracujeme s proměnnou y a druhým řádkem, abychom nakonec získali trojúhelníkovou matici.

Podívejme se na příklad:

- eliminace proměnné x z řádků 2 a 3 (vliv přičtení řádku je patrný v následujícím kroku):

KAPITOLA 11 Numerické algoritmy

$$\begin{array}{l} *(-0, 2) \quad 5x + 0y + 1z = 9 \\ \quad \quad \quad \rightarrow 1x + 1y - 1z = 6 \\ \quad \quad \quad 2x - 1y + 1z = 0 \end{array} \quad *(-0, 4)$$

- eliminace proměnné y z řádku 1 a 3 (na prvním řádku už není nutné nic dělat):

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \quad \rightarrow *1 \\ 0x - 1y + 0,6z = -3,6 \end{array}$$

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \quad \rightarrow *1 \\ 0x - 1y + 0,6z = -3,6 \end{array}$$

- nakonec získáváme trojúhelníkovou matici:

$$\begin{array}{l} 5x + 0y + 1z = 9 \\ 0x + 1y - 1,2z = 4,2 \\ -0x + 0y - 0,6z = 0,6 \end{array}$$

Máme-li matici v této formě, můžeme se již pokusit o vyčíslení proměnných (pomocí *zpětné redukce*, kdy postupujeme od posledního řádku soustavy k prvnímu):

$$\begin{array}{l} z = -0,6/0,6 = -1 \\ y = 1,2z + 4,2 = 3 \\ x = (9 - z)/5 = 2 \end{array}$$

Metoda není složitá, i když její zápis v jazyce C++ zpočátku nemusí vypadat příliš čitelně. Jedinou nebezpečnou operací v metodě *Gaussovy eliminace* je právě eliminace proměnných, která může občas vést k dělení nulou (pokud se proměnná eliminovaná ve fázi i v dané rovnici nevy-skytuje). Ovšem díky tomu, že na řešení soustavy nemá žádný vliv záměna pořadí řádků, můžeme se nebezpečí dělení nulou snadno vyhnout právě tímto postupem.

Může se samozřejmě ukázat, že řádky nelze zaměnit, protože není splněna podmínka, že pod řádkem i musí existovat takový řádek, který by měl u konfliktní proměnné nenulový koeficient. V tomto případě soustava rovnic nemá řešení, což je z hlediska uživatele také užitečná informace.

Toto je plná verze programu, který zajišťuje *Gaussovou*⁴ eliminaci a kromě toho obsahuje ukázková data:

```
gauss.cpp
const int N=3;
double x[N]; // výsledky

double a[N][N+1]=
{
    {5, 0, 1, 9},
    {1, 1,-1, 6},
    {2, -1, 1, 0}
};

int gauss(double a[N][N+1], double x[N])
{
```

⁴ http://www.algorytm.org/index.php?option=com_content&task=view&id=82&Itemid=28

```

int max;
double tmp;
for(int i=0; i<N; i++) // eliminace
{
    max=i;
    for(int j=i+1; j<N; j++)
        if(fabs(a[j][i])>fabs(a[max][i])) // fabs = absolutní hodnota
            max=j;
    for(int k=i; k<N+1; k++) // nahrazení řádků hodnotami
    {
        tmp=a[i][k];
        a[i][k]=a[max][k];
        a[max][k]=tmp;
    }
    if(a[i][i]==0)
        return 0; // soustava nemá řešení
    for(int j=i+1; j<N; j++)
        for(int k=N; k>=i; k--) // násobení řádku j "nulujícím" koeficientem:
            a[j][k]=a[j][k]-a[i][k]*a[j][i]/a[i][i];
    for(int j=N-1; j>=0; j--) // zpětná redukce
    {
        tmp=0;
        for(int k=j+1; k<=N; k++)
            tmp=tmp+a[j][k]*x[k];
        x[j]=(a[j][N]-tmp)/a[j][j];
    }
    return 1; // všechno v pořádku
}

int main()
{
    if(!gauss(a,x))
        cout << "Rovnice (1) nemá řešení!\n";
    else
    {
        cout << "Řešení:\n";
        for(int i=0;i<N;i++)
            cout << "x["<<i<<"]="<<x[i] << endl;
    }
}

```

Závěrečné poznámky

V této krátké kapitole jsme nemohli otevřít mnoho otázek z oblasti numerických výpočtů, ale představili jsme alespoň několik metod, které se v praktických programech používají nejčastěji. Platí to, co jsme uvedli v úvodu kapitoly. Stojí ještě za zmínu, že implementování numerických algoritmů v jazyce C++ je občas poněkud násilné, protože modelování úkolů, které mají čistě numerickou povahu, tento jazyk nijak přímo neusnadňuje. Matematikům a fyzikům, kteří potřebují

KAPITOLA 11 Numerické algoritmy

výkonné výpočetní nástroje, lze místo C++ doporučit některou z moderních verzí jazyka *Fortran*. Nejde sice o univerzální jazyk (na rozdíl např. od jazyků C++ a Pascal), ale s jeho komplátorem se obvykle dodávají knihovny s mnoha výpočetními procedurami (na inverzi matic, integrování, interpolace atd.) – tedy všechny procedury, které musí programátor v jazyce C++ obvykle psát od začátku. Také pro jazyk C++ jsou samozřejmě k dispozici numerické knihovny (viz např. GNU Scientific Library – <http://www.gnu.org/software/gsl>).

KAPITOLA 12

Mohou počítače myslet?

V této kapitole:

- Přehled oblastí zájmu umělé inteligence
- Reprezentace problémů
- Hry pro dvě osoby a stromy her
- Algoritmus mini-max

Když do obecné příručky algoritmiky zahrneme kapitolu věnovanou oblasti, která má dosti zavádějící název „umělá inteligence“, můžeme narazit na několik problémů. Především se jedná o natolik rozlehlu oblast, že je těžké vytvořit nějaký kvalitní souhrn, který by problematiku dokázal osvětlit bez přílišného zjednodušení. Tento úkol je téměř nesplnitelný. Do jedné přihrádky zvané umělá inteligence se dostává hodně oborů, které se značně liší: teorie her, plánování, evoluční algoritmy (vyhledávají nejlepší řešení na principu, který napodobuje biologickou evoluci), expertní systémy atd. Problémy související s umělou inteligencí navíc obecně bývají dosti obtížné a je potřeba hodně úsilí, abychom je dokázali podat zajímavým způsobem. Bezpochyby se to podařilo Nilssonovi, který tomu ovšem v knize [Nil82] věnoval několik stovek stran!

Snažil jsem se proto vybrat několik zajímavých příkladů a popsat je natolik jednoduchým jazykem, aby jim dokázala bez přílišných potíží porozumět i osoba bez informatického vzdělání. Volba padla na prvky teorie her. Toto téma souvisí s věčnou lidskou touhou najít pro danou hru optimální strategii, která by zaručeně vedla k vítězství.

Otzáka v nadpisu této kapitoly připomíná, jakým způsobem o počítačích často uvažují laici. Z toho, že nějaký stroj dokáže hrát, kreslit či animovat, jim jednoznačně vyplývá, že umí myslet. „To je samozřejmě mylné přesvědčení,“ prohlásí informatik, který ví, že ve skutečnosti jsou počítače jen komplikované automaty, jejichž možnosti závisí na programech, kterými je vybavíme. Právě v těchto programech spočívá možnost *simulování inteligenčního chování* počítače, kdy počítač postupuje podobným způsobem jako člověk. V současnosti umíme dosáhnout jen toho, aby počítač *napodoboval* inteligenční chování, protože lidský mozek¹ je mnohem složitější než ten nejkomplikovanější počítač. Nemůžeme však vyloučit, že se za několik let objeví technologie, která by umožnila zkonztruovat principiální protějšek lidského mozku a naučila tento umělý mozek řešit dokonce i problémy, které leží mimo lidské možnosti!

Tuto kapitolu tedy považujme jen za motivační úvod k dalšímu studiu velmi rozsáhlé oblasti umělé inteligence. Přitom lze rozhodně doporučit přečtení titulu [Nil82]. Na polském trhu je k dispozici také zajímavá práce [BC89], která se zabývá vyhledávacími metodami, jež hrají v oblasti umělé inteligence zásadní roli. Za přečtení stojí také kniha [Kas03], která se neomezuje na tradiční popisy problémů umělé inteligence, už jen díky obsáhlému popisu historického rozvoje zkoumání umělé inteligence, kde najdeme mnoho zajímavých odkazů na bohatou oborovou bibliografi.

¹ Současná věda kromě toho zatím dokonale nerozumí tomu, na jakém principu lidský mozek funguje.

Přehled oblastí zájmu umělé inteligence

Umělá inteligence (označuje se zkratkou *AI* z anglického „*artificial intelligence*“) je vědní oblast, jejímž cílem je zkoumat problémy téměř filozofické povahy, které souvisejí s věčným hledáním odpovědí na otázku z titulu této kapitoly: Mohou počítače myslit? Alan Turing v roce 1950² navrhl test, který měl na tuto otázku odpovědět. V systému založeném na dialogu vystupuje počítač *jako jeden z hráčů*. V situaci, kdy odpovědi hráče-počítače nebude možné rozpoznat od odpovědí hráče-člověka, můžeme počítači přiznat inteligenci. Tento test se dokonce dočkal ceny ve výši 100 tis. dolarů pro tvůrce prvního počítače, jehož odpovědi nebude možné rozlišit od odpovědí člověka (Loebnerova cena, kterou některé autority v oboru *AI* zpochybňují).

Kdyby však počítače dokázaly myšlení třeba jen napodobovat, nemohli bychom snad využít jejich technických možností (rychlosť operací, neúnavnost atp.) při řešení typicky lidských problémů?

Praktické cíle *AI* souvisejí právě se snahou zapojit počítače do řešení úkolů, pro které neexistují jednoduché klasické algoritmy, např.:

- automatické dokazování výroků,
- přijímání rozhodnutí na základě souboru kritérií,
- inteligentní vyhledávání a asociování informací ve velkých datových množinách,
- rozpoznávání obrazu (a také zvuků či dotykových podnětů),
- konstrukce samostatných automatů (např. projekt automatického řidiče, který by dokázal nahradit člověka za volantem)³,
- hry (např. šachy, dáma, reversi, go atd.),
- plánování.

Počítače mají tedy mnoho výhod, které chceme využít, ale i nadále hodně zaostávají za výkonem lidského mozku, jeho mimořádnou kapacitou (a nejde jen o ni, ale také o rychlosť přístupu k uloženým informacím) a především za jeho možnostmi učení, nemluvě již o schopnostech regenerace⁴!

Hrami se budeme podrobněji zabývat dále, protože se jedná o velmi atraktivní oblast, kde lze dosáhnout působivých výsledků. V dalších částech kapitoly pojednáme o dvou oblastech *AI*, které se vzhledem k rozsahu problematiky do této knihy nevešly, ale stojí za to se o nich alespoň zmínit. Jedná se o expertní systémy a neuronové sítě.

Expertní systémy

Expertní systém je počítačový program, který dokáže přijímat komplikovaná rozhodnutí nebo odpovídat na složité otázky a přitom umí poskytované odpovědi zdůvodnit. Systém je obvykle založen

² V článku „Computing Machinery and Intelligence“ (Výpočetní stroje a inteligence), který publikoval v časopise *Mind*.

³ Agentura DARPA, která spadá pod americké ministerstvo obrany, vyhlásila roku 2004 konkurs „DARPA Grand Challenge“ pro zájemce o účast v závodě vozidel, které se pohybují bez lidské účasti. Při třetím opakování konkursu v roce 2007 se objevila vozidla, která se dokáží pohybovat po městě a dodržují pravidla silničního provozu (u nás by taková auta neměla příliš velkou šanci projet povinných 60 mil bez nehody).

⁴ Poškozený lidský mozek není zcela zbytečný a v mnoha případech dokonce dokáže své původní funkce obnovit, čehož lze v případě počítačů zatím dosáhnout výhradně pomocí záložních systémů. V souvislosti s technologickým vývojem se možná časem objeví počítače, které budou fungovat na poněkud jiných principech než v současnosti (logika diskrétních systémů) a kde bude snadnější regenerovat poškozenou část systému na základě informací obsažených ve funkční části.

na tzv. znalostní bázi⁵, což je sada pravidel zapsaných formou implikace. Znalostní báze se tvoří na základě znalostí lidských expertů, ačkoli v některých případech mohou být generovány i automaticky. Odpovědi systému a jejich důvody vycházejí z logických pravidel, která jsou systému známa. Uvedme příklad pravidel v expertním systému:

- **if A then B**
- **if B then C**
- **if C then D**
- **if (A and F) then K**
- atp.

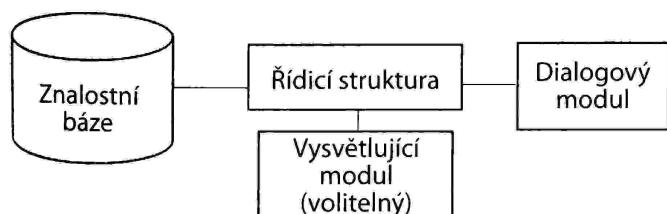
Pravidla sestavená tímto způsobem lze vyjádřit např. pomocí stromů nebo polí. Vnitřní reprezentace znalostní báze značně závisí na programovacím jazyce. Když máme znalostní bázi, potřebujeme ještě interpret, který ji dokáže prohledávat a poskytovat fakta nebo závěry. Takový modul označuje jako řídicí strukturu⁶. Inovativní přístup uplatněný v expertních systémech spočívá v důsledném oddělení kódu systému (logika, zásady zpracování pravidel) od znalostní báze, kterou lze po vytvoření systému kdykoli doplňovat novými externími daty. Jako expertní systém lze tedy označit také techniku konstrukce informačních systémů.

Expertní systém by měl co nejlépe napodobovat lidského expertsa a měl by se vyznačovat uživatelsky přívětivou interakcí. Tvůrci takových systémů věří, že znalosti vysoko kvalifikovaných expertů lze zapsat do programu takovým způsobem, aby z nich tento program dokázal vyvozovat závěry a generovat výsledky shodné s těmi, jaké by poskytl expert. Živý expert bývá velmi drahý a nemusí mít právě čas. Vybudujeme-li systém, který jej simuluje, můžeme výrazně snížit náklady na konzultace a usnadnit přístup ke znalostem.

Příkladem expertního systému může být „umělý lékař“, tj. program, který využívá znalosti lékařského specialisty (např. o příznacích nemoci) a na základě zadané otázky a představených faktů dokáže určit diagnózu. Za první expertní systém se obecně považuje MYCIN (Shortliffe, 1972). Tento program na analýzu krve uměl doporučit vhodná antibiotika. Do stejné třídy patří i jiné diagnostické systémy. Z hlediska počítače totiž není příliš důležité, zda onemocněl člověk či zda se porouchal automobil.

Jiné příklady: Systém profesního vzdělávání, systémy na výuku postižených dětí (program se snaží přizpůsobit způsob výuky tomu, jak rychle si dítě dokáže vědomosti osvojovat), předpovědi počasí, prognózy vývoje nemoci u pacienta, plánování prací (např. pohybů robota).

Kvalita expertního systému úzce závisí na prvcích tohoto systému (viz obrázek 12.1).



Obrázek 12.1: Schéma expertního systému

⁵ Ang. *knowledge database*.

⁶ Ang. *inference engine*.

KAPITOLA 12 Mohou počítače myslit?

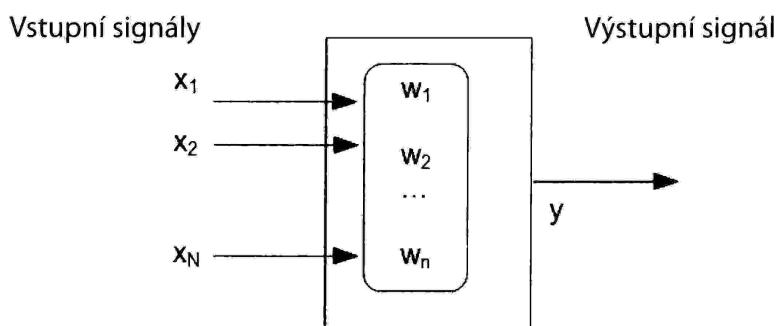
Rozlišujeme tyto moduly:

- Modul *znalostní báze* a moduly její aktualizace – modul zajišťující počítačovou reprezentaci oborových znalostí. Využívá se zápis tvrzení, výpočet vět a je-li to možné, také výpočetní matematické modely. V případě specializovaných systémů (např. medicína, právo, bankovnictví) je pojem „expert“ poněkud zavádějící, protože znalosti v systému nemusí vždy pocházet přímo od lidí (nejsou získány dotazováním odborníků).
- *Řídící struktura* – modul, který řídí řešení problému v závislosti na obsahu znalostní báze. Měl by se vyznačovat zejména rychlosí a kromě toho by měl zvládat filtrování znalostí, umožňovat návraty ze slepých uliček při odvozování, odstraňovat konflikty atp. Jedná se o nejdůležitější prvek kvalitního *expertního systému*.
- *Dialogový modul* (rozhraní) systému – modul odpovědný za komunikaci s vnějším světem – klade otázky a vysvětluje vlastní diagnózy. Měl by komunikovat jazykem, který je blízký přirozenému.

Expertní systémy akademické třídy se často budují v jazycích, které se k tomuto účelu zvláště hodí (Lisp, Prolog), avšak mezi produkty počítačových firem se můžeme setkat s nejrůznějšími hybridy, které nevynechávají ani nám dobře známý jazyk C++. Tyto systémy se osvědčují v případech, kdy jsou shromážděny a zapsány rozsáhlé empirické znalosti, které si jeden člověk obvykle jen těžko dokáže zapamatovat. Jejich vadou je jistá mechaničnost: obvykle nemají schopnost se učit ani objevovat nové znalosti a modifikace znalostní báze může mít nepředvídatelné důsledky. Značný problém představuje rozšiřování a aktualizace znalostní báze (ideální je, pokud existují katalogy nebo sestavy, které lze do znalostní báze přímo zahrnout, samozřejmě po určitých úpravách, aby se jejich struktura přizpůsobila reprezentaci modulu znalostní báze) a ověřování informací obsažených ve znalostní bázi, protože nemusí být spolehlivé.

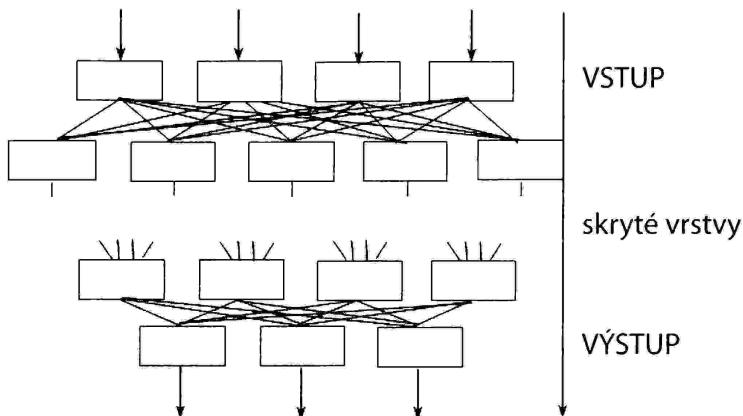
Neuronové sítě

Neuronové sítě jsou programovým nebo hardwarovým modelem fungování lidského mozku. Aniž bychom zabíhali do biologických podrobností, lidský mozek se skládá z nervových buněk zvaných neurony, které přijímají data (elektrochemické signály z jiných neuronů a receptorů) pomocí svých dendritů a předávají je dále prostřednictvím axonů se synapsemi. Odborníky napadlo, že bychom mohli napodobit fenomenální možnosti mozku simulací činnosti neuronů a vytvořením neuronové sítě, která by odpovídala když ne celému mozku, pak alespoň jeho funkční náhražce se schopností učení a řešení problémů. Obrázek 12.2 představuje informatický model neuronu jako systému s mnoha vstupy a jedním výstupem. Závislost mezi vstupními a výstupními daty lze interpretovat jako řešení nějakého problému, např. rozpoznání obrazu nebo klasifikace dat.



Obrázek 12.2: Model neuronu

Neuron zpracovává vstupní signály podle jisté vnitřní funkce (lineární či nelineární), která závisí na vahách w_1, w_2, \dots, w_n . Neuronová síť spojuje mnoho jednotlivých neuronů systémem „každý s každým“. Zvenčí samozřejmě vypadá jako černá skřínka se vstupy, na které přivádíme vstupní signály, a jedním či více výstupy. Příklad sítě vidíme na obrázku 12.3.



Obrázek 12.3: Neuronová síť

Síť může mít celkem libovolnou strukturu, může obsahovat jednu nebo více vrstev a případně také zpětné vazby (alespoň jeden výstup propojený se vstupem). K čemu mohou neuronové sítě sloužit? Uvedeme typické příklady jejich využití:

- klasifikace signálů (což jsou v podstatě data),
- rozpoznávání, předpovídání nebo filtrování signálů,
- simulace fungování smyslů (oko, ucho).

Jak ale mohou sítě fungovat? Již jsme zmínili, že výsledek zpracování daným neuronem závisí na vahách w_1, w_2, \dots, w_n . Chceme-li neuron naučit určité reakci (např. na optické podněty), musíme vhodně zvolit váhy a neuron bude provádět činnost, kterou od něj očekáváme.

Zde se však projevuje problém struktury sítě a její složitosti: neuronů mohou být tisíce, a navíc jsou vzájemně propojeny. Jak tedy v takových nezvládnutelných podmírkách vybírat nějaké váhy?!

Na uvedenou otázku zde přímo neodpovíme, protože o neuronových sítích lze napsat samostatnou publikaci (která se navíc neobejde bez pořádné porce matematiky, čemuž se v této knize snažíme pokud možno vyhýbat). V rámci tohoto maximálně zjednodušeného úvodu zmíníme pouze základní metodu učení sítě:

- Váhy losujeme během inicializace neuronové sítě.
- V dalších iteracích váhy korigujeme na základě porovnání očekávaného výsledku fungování sítě se skutečným výsledkem.
- Váhy upravujeme tím více, čím větší byla chyba odpovědi (odchylka od očekávaného výsledku).

V programech pro neuronové sítě plní roli učitele samozřejmě sada výukových algoritmů, kterým napomáhá báze dat (váhy). V procesu učení síť samostatně získává schopnost generalizace, tj. očekávaného reagování na vstupní data, která nebyla součástí výukové množiny. Z tohoto důvodu mluvíme o učení neuronové sítě, ačkoli to člověka může zpočátku zaskočit (jak se může učit něco, co nemá oči ani uši a nedokáže myslit?).

KAPITOLA 12 Mohou počítače myslit?

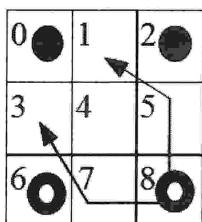
Reprezentace problémů

Při psaní „inteligentních“ programů je nejdůležitější vhodně *modelovat řešenou úlohu*. Píšeme-li například program, který dokáže hrát šachy, musíme si položit následující otázky:

- Jaký jazyk se pro náš účel nejlépe hodí?
- Pomocí kterých datových struktur je vhodné reprezentovat šachovnici a kameny?
- Které datové struktury umožní modelovat postup uvažování hráče? Tyto dotazy nejsou triviální a na příslušných odpovědích občas závisí, zda je daná úloha vůbec řešitelná!

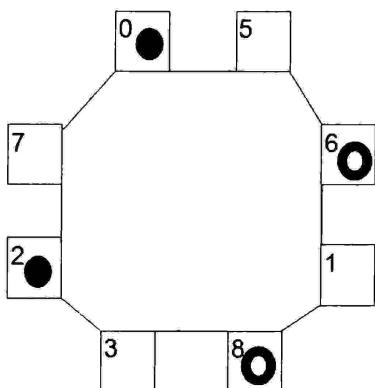
Příklad:

Disponujeme šachovnicí velikosti 3×3 pole, na které chceme střídat umístění bílých a černých koní (viz obrázek 12.4). Možné tahy koně, který se nachází na pozici s číslem 8, jsou znázorněny pomocí šipek.



Obrázek 12.4: Problém šachového koně (1)

Když úlohu reprezentujeme v podobě jako na tomto obrázku, vůbec si tím řešení neusnadníme, protože nevidíme jasně, jaké tahy jsou povoleny, ani cíl, kterého potřebujeme dosáhnout. Podívejme se na stejnou situaci, která je však zobrazena jiným způsobem (viz obrázek 12.5).



Obrázek 12.5: Problém šachového koně (2)

Budeme-li předpokládat, že se příslušný kůň může pohybovat pouze o dvě pole (dopředu i dozadu po vyznačené trase 0-5-6-1-8-3-2-7-0), můžeme tímto způsobem velmi snadno modelovat povolené tahy a dokážeme napsat funkci, která vytvoří seznam takových tahů pro určitého koně. Z obrázku rovněž vypadla mrtvá pozice (4), která není nijak dostupná, a proto ji vůbec nepotřebujeme.

Cvičení 1

Zamyslete se, jak vyřešit zadaný úkol, pokud bude povoleno *současně* táhnout několika figurami?

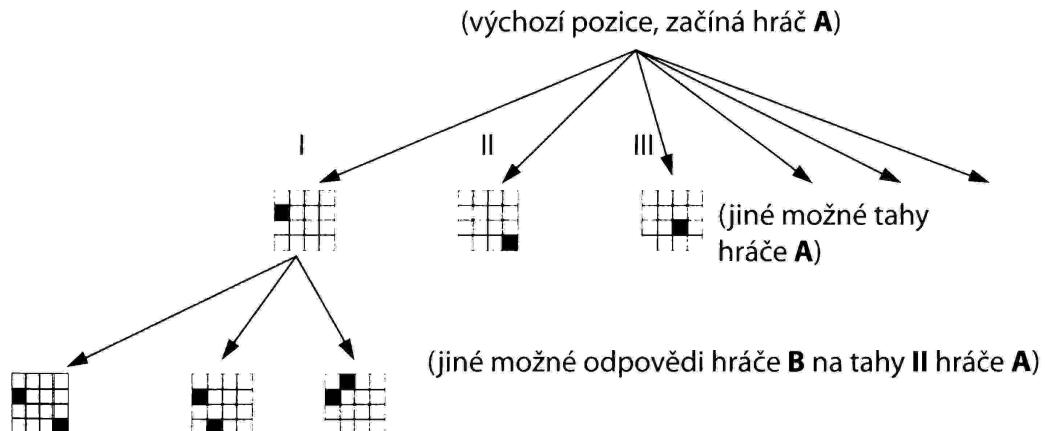
Při reprezentaci úloh umělé inteligence mají velký význam tzv. *stavové diagramy*, které ilustrují stavy problému pomocí uzlů (mohou to být např. šachovnice s postavením kamenů) a jejichž hrany znázorňují možnost přechodu jednoho stavu na jiný (třeba provedením tahu). V případě šachů bychom museli v uzlech ukládat aktuální pozice na šachovnice, takže by celá reprezentace byla poměrně nákladná, zohledníme-li počet možných situací a s tím související rozměry grafu.

Hry pro dvě osoby a stromy her

Výhodou typických her pro dvě osoby je jejich relativně snadná programová implementace. Podílí se na tom následující vlastnosti:

- V dané fázi máme kompletní informace o situaci, v jaké se hra nachází (stav hrací desky).
 - Role hráčů jsou *symetrické*.
 - Pravidla hry jsou známa předem.

V případě her pro dvě osoby je velmi výhodné použít *stromovou* datovou strukturu, která usnadňuje reprezentaci stavů a průběhu hry. Střídající se tahy obou hráčů jsou znázorněny pomocí uzlů stromu, jehož jednotlivá patra (úrovne hloubky) odpovídají všem přípustným tahům daného hráče. Příklad stromu je znázorněn na obrázku 12.6.



Obrázek 12.6: Příklad stromu myšlené hry

Jednotlivé uzly mají samy o sobě dosti komplikovanou strukturu, která umožnuje zapsat úplný stav hrací plochy (v tomto případě se jedná o šachovnici 4×4 pole, popisovaná hra je přitom zcela fiktivní). Hráč A, který hru začíná, má největší volnost tahů. Pokud vybere tah I, musí se hráč B jeho volbě přizpůsobit. Přitom uplatňuje dvě kritéria:

- Výběr musí být nejvýhodnější pro hráče B (kritérium *zdravého rozumu*).
 - Výběr musí odpovídat pravidlům hry (kritérium *správnosti*).

Strom hry je tím jednodušší, čím méně tahů hra umožňuje. Snadno tedy můžeme odvodit, že strom hry „piškvorky“ je mnohem prostší než strom dámky či šachů.

Všechny stromy hry v určitém okamžiku končí (i v případě, že jsou velmi rozsáhlé): každá rozumná hra přece dříve nebo později vede k výhře či prohře jedné ze stran, případně k remíze. Můžeme

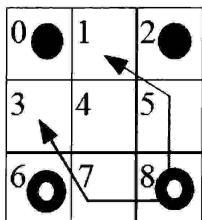
Reprezentace problémů

Při psaní „inteligentních“ programů je nejdůležitější vhodně *modelovat řešenou úlohu*. Píšeme-li například program, který dokáže hrát šachy, musíme si položit následující otázky:

- Jaký jazyk se pro náš účel nejlépe hodí?
- Pomocí kterých datových struktur je vhodné reprezentovat šachovnici a kameny?
- Které datové struktury umožní modelovat postup uvažování hráče? Tyto dotazy nejsou triviální a na příslušných odpovědích občas závisí, zda je daná úloha vůbec řešitelná!

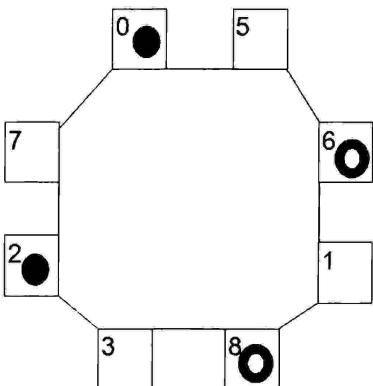
Příklad:

Disponujeme šachovnicí velikosti 3×3 pole, na které chceme střídat umístění bílých a černých koní (viz obrázek 12.4). Možné tahy koně, který se nachází na pozici s číslem 8, jsou znázorněny pomocí šipek.



Obrázek 12.4: Problém šachového koně (1)

Když úlohu reprezentujeme v podobě jako na tomto obrázku, vůbec si tím řešení neusnadníme, protože nevidíme jasně, jaké tahy jsou povoleny, ani cíl, kterého potřebujeme dosáhnout. Podívejme se na stejnou situaci, která je však zobrazena jiným způsobem (viz obrázek 12.5).



Obrázek 12.5: Problém šachového koně (2)

Budeme-li předpokládat, že se příslušný kůň může pohybovat pouze o dvě pole (dopředu i dozadu po vyznačené trase 0-5-6-1-8-3-2-7-0), můžeme tímto způsobem velmi snadno modelovat povolené tahy a dokážeme napsat funkci, která vytvoří seznam takových tahů pro určitého koně. Z obrázku rovněž vypadla mrtvá pozice (4), která není nijak dostupná, a proto ji vůbec nepotřebujeme.

Cvičení 1

Zamyslete se, jak vyřešit zadaný úkol, pokud bude povoleno současně táhnout několika figurami?

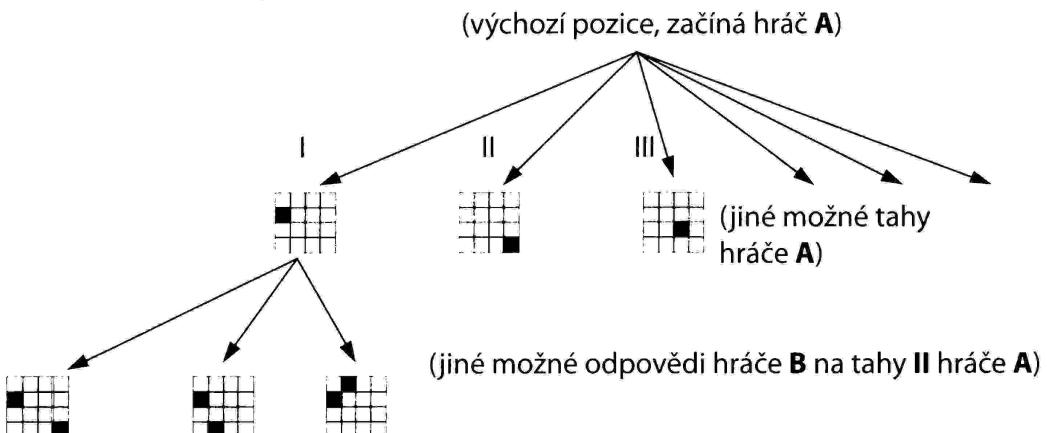
Při reprezentaci úloh umělé inteligence mají velký význam tzv. *stavové diagramy*, které ilustrují stavy problému pomocí uzlů (mohou to být např. šachovnice s postavením kamenů) a jejichž hrany znázorňují možnost přechodu jednoho stavu na jiný (třeba provedením tahu). V případě šachů bychom museli v uzlech ukládat aktuální pozice na šachovnice, takže by celá reprezentace byla poměrně nákladná, zohledníme-li počet možných situací a s tím související rozměry grafu.

Hry pro dvě osoby a stromy her

Výhodou typických her pro dvě osoby je jejich relativně snadná programová implementace. Podílí se na tom následující vlastnosti:

- V dané fázi máme kompletní informace o situaci, v jaké se hra nachází (stav hrací desky).
 - Role hráčů jsou *symetrické*.
 - Pravidla hry jsou známa předem.

V případě her pro dvě osoby je velmi výhodné použít *stromovou* datovou strukturu, která usnadňuje reprezentaci stavů a průběhu hry. Střídající se tahy obou hráčů jsou znázorněny pomocí uzlů stromu, jehož jednotlivá patra (úrovně hloubky) odpovídají všem přípustným tahům daného hráče. Příklad stromu je znázorněn na obrázku 12.6.



Obrázek 12.6: Příklad stromu myšlené hry

Jednotlivé uzly mají samy o sobě dosti komplikovanou strukturu, která umožňuje zapsat úplný stav hrací plochy (v tomto případě se jedná o šachovnici 4×4 pole, popisovaná hra je přitom zcela fiktivní). Hráč A, který hru začíná, má největší volnost tahů. Pokud vybere tah I, musí se hráč B jeho volbě přizpůsobit. Přitom uplatňuje dvě kritéria:

- Výběr musí být nejvhodnější pro hráče B (kritérium *zdravého rozumu*).
 - Výběr musí odpovídat pravidlům hry (kritérium *správnosti*).

Strom hry je tím jednodušší, čím méně tahů hra umožňuje. Snadno tedy můžeme odvodit, že strom hry „piškvorky“ je mnohem prostší než strom dámky či šachů.

Všechny stromy hry v určitém okamžiku končí (i v případě, že jsou velmi rozsáhlé): každá rozumná hra přece dříve nebo později vede k výhře či prohře jedné ze stran, případně k remíze. Můžeme

KAPITOLA 12 Mohou počítače myslit?

si tedy položit praktickou otázku: Lze vést průběh partie tak, abychom jednomu z hráčů navrhli výherní strategii? Má-li počítač „uvažovat“ v kategoriích výherní či prohrávající strategie, musíme mu poskytnout algoritmus, který bude efektivně simulovat schopnost inteligentního rozhodování. V praxi to znamená, že program musí obsahovat dva typy funkcí:

- **Hodnocení** – aktuální stav hry se hodnotí z hlediska převahy jedné ze stran a na tomto základě se generuje reálné číslo. Porovnání dvou stavů hry lze tedy převést na pouhé porovnání dvou čísel!
- **Rozhodování** – na základě hodnocení aktuálního stavu hry a případně několika následujících stavů (známých díky kompletně nebo částečně generovanému stromu hry) se přijímá rozhodnutí, který tah v dané fázi hry vybrat.

První funkce se principiálně tvoří dosti snadno, protože dokáže vyhodnotit sílu jedné ze stran. Samotná idea funkce sice není složitá, ale výběr jedné z možných funkcí občas bývá velmi obtížné matematicky zdůvodnit. Programy často využívají jistá intuitivní pozorování, která se těžko převádějí do matematického zápisu, i když bývají efektivní v praxi.

Funkce rozhodnutí se snaží ve formě počítačového kódu vyjádřit něco, co jednoduše nazýváme herní strategií. Funkce rozhodnutí je triviální v situacích, kdy dokážeme rychle generovat celý strom hry a vyhodnotit cesty, kterými může daná partie probíhat. U většiny her, které v současnosti považujeme za intelektuálně zajímavé (např. šachy, reversi, go atd.), takový strom bohužel zatím vytvořit nedokážeme. Počítače jsou v určitých případech stále příliš pomalé, ačkoli při prohlížení ohromujících animací či fascinujících a propracovaných scenérií počítačových her na to občas zapomínáme. Zbývá nám generovat fragment stromu hry (do nějaké rozumné hloubky) a na jejím základě přijmout odpovídající rozhodnutí.

V oblasti umělé inteligence bylo dosaženo velkých úspěchů při hledání vhodných algoritmů, které se mnohdy označují pouze jako *strategie prohledávání*. Z nich jsou nejznámější: A^* , *mini-max*, algoritmus řezů $\alpha-\beta$ či *SSS**. Všechny uvedené algoritmy v této knize nedokážeme podrobně představit, protože se řídíme zásadou, že probírané úlohy jsou zpravidla ilustrovány pomocí hotových programů jazyka C++. Aby to však vůbec mělo smysl, příslušné výpisby zabraly příliš mnoho místa. Proto se omezíme výhradně na podrobný výklad o algoritmu *mini-max* na příkladu hry „piškvorky“, který lze snadno naprogramovat. Avšak i takto jednoduchá hra vyžaduje poměrně dost řádků kódu. Abychom tedy zbytečně nezaplňovali stránky knihy, popíšeme pouze klíčová místa příslušného programu. Máte-li dostatek volného času, měli byste být schopni na tomto základě napsat program pro libovolnoujinou hru pro dvě osoby. Cílem následující prezentace bude výhradně vysvětlit využité mechanizmy. (Plnou verzi hry najeznete v archivu ke stažení jako soubor `tictac.cpp`.)

Kompletní popisy algoritmů, které jsme vynechali (i když jsou v praxi velmi důležité), jsou k dispozici například v publikaci [BC89]. V uvedené knize nejsou algoritmy představeny v nějakém konkrétním programovacím jazyce, ale zkušenému programátorovi by to nemělo nijak vadit.

Algoritmus mini-max

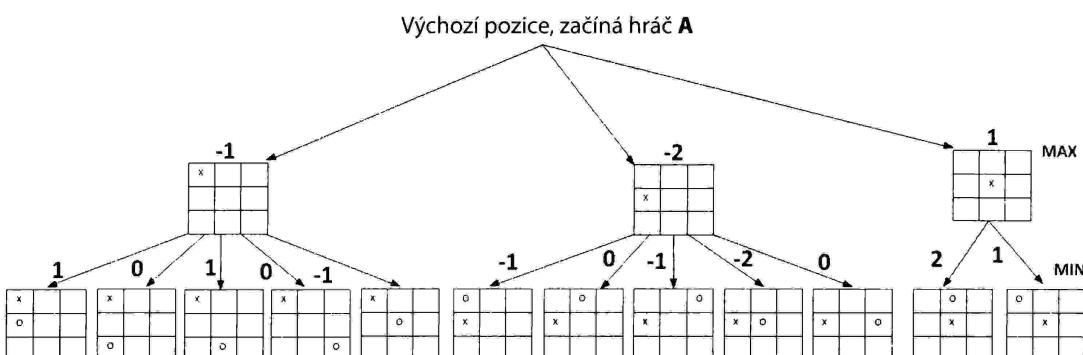
Vycházíme z počáteční pozice (stavu hry) a hledáme nejlepší možný tah. Máme dva typy uzlů: „max“ a „min“. Algoritmus předpokládá, že protivník postavený před několik voleb by provedl ze svého pohledu nejlepší tah (čili nejhorší pro nás). Naším cílem bude tedy najít tah, který nám maximalizuje hodnotu pozice, po které protivník provedl svůj optimální tah (čili táhneme tak, abychom z jeho hlediska hodnotu minimalizovali). Tímto způsobem zkoumáme jistý počet úrov-

ní⁷ (je zřejmé, že máme co do činění se stromem analýzy) a hodnoty z posledních úrovní přitom „přenášíme“ vzhůru podle pravidel algoritmu *mini-max*.

Jednoduchý příklad znázorněný na obrázku 12.7 ilustruje, jakým způsobem lze vybrat nejlepší první tah. Číselné hodnoty reprezentují útočnou sílu dané pozice.

Myšlenka algoritmu *mini-max* spočívá v tom, že systematicky propaguje datové hodnoty pozic úplně odspodu až k vrcholu stromu. Pokud aktuální vrchol předka představuje tahy hráče A, pak je s tímto vrcholem spojeno maximum hodnot jeho následnických vrcholů.

V případě, že uzel reprezentuje protivníka (hráče B), bereme minimum těchto hodnot. Proč právě tak, a nikoli třeba opačně? Souvisí to s praktickým předpokladem o zdravém rozumu obou hráčů: B se snaží maximalizovat svou šanci na výhru, jinak řečeno minimalizovat tuto šanci pro hráče A. Pokud analýzu celého stromu v praxi nelze provést, algoritmy se zastavují v jisté vybrané hloubce – v našem příkladu se jedná o $h = 2$.



Obrázek 12.7: Pravidlo mini-max

Dále předpokládejme, že jsme získali číselné hodnoty uzlů z poslední úrovni pomocí jisté funkce hodnocení. V analyzovaném příkladu jsme vybrali uzel s hodnotou $1 = \max(-1, 2, 1)$. Pamatujme, že tento výběr záleží na hloubce analýzy stromu hry a při jiné hodnotě h by první tah mohl být úplně jiný!

Existuje upravená verze algoritmu *mini-max*, která umožňuje značně zkrátit čas analýzy. Eliminuje totiž zbytečná porovnání hodnot pocházejících z podstromů, které při propagaci podle pravidel algoritmu *mini-max* stejně nemají šanci na vynesení nahoru. Tato verze se obecně označuje jako *algoritmus řezů α-β*. Například hodnota -1 vynesená nahoru na obrázku 12.7 (hodnotíme terminální uzly zleva doprava) naznačuje, že nemá smysl analyzovat ty části stromu, které by mohly vynést hodnotu nižší než -1 . Samozřejmě přitom využíváme matematické vlastnosti funkcí *min* a *max*.

Ukažme tedy konečně tu tajemnou proceduru *mini-max*. Kvůli jednodušší programové implementaci ji představíme formou pseudokódu⁸.

Algoritmus prohledávání stromu hry s využitím pravidla *mini-max* má následující formu⁹:

```
Minimax(uzel w)
{
```

7 Jejich počet je volitelný a určuje hloubku zanoření procedury *mini-max*. U některých her samozřejmě příliš hluboké prohledávání nemá valný smysl: jsou totiž příliš „mělké“.

8 Měli bychom si uvědomit, že konkrétní implementace procedury *mini-max* se v závislosti na konkrétní hře a způsobu její reprezentace může od naší ukázky zásadně lišit.

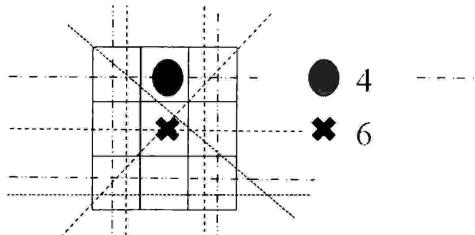
9 Verze je orientována na vítězství hráče MAX.

KAPITOLA 12 Mohou počítače myslit?

```
jestliže w je typu MAX pak v = - ∞
jestliže w je typu MIN pak v = + ∞
jestliže w je terminální uzel pak
    vrát hodnocení(w)
p1, p2, ... pk = generuj(w) // potomci uzlu w
pro j=1...k proved
{
    pokud w je typu MAX pak
        v=max(v, mimimax(pk))
    v opačném případě
        v=min(v, (v, mimimax(pk)))
}
vrát v
```

Zatím jsme se vyhnuli funkci hodnocení. Důvod je poměrně prozaický: tato funkce úzce souvisí s implementovanou hrou a nemá smysl ji popisovat mimo příslušný kontext.

Podle čeho poznáme sílu své pozice v dané fázi hry „piškvorky“? Můžeme vymyslet hodně podivných kritérií, ale v literatuře se často uvádí jedno z nich. Využijeme princip *otevřených linií* pro daného hráče, což jsou takové linie, které protivník neblokuje. Dávají nám tedy šanci, abychom sestavili plnou řadu, která je cílem hry. Popsanou zásadu znázorňuje obrázek 12.8. Hodnota tohoto čísla je snížena o počet otevřených linií protivníka.



Obrázek 12.8: Pojem otevřených linií ve hře „piškvorky“

Obrázek mimochodem naznačuje, pomocí které struktury dat lze ukládat stavy hry. Je to běžné pole `int t[9]`, jehož indexy odpovídají pozicím v mřížce na obrázku 12.8. Chceme-li zvýšit srozumitelnost programu, můžeme místo hodnot typu `int` použít výčtový typ¹⁰:

```
enum KDO{nic, pocitac, clovek};
```



Poznámka: Vzhledem ke své značné délce nejsou výpisy v této kapitole přetištěny kompletně. Úplný kód se nachází v souboru `tictac.cpp`, který je k dispozici v archivu ke stažení.

Hodnoty daného pole mřížky by tedy nebyly uloženy pomocí proměnných typu `int`, ale typu `KDO`, ačkoliv znalci jazyků C/C++ vědí, že vnitřně se také jedná o typ `int`.

Funkce `hodnocení` dostává jako parametr mřížku a informaci o tom, pro koho má provést výpočet. Problém s hodnotami kladného (záporného) nekonečna můžeme v programu C++ vyřešit tím, že vybereme čísla, která jsou značně větší než hodnoty vrácené funkci `hodnocení`, např.:

¹⁰ Konstantám `pocitac` a `clovek` odpovídají v mřížce znaky kolečko a křížek.

```
const plus_nekonecno = 1000;
const minus_nekonecno = -1000;
```

V průběhu hry se hráči střídají, a proto je vhodné vytvořit funkci, která bude informovat o tom, kdo má hrát:

```
KDO Další_Hrac(KDO hrac)

{
    if (hrac==pocitac)
        return clovek;
    else
        return pocitac;
}
```

Uplatní se rovněž pomocné funkce (nepříliš složitý kód vynecháme):

```
void ZobrazMrizku(mrizka);
void NulujeMrizku(mrizka);
int KonecHry(mrizka);
```

Funkce KonecHry kontroluje, zda některý hráč nevytvořil řadu ze třech stejných znaků, což – jak si vzpomínáme – zajišťuje v této hře vítězství, či zda nenastala remízová situace.

Kód samotné hry je tvořen běžným cyklem, který zajišťuje, že se hráči střídají na tahu. Řekněme, že tento cyklus je součástí funkce Hraj:

```
tictac.cpp
void Hraj(mrizka, hrac)
{
    hrac_tmp=hrac;
    while (!KontrolujKonecHry(mrizka,hrac_tmp))
    {
        ZobrazMrizku(mrizka);
        tah=VyberTah(hrac_tmp, mrizka);
        ProvedTah(hrac_tmp, mrizka, tah);
        hrac_tmp=Dalsi_Hrac(hrac_tmp);
    }
}
```

Uvedené schéma platí pro většinu her pro dvě osoby. Úplně na začátku musíme určit, kdo začíná (počítač či člověk), např. náhodným výběrem. Toto losování by mělo jednou proběhnout ve funkci main, která pak po vynulování herní plochy vyvolá proceduru Hraj. Cyklus pokračuje, dokud se hra nedostane do stavu, který odpovídá vítězství jednoho hráče nebo remíze. Procedura ProvedTah úzce závisí na využitých datových strukturách. V tomto případě to může být jednoduše:

```
hra[tah]=hrac_tmp;
```

O něco těžší je táhnout v mnohem komplikovanější hře, jako jsou šachy nebo reversi. Musíme totiž zohlednit i vedlejší efekty jako výměna pěšců, rošády atp.

Jak se ale můžeme rozhodnout, který tah provést? Odpovědi by měla poskytovat funkce VyberTah, která využívá dříve představenou funkci mini-max:

```
int VyberTah(hrac, mrizka)
{// výběr tahu závisí na tom, kdo hráje
```

KAPITOLA 12 Mohou počítače myslit?

```

if (hrac==clovek)
    do{
        cout << "Tvůj výběr(0..8):";
        cin >> tah;
    }while(!Obsazeno(mrizka, tah));
else
{
    cout << "Tah počítače:\n";
    tah=MiniMax(mrizka,hrac);
    return tah;
}

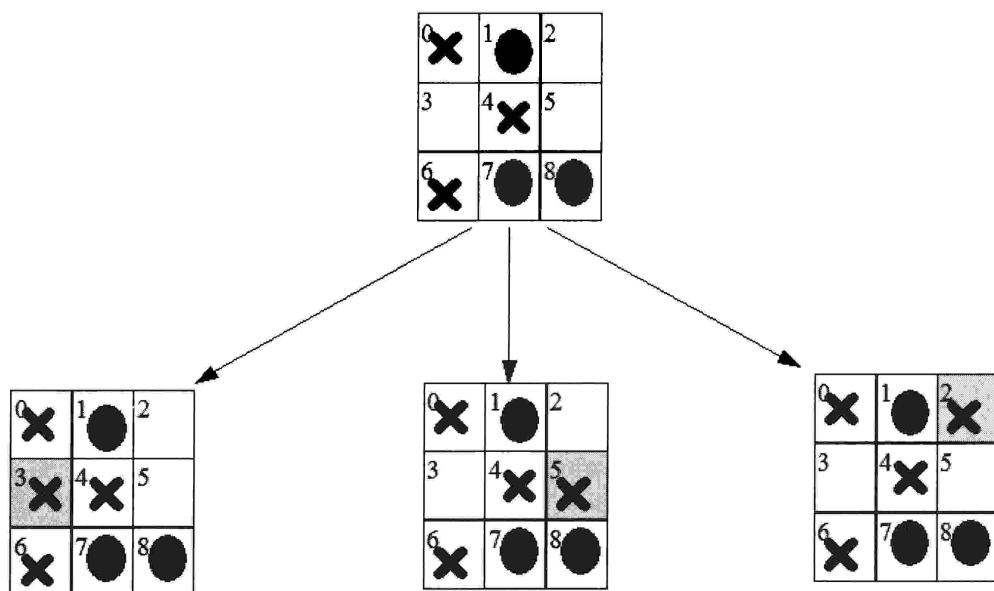
```

Obsah procedury `MiniMax`, která se nachází v souboru `tictac.cpp`, přesně odráží výše uvedený algoritmus, samozřejmě s ohledem na příslušnou datovou strukturu dané hry.

Problémy může na první pohled představovat generování následníků příslušného uzlu. Obrázek 12.9 představuje výsledek funkce generuj pro jistý uzel w (předpokládáme, že táhl hráč, který kreslí krížky).

Na tomto místě si můžeme vzpomenout na nějaké seznamy, stromy, množiny atp. Podívejme se však, jak lze elegantně zakódovat seznam potomků daného uzlu pouze díky jedné pomocné mřížce (viz obrázek 12.10). Stačí přijmout konvenci, že zapsání jiné hodnoty než minus jedna označuje jeden vygenerovaný uzel: tímto způsobem dokážeme do jedné mřížky umístit celý seznam možných tahů!

Tím můžeme pojednání o technických aspektech programování her pro dvě osoby zakončit. Čtenáři, které tato tematika více zajímá a chtěli by vytvořit například vlastní šachový program, mohou své znalosti rozšířit pomocí specializované literatury.



Obrázek 12.9: Generování seznamu možných tahů hráče na základě daného uzlu

0 -1	1 -1	2 1
3 1	4 -1	5 1
6 -1	7 -1	8 -1

Obrázek 12.10: Kódování seznamu následných uzelů s použitím jediného uzlu

Algoritmus *mini-max* je ve své základní formě dosti pomalý a v praxi se často nahrazuje procedurou *řezů α-β*. Kromě toho se při programování určitých her vzhledem ke složité obsluze datových struktur nehodí všechny prohledávací algoritmy. Dobré algoritmy vyhledávání vhodné herní strategie bohužel bývají značně komplikované. Programátoři začínají stále častěji využívat rychlost moderních počítačů. Tím se zjednoduší samotný proces psaní kódu, protože lze uplatnit nejjednodušší vyhledávací algoritmy typu *hrubou silou*. Tímto způsobem v roce 1996 postupovali autoři softwaru pro počítač, který měl porazit samotného šachového velmistra Kasparova¹¹. V tehdejším turnaji si převahu udržel člověk, ale již o rok později (11. května 1997) konstruktéři přišli s ještě silnějším počítacem (s 256 procesory¹²) – a tenkrát Kasparov prohrál.

11 Počítač v zadaném čase generoval co největší počet možných strategií, počítal jejich sílu (pomocí funkce hodnocení) a vybíral ty lokálně nejlepší.

12 V době psaní této knihy fungují superpočítače, které jsou postaveny z více než 200 000 procesorů (projekt Blue Gene/L firmy IBM).

KAPITOLA 13

Kódování a komprese dat

V této kapitole:

- Kódování dat a aritmetika velkých čísel
- Techniky komprese dat

Při psaní prvního vydání této knihy si Internet – tato nejpopulárnější globální počítačová síť – získával oblibu hlavně v akademických institucích, protože poskytoval efektivní přístup k informacím na celém světě a pomáhal rozvíjet odborné kontakty. Při práci na třetím vydání je již zřejmé, že Internet proniká téměř všude: většina lidí si už nedokáže představit nákup počítače bez síťové karty, dřívější přístup k Internetu pomocí telefonní linky a modemu se postupně nahrazuje stálými přípojkami, které vedou až do domácností (jedná se o širokopásmový přístup založený na technologii ADSL¹ nebo na kabelových sítích).

Výhod Internetu využívá stále více lidí, kteří zpravidla nejsou specialisty na počítače a informatiku. Uživatelé si zvykají, že mohou pomocí snadno použitelných grafických prohlížečů (např. Internet Explorer, Opera, Firefox) procházet webové stránky a vyhledávat důležitá data či trávit čas zábavou. Mnozí z nás si už svůj život bez Internetu nedokáží představit.

Díky jednoduchému uživatelskému rozhraní programů, které umožňují pracovat se síťovými zdroji, jsou běžní uživatelé odstíněni od problémů, s nimiž si musí komunikační programy poradit. Když ze začátku hlavní problém spočíval v nízké propustnosti linek, měla klíčový význam *kompresce* přenášených dat neboli takové jejich kódování², které by umožnilo stejné množství informace přenést pomocí menšího počtu bitů. Později se hlavní důraz přenesl na bezpečnost dat, tzn. na jejich ochranu před nepovolaným přístupem zvenějšku (přenosy v Internetu lze pomocí vhodných programů velmi snadno odposlouchávat). V současnosti se pozornost opět soustřeďuje na kompresi, protože objem multimediálních dat (obraz, zvuk) přenášených v Internetu roste exponenciálním způsobem – lidé si zvykli posílat e-mailem velké přílohy s obrázkovými dokumenty a stahovat – často nelegálně – hudební a filmové soubory. Bez kódování dat by se v Internetu prakticky nedalo pracovat, dokonce ani s širokopásmovou přípojkou.

Týká se komprese dat pouze domény počítačů? Samozřejmě nikoli: uplatňuje se také například v mobilních telefonech, digitální satelitní televizi (při psaní této knihy mizí analogové satelitní kanály jeden za druhým a nahrazují je kódované digitální programy, které jsou pro mediální firmy levnější) nebo i v modemových a faxových přenosech.

1 Ang. *Asymmetric Digital Subscriber Line*.

2 Kódování – převod informací do podoby, která je vhodná k jejich přenosu, např. do formy řetězce nul a jedniček neboli prostě dvou elektrických signálů, které lze od sebe snadno odlišit (třeba změřením jejich amplitudy či frekvence).

KAPITOLA 13 Kódování a komprese dat

Jak lze kompresi dat obecně realizovat? Pro laika může proces komprese dat vypadat jako magie, ale už po zběžném prozkoumání se ukáže, že na tom není nic tajemného. Jako příklad uvedeme zprávu s délkou 50 znaků: „*SEJDEME SE POZÍTŘÍ O SEDMÉ NA LAVIČCE PŘED RADNICÍ*“. Budeme-li předpokládat nejjednodušší kódování pomocí osmibitového kódu ASCII (v němž na každý z 256 znaků připadá jistá osmibitová posloupnost nul a jedniček), můžeme odhadnout, že uvedená zpráva bude mít délku $50 \cdot 8 = 400$ bitů³. Musíme však v případě běžných textů českého jazyka používat nákladné osmibitové kódování? Česká abeceda přece obsahuje mnohem méně než $2^8 = 256$ znaků! Předpokládejme, že se u běžných textů omezíme na abecedu, kterou shrnuje tabulka 13.1:

Tabulka 13.1: Zjednodušená abeceda definovaná kvůli kódování

Znaky	Komentář	Počet znaků
„A“ ... „Z“	Základní znaky	26
„“	Mezera	1
,	Čárka	1
:	Středník	1
.	Tečka	1
-	Pomlčka (spojovník)	1
á, č, đ, é, ě, í, ň, ó, ř, š, ū, ú, ý, ž		15
Celkem:		46

K zakódování 46 znaků úplně stačí 6 bitů ($A = 00\ 0000$, $B = 00\ 0001$, $C = \dots$). Zpráva se tedy zkraje ze 400 na 300 bitů⁴.

Tabulkové kódy jsou sice poněkud primitivní, ale přesto mají ve výpočetní technice široké uplatnění. Znakové terminály například používají dosti prostý, ale efektivní kód ASCII (ang. *American Standard Code for Information Interchange*). Každý jeho znak zabírá jeden bajt, takže je možné zakódovat 256 různých znaků:

- 26 velkých a malých písmen latinské abecedy,
- číslice od 0 do 9, mezery,
- speciální znaky, např. %, !, *,
- řídicí symboly (kódy ASCII od 0 do 31), např. příkaz „přechod na další řádek“ neboli LF (z ang. *Line Feed*), znak konce textu atp.,
- kódy ASCII s hodnotami nad 127 patří do tzv. rozšířené sady, kde jsou zapsány znaky národních abeced a semigrafické znaky (symboly, které umožňují např. vytvářet rámečky na obrazovce).

Elektronické zařízení – zde alfanumerický terminál – přijímá posloupnosti bajtů a díky integrované elektronice dokáže zobrazit správný znak čitelný pro člověka.

Je zřejmé, že známe-li abecedu přenášených zpráv, můžeme je vhodnou volbou kódu značně zkrátit, aniž bychom ztratili část obsažených informací. Existuje hodně kódů, které jsou značně složitější

³ Pro zjednodušení zde nezohledňujeme žádné dodatečné bity, které souvisejí s kontrolou správnosti přenesených dat, ani technické podrobnosti konkrétního telekomunikačního protokolu – jinak řečeno, nacházíme se na *aplikační* úrovni.

⁴ Šetříme 25 % původní délky textu!

než primitivní tabulkové kódy typu ASCII. Nebudeme však z této kapitoly dělat příručku teorie kódování a informací. Aniž bychom zabíhali do podrobností, stojí za zmínu, že existují dvě základní skupiny kódů: *rovnoměrné* (se stálou délkou kódového slova) a *nerovnoměrné* (s proměnlivou délkou kódového slova). V obou případech lze zakódovanou informaci doplnit o dodatečné kontrolní bity, které usnadňují obnovení informace i v případě, že byla přenášená zpráva částečně poškozena (tímto způsobem získáváme tzv. *redundantní kódy*).

Tuto problematiku však nebudeme podrobněji rozebírat, protože souvisí spíše s přenosem signálů (fyzickými přenosy dat, kdy nezáleží na smyslu předávaných informací) než s informatikou v čisté formě (uživatelskými aplikacemi, kdy má smysl předávaných informací klíčový význam).

V další části kapitoly detailně popíšeme oblíbený systém šifrování s tzv. *veřejným klíčem* a *Huffmanovo* kódování, které představuje vynikající a srozumitelný příklad univerzálního algoritmu komprese dat. Vzhledem k tomu, že jsme v úvodu zmínili síť Internet, vysvětlíme také princip kódování *LZW*, které slouží ke kompresi souborů *GIF*. Jako obvykle přitom budeme vycházet z jednoduchých příkladů.

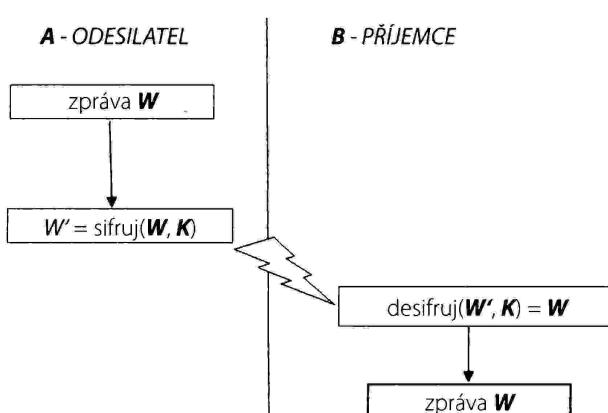
Čtenářům, kteří by si chtěli rozšířit své znalosti o komprezi dat, lze doporučit obsáhlou příručku [Say06], která je však napsána velmi srozumitelně. O rozsahu problematiky svědčí už tloušťka zmíněné publikace: zahrnuje více než 600 stran teorie a sám autor označuje svou knihu jako úvod do tématu.

Kódování dat a aritmetika velkých čísel

Kódování dat (nebo přesněji řečeno šifrování zpráv) se uplatní všude tam, kde z jistých důvodů chceme utajit obsah předávaných zpráv, aby se k jejich obsahu nedostaly nepovolané osoby, které by jej mohly zneužít. Šifrování se může týkat i osobní korespondence, ale v praxi se vzhledem k ekonomickým zájmům nejčastěji využívá v obchodních transakcích nejrůznějšího typu.

Symetrické šifrování

Lidé se o šifrování zajímali odedávna a usilovně se snažili vymyslet takové šifrovací algoritmy, které by bylo těžké prolomit v rozumném čase. Proces šifrování a dešifrování můžeme představit v podobě prostého schématu (viz obrázek 13.1).



Obrázek 13.1: Symetrické šifrování

KAPITOLA 13 Kódování a komprese dat

Odesilatel A zašifruje určitou zprávu W pomocí šifrovací procedury $sifruj$, která přijímá dva parametry: text k zašifrování a jistý dodatečný parametr K , který se označuje jako klíč. Prvek klíče K zvyšuje složitost obecně známého šifrovacího algoritmu a znemožňuje nepovolaným osobám, aby zprávu přečetly. Příjemce B dostane zašifrovanou (nečitelnou) zprávu W' , ale díky tomu, že má k dispozici dešifrovací proceduru a klíč K , zprávu W bez problémů otevře.

Jeden z nejprostších kódů připisuje písmenům abecedy čísla (řekněme, že se naše abeceda skládá z 46 znaků). Jedná se o běžné tabulkové kódování, které dokáže snadno prolomit každý znalec jazyka vybavený elektronickým „počitadlem“ a svými znalostmi. Jak tento obecně známý šifrovací algoritmus zkomplikovat? K přenášenému kódovému číslu můžeme například přičíst jistou hodnotu K , aby nebylo možné zprávu přečíst jednoduchým porovnáním s pozicí kódové tabulky. Příjemce B, který si chce zprávu přečíst, musí nejdříve od přijatých čísel odečíst hodnotu K . Tímto způsobem dostane kanonický tabulkový kód⁵.

Pozorný čtenář si na obrázku 13.1 jistě povšiml, že takový šifrovací systém se vyznačuje zásadní nevýhodou: odesilatel i příjemce musí znát hodnotu klíče K . Předávání klíče klasickými metodami (např. kurýrem) je velmi nepraktické a kromě toho je přitom ohrožena důvěrnost dat i ... bezpečí samotného kurýra.

Metoda popsaná v tomto odstavci se označuje jako *symetrické šifrování* – k šifrování i dešifrování dat se používá stejný klíč. Snadno dojdeme k závěru, že tato metoda není příliš bezpečná, protože k jejímu prolomení stačí krádež klíče nebo metody *zpětného inženýrství*. Symetrické algoritmy se však vyznačují velkou výhodou: jsou rychlé! Některé zajímavé algoritmy symetrického šifrování byly publikovány a staly se součástí protokolů a známých programů. Jako příklad můžeme uvést algoritmus *DES* (ang. *Data Encryption Standard*), což je bloková šifra, která šifruje data po částech s délkou 64 bitů (tj. 8 znaků ASCII obsahujících paritní bit).

Klíč algoritmu *DES* má délku 56 bitů, i když se zapisuje pomocí 64 bitů (každý osmý bit je paritní). V těle algoritmu probíhají cyklické permutace a směšování datových bloků. Změny původní posloupnosti bitů závisí na hodnotě podklíčů K_1, K_2, \dots, K_{16} které se generují na základě klíče zadaného uživatelem (K_0). Podklíče slouží k permutaci a směšování datových bloků v následných průchodech algoritmu. Při dekódování se používají stejně podklíče, jen v opačném pořadí.

Algoritmus *DES* se uplatňuje při šifrování příloh elektronické pošty. Kdysi se široce využíval v podnikové sféře (např. ve finančnictví). V současnosti se tam však místo něj setkáme spíše s jeho silnější verzí s názvem *3DES*, která pracuje se 168bitovým klíčem a lépe odolává útokům.

Algoritmus *DES* vybírá klíč pro každou zprávu náhodně z 72 000 000 000 000 000 (72 milionů miliard) možností. Obecně se proto předpokládalo, že zprávy šifrované algoritmem *DES* nelze prolomit. V roce 1997 však firma RSA (je držitelem patentu k jiné šifrovací metodě, kterou popíšeme dále) vypsalala za rozbití algoritmu *DES* odměnu ve výši 10 000 USD. Díky úsilí sestaveného týmu a volné kapacity tisíců počítačů připojených k Internetu se kód podařilo prolomit za necelé 3 měsíce. O deset let později pak organizace *Electronic Frontier Foundation* tento výsledek ještě vylepšila a text šifrovaný algoritmem *DES* přečetla za pouhých 9 dní.

Závěrem ještě dodejme, že existují hardwarové kodéry algoritmu *DES* (založené na integrovaných obvodech), které jsou v průměru tisíckrát rychlejší než jejich softwarové protějšky.

⁵ Jednodušší příklad šifry i klíče už bychom mohli sotva vymyslet. Žádná armáda by tímto způsobem nezašifrovala ani jídelní lístek, aby se nestala terčem posměchu svých protivníků.

Asymetrické šifrování

Asymetrické šifrování eliminuje nevýhody složité logistiky a nutnosti střežit klíč, kterými se vyznačují symetrické algoritmy. Efektivně řeší *problém přenosu klíče* ve světě, kde je důležité, aby příjemce zprávu dostal za zlomek sekundy a nemusel se přitom starat o věrohodnost použitého klíče K.

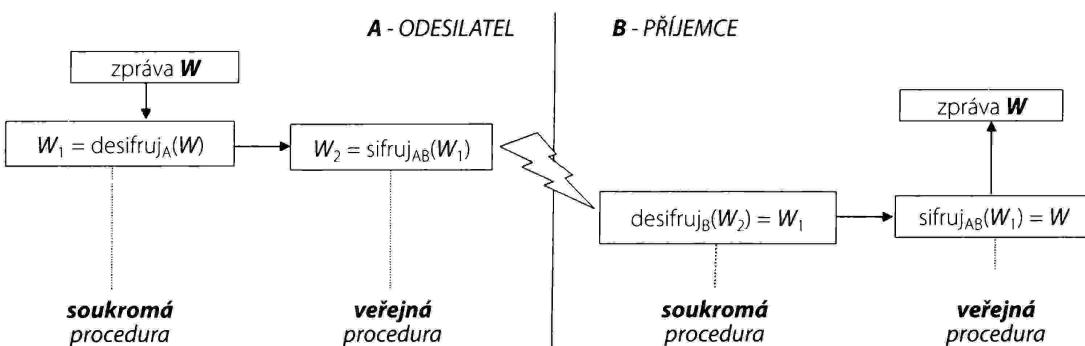
Vycházíme z toho, že příjemce může odesilateli předat svůj šifrovací klíč otevřeným způsobem. Odesilatel s ním zašifruje zprávu a poše ji příjemci. Ten zprávu rozšifruje druhým klíčem, kterým však disponuje výhradně on sám. Odposlechnutí šifrovacího klíče bez druhého (dešifrovacího) klíče útočníkovi nijak nepomůže!

První šifrovací klíč se nazývá *veřejný klíč* a druhý dešifrovací se označuje jako *soukromý klíč*. Soukromý klíč si každý musí hlídat jako oko v hlavě, ale veřejný klíč můžeme poskytnout každému, s nímž chceme komunikovat. Každý, kdo od nás dostal náš veřejný klíč, nám může poslat zprávu zašifrovanou tímto klíčem. Zprávu však dokážeme přečíst pouze my, protože nikdo jiný nemá náš soukromý klíč.

Metoda *šifrování s veřejným klíčem*, která zcela odstranila nutnost předávání sdíleného klíče, byla objevena v roce 1976. Jejími autory jsou W. Diffie a M. Hellman, ale s první praktickou realizací přišli R. Rivest, A. Shamir a L. Adleman. Výsledek jejich práce se proslavil jako šifrovací systém RSA. Metoda RSA zaručuje velmi vysoký stupeň zabezpečení přenášených dat. Vzhledem k tomu, že podle názoru matematiků ji nelze prolomit, v současnosti se o ni zajímají počítačoví nadšenci po celém světě, pro které je prolomení šifrovacího klíče záležitostí osobní prestiže⁶.

Než budeme systém RSA analyzovat na konkrétním číselném příkladu, pokusme se pochopit samotný princip šifrování s veřejným klíčem.

Šifrovací systém s veřejným klíčem je znázorněn na obrázku 13.2. Skládá se ze tří procedur: dvou *soukromých desifruj_A* a *desifruj_B* a *veřejné* procedury *sifruj_{AB}*.



Obrázek 13.2: Šifrovací systém s veřejným klíčem

Odesilatel A, který chce příjemci B poslat zprávu W, přitom začíná docela zvláštní operaci: místo toho, aby zprávu normálně zašifroval a přenosovým kanálem ji odesal příjemci, aplikuje nejdříve funkci *desifruj_A* na nezašifrovanou zprávu! Tato činnost, která může na první pohled vypadat absurdně, má svůj praktický důvod: ke zprávě W vytvoří neopakovatelný *digitální podpis* odesilatele A, což má

⁶ Autorská práva k algoritmu RSA vlastní společnost RSA Data Security Inc., která uděluje komerční licence na jeho použití v programech jiných dodavatelů (tuto licenci používají mj. prohlížeče Internet Explorer a Netscape Navigator). Algoritmus RSA je součástí mnoha sítových standardů a protokolů (např. S/MIME, SSL) a programů (mj. PGP k zabezpečení e-mailové komunikace).

KAPITOLA 13 Kódování a komprese dat

v mnoha systémech (např. bankovních) přímo strategický význam. Podepsaná zpráva (w_1) je poté zašifrována obecně známou šifrovací procedurou $sifruj_{AB}$ a teprve poté je odeslána příjemci B. Příjemce B dostane zašifrovanou sekvenci a zpracuje ji svou soukromou funkcí $desifruj_B$, jejímž výstupem je podepsaná zpráva w_1 . Speciálními vlastnostmi se musí vyznačovat i funkce $sifruj_{AB}$, která by z digitálně podepsané zprávy w_1 měla obnovit původní zprávu w .

Upozornění: Na základě bezpečnostních požadavků prakticky *nelze odvodit tajné dešifrovací procedury na základě veřejných šifrovacích procedur.*

Idea tedy vypadá lákavě. Musíme jen najít tři tajemné procedury, na které klademe dosti tvrdé požadavky. Teprve rok po zveřejnění principu systému šifrování s veřejným klíčem se objevila jeho první (a dosud také nejlepší) praktická realizace: šifrovací systém RSA. Tento systém vychází z toho, že příjemce B náhodně vybírá tři velmi velká prvočísla S, N_1 a N_2 (typicky se 100 číslicemi) a veřejně zpřístupní jen jejich součin⁷ $N = N_1 \cdot N_2$ a jisté číslo P , které splňuje podmínu:

$$P \cdot S \bmod (N_1 - 1) \cdot (N_2 - 1) = 1.$$

Bylo dokázáno, že pro každou šifrovou posloupnost M (text je nahrazen odpovídající číselnou posloupností určité konečné délky) platí rovnost: $M^P \bmod N = M$.

Šifrování lze tedy převést na výpočet rovnosti:

$$\{\text{šifrová posloupnost}\} = sifruj(M) = M^P \bmod N,$$

zatímco dešifrování odpovídá výpočtu:

$$M = desifruj(\{\text{šifrová posloupnost}\}) = \{\text{šifrová posloupnost}\}^S \bmod N.$$

Přes zdánlivě náročné zpracování velmi velkých čísel se ukazuje, že díky vlastnostem funkce \bmod naleží šifrová posloupnost i jeho zašifrovaná podoba do stejného rozsahu čísel. Systém RSA by se nám podařilo prolomit, pokud bychom na základě známých hodnot N a P uměli určit utajenou hodnotu S , bez níž nelze zprávu dešifrovat. Dosud však nebyl nalezen algoritmus, který by tento úkol dokázal splnit v rozumném čase.

Všechny šifrovací algoritmy musí řešit problém výpočtů s velmi velkými celými čísly. Ukazuje se, že tyto výpočty lze značně zjednodušit za podmínky, že je budeme považovat za koeficienty mnohočlenů. Jako příklad vezměme číslo:

$$12\ 9876\ 0002\ 6000\ 0000\ 0054$$

V systému o základu $x = 10$ je možné uvedené číslo vyjádřit jako:

$$x^{21} + 2x^{20} + (9x^{19} + 8x^{18} + 7x^{17} + 6x^{16}) + (2x^{12}) + (6x^{11}) + (5x^1 + 4).$$

Pokud nám hodnota $x = 10$ připadá příliš malá, dostaneme stejný výsledek, když dosadíme např. $x = 10\ 000$:

$$(12x^5) + (9876x^4) + (2x^3) + (6000x^2) + 54.$$

V důsledku: Budeme-li velká čísla interpretovat jako mnohočleny, můžeme všechny operace s těmito čísly nahradit algoritmy, které pracují s mnohočleny.

Abychom mohli sčítat a násobit velká celá čísla, musíme se tedy naučit sčítat a násobit... mnohočleny.

⁷ Vzhledem k tomu, že v současnosti neznáme žádné rychlé metody prvečíselného rozkladu, je velmi nepravděpodobné, že by se to podařilo třetí straně.

V jazyce C++ lze mnohočleny nejsnáze reprezentovat pomocí polí, která slouží k uložení koeficientů. Mnohočlen stupně n a proměnné x se obecně definuje takto:

$$W(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

Výpočet hodnoty $W(b)$ pro jisté b vypadá dosti náročně, protože vyžaduje mnoho operací násobení a sčítání:

```
int vypocitej_mnohoclen(int a, int w[], int rozm)
{ // klasická metoda
    int res=0, pot=1;
    for(int j=rozm-1; j>=0; j--)
    {
        res+=pot*w[j]; // částečné součty
        pot*=a;          // další mocnina a
    }
    return res;
}
```

(V případě mocnin s jinými než celočíselnými koeficienty je potřeba všude zaměnit typ `int` na `double`.)

Existuje však tzv. *Hornerovo schéma*⁸, díky kterému lze výpočet $W(b)$ značně zjednodušit:

$$W(b) = (\dots((a_n b + a_{n-1}) b + a_{n-2}) b + \dots + a_1) b + a_0.$$

Hornerovo schéma je možné realizovat takto:

```
horner.cpp
const int n=5; // stupeň mnohočlenu

int vypocitej_mnohoclen_H(int a, int w[], int rozm)
// Hornerovo schéma
{
    int res=w[0];
    for(int j=1; j<rozm; res=res*a+w[j++]);
    return res;
}
int main()
{
    int w[n]={1,4,-2,0,7};           // koeficienty mnohočlenu
    cout << vypocitej_mnohoclen(2,w,n) << endl; // klasická metoda
    cout << vypocitej_mnohoclen_H(2,w,n) << endl; // Hornerovo schéma
}
```

Díky reprezentaci založené na polí lze mnohočleny také snadno sčítat a násobit:

```
wielom.cpp
const int n=3; // stupeň mnohočlenu

void sekti_mnohocleny(int x[], int y[], int z[], int rozm)
{
    for(int i=0; i<rozm; i++)

```

⁸ Proceduru ve skutečnosti objevil Isaac Newton, ale historie ji připsala Hornerovi.

KAPITOLA 13 Kódování a komprese dat

```
z[i]=x[i]+y[i];           // mnohočlen z=x+y
}

void vynasob_mnohocleny(int x[], int y[], int z[], int rozm)
{
    int i,j;
    for(i=0; i<2*rozm-1; i++)
        z[i]=0;                      // nulování výsledku
    for(i=0;i<rozm;i++)
        for(j=0;j<rozm;j++)
            z[i+j]=z[i+j] + x[i]*y[j];
}
```

Uvedené algoritmy přímo odpovídají praktickým postupům, které jsme se naučili na základní nebo střední škole. Jakou mají zásadní vadu? Spočívá právě v tom, co jsme považovali za výhodu: v reprezentaci pomocí pole (která poskytuje jednoduchý přístup ke koeficientům). Tato reprezentace není příliš efektivní z hlediska využití paměti. Nejlépe se o tom přesvědčíme, když se pokusíme vynásobit například následující mnohočleny:

$$(2x^{1600} + 3x^{900}) \cdot (3x^{85} + 1).$$

Můžeme samozřejmě vyhradit pole velikosti 1 600, 85 a 1 600+85 (na výsledek), ale vzhledem k tomu, že budou obsahovat hlavně nuly, to není zrovna nejrozumnější.

Východisko však poskytuje reprezentace mnohočlenu pomocí jednosměrného seznamu. Vybereme nejjednodušší řešení, ve kterém nové členy mnohočlenu vkládáme na začátek seznamu (uživatel musí nové členy vkládat v určeném pořadí: do nejvyšších mocnin k nejnižším nebo naopak). Nulové členy není nutné uchovávat:

```
wielom2.cpp
typedef struct wsp
{
    int c;
    int j;
    struct wsp *dalsi;
}KOEFICIENTY, *KOEFICIENTY_PTR;

KOEFICIENTY_PTR vloz(KOEFICIENTY_PTR p, int c, int j)
{ // přidání nového uzlu (koeficientu) do mnohočlenu
    if(c!=0) // pouze prvky c*(x^j), pro c!=0
    {
        KOEFICIENTY_PTR q=new KOEFICIENTY;
        q->c=c;
        q->j=j;
        q->dalsi=p;
        return q;
    }
    else
        return p;// seznam se nezměnil
}
```

Funkce obsluhující tuto reprezentaci se poněkud komplikují, ale algoritmy jsou výrazně efektivnější a mnohem úsporněji pracují s pamětí. Podívejme se na funkci, která sčítá dva mnohočleny:

```
KOEFICIENTY_PTR secti(KOEFICIENTY_PTR x, KOEFICIENTY_PTR y)
{ // vrácení mnohočlenu x+y
    KOEFICIENTY_PTR res=NULL;
    while((x!=NULL) && (y!=NULL))
        if(x->j==y->j)
        {
            res=vloz(res,x->c+y->c,x->j);
            x=x->dalsi;
            y=y->dalsi;
        }
        else
            if(x->j<y->j)
            {
                res=vloz(res,x->c,x->j);
                x=x->dalsi;
            }
            else
                if(y->j<x->j)
                {
                    res=vloz(res,y->c,y->j);
                    y=y->dalsi;
                }
        // V této fázi může x nebo y ještě obsahovat prvky,
        // které zatím neobsloužil cyklus while
        // vzhledem ke své podmínce. Vkládáme tedy zbytek
        // koeficientů (pokud existují):
        while (x!=NULL)
        {
            res=vloz(res,x->c,x->j);
            x=x->dalsi;
        }
        while (y!=NULL)
        {
            res=vloz(res,y->c,y->j);
            y=y->dalsi;
        }
    return res;
}
```

Algoritmus funkce `secti` jsme uvedli v podobě, která je co nejjednodušší a nejlépe se analyzuje (máte-li dost volného času, můžete v kódu provést různá drobná zlepšení).

Ještě se podívejme, jak lze výše uvedené funkce používat⁹:

```
int main()
{
    KOEFICIENTY_PTR pw1, pw2, pw3, pwtemp;
    pw1=pw2=pw3=pwtemp=NULL;
    pw1=vloz(pw1,5,1700);
    pw1=vloz(pw1,6,700);
    pw1=vloz(pw1,10,50);
    pw1=vloz(pw1,5,0);
```

⁹ Verze dostupná ke stažení obsahuje funkci na zjištění hodnoty, kterou můžete vyzkoušet v praxi.

```

pw2=v1oz(pw2, 6, 1800);
pw2=v1oz(pw2, -6, 700);
pw2=v1oz(pw2, 5, 50);
pw2=v1oz(pw2, 15, 0);

pw3=secti(pw1, pw2);
}

```

Je zřejmé, že kód implementuje následující mnohočleny:

- $pw1(x) = 5x^{1700} + 6x^{700} + 10x^{50} + 5$,
- $pw2(x) = 6x^{1800} - 6x^{700} + 5x^{50} + 15$,
- $pw3(x) = pw1(x) + pw2(x) = 6x^{1800} + 5x^{1700} + 15x^{50} + 20$.

Ve výkladu o systému šifrování dat RSA jsme narazili na nevýhodu, která se týká operací s velmi velkými celými čísly. Chceme-li získat šifrovou posloupnost, která vychází z určitého textu M^{10} , musíme vypočítat výraz:

$$\{\text{šifrová posloupnost}\} = M^P \bmod N.$$

Umocnění můžeme realizovat běžným násobením, ale jak postupovat při vyhodnocení funkce modulo? Jak například zvládnout výpočet:

$$12\ 9876\ 0002\ 6000\ 0000\ 0054 \bmod N?$$

Když ovšem uvedené číslo vyjádříme jako mnohočlen o základu $x = 10\ 000$, dostaneme značně jednodušší výraz:

$$12(x^5 \bmod N) + 9876(x^4 \bmod N) + 2(x^3 \bmod N) + 6000(x^2 \bmod N) + 54.$$

Hodnoty v závorkách jsou konstanty, které stačí vypočítat jen jednou a poté je „natvrdo“ zapsat do šifrovacího programu. Tím poněkud zmírníme principiální vadu systému RSA, která spočívá v jeho pomalosti.

Kvůli tomuto nedostatku algoritmu RSA vznikla zajímavá varianta metody, která se poprvé uplatnila nejspíše v programu PGP. Spočívá v tom, že data jsou šifrována rychlou symetrickou šifrou (např. 3DES) pomocí zcela náhodného klíče. Samotný klíč je pak zašifrován pomalejší asymetrickou šifrou a připojen ke zprávě. Příjemce nejdříve rozšifruje symetrický klíč a poté pomocí něj rozšifruje vlastní data. Tímto způsobem dosáhneme požadovaných cílů – neboli rychlého šifrování objemných dat a současně i bezpečného předávání klíče, který umožní tato data dešifrovat.

Primitivní metody

Postupy uvedené v této kapitole je potřeba brát spíše jako kuriozity než metody bezpečného šifrování, ačkoli je můžeme využít v jednoduchých programech, které slídilům v našich datech poněkud přidělají práci.

První metoda využívá vlastnost funkce XOR (funkce nonekvivalence, viz přílohu B) při inverzi bitů. Při první aplikaci funkce XOR na jistou binární hodnotu a znak smluvně nazývaný klíčem (znak je totiž tvořen posloupností bitů) dochází k šifrování a při druhé k dešifrování.

Podívejme se na ukázkový program v jazyce C++:

10 Vzpomeňme si, že když každé písmeno tohoto textu zaměníme jistým číslem (např. v kódu ASCII), můžeme celek považovat za jedno velmi velké číslo M .

```
xor.cpp
void Xor(char *s, char xor_key)
{
    for (int i=0; s[i] != '\0'; i++)
        s[i]= s[i] ^ xor_key;
}

int main()
{
    char s[]="ota je u auta";
    cout << "Původní posloupnost:\t" << s << endl;
    Xor(s,12);
    cout << "Zašifrovaná posloupnost:\t" << s << endl;
    Xor(s,12);
    cout << "Dešifrovaná posloupnost:\t" << s << endl;
}
```

Výstup programu:

```
Původní posloupnost: ota je u auta
Zašifrovaná posloupnost: cxm,fi,y,myxm
Dešifrovaná posloupnost: ota je u auta
```

Podobně prostá metoda využívá vlastnost kódu ASCII, kde známe maximální hodnotu kódu přiřazeného některému znaku:

```
255.cpp
void odecti(char *s)
{
    for (int i=0; s[i] != '\0'; i++)
        s[i]=255-s[i];
}
```

Funkce main je téměř identická (místo funkce xor voláme funkci odecti(s) bez parametru):

```
int main()
{
    char s[]="ota je u auta";
    cout << "Původní posloupnost:\t" << s << endl;
    odecti(s);
    cout << "Zašifrovaná posloupnost:\t" << s << endl;
    odecti(s);
    cout << "Dešifrovaná posloupnost:\t" << s << endl;
}
```

Výstup programu:

```
Původní posloupnost: ota je u auta
Zašifrovaná posloupnost: flžšøšššžšž
Dešifrovaná posloupnost: ota je u auta
```

Luštění šifer

V předchozích částech kapitoly jsme se zabývali dvěma základními třídami šifrovacích algoritmů. Na odlehčení tématu tedy nyní můžeme uvést několik základních postupů luštění šifer. Pokud bychom měli tuto problematiku zpracovat vyčerpávajícím způsobem, museli bychom napsat další knihu. Proto se jen zběžně zmíníme o několika nejoblíbenějších metodách:

- *Krádež klíče* – není nutné provést to osobně, protože někdy stačí zachytit klíč během přenosu mezi příjemcem a odesilatelem informací. U symetrických algoritmů může mít záludnou formu záměny klíče ve fázi, kdy si obě strany vyměňují klíče, aby mohly později bezpečně komunikovat. Útočník zaujme na komunikační trase pozici mezi odesilatelem a příjemcem a provádí nezávislou výměnu klíčů s oběma stranami!
- *Metoda „hrubou silou“* (ang. *brute-force*) – útočník postupně zkouší všechny možné kombinace klíčů. Požadavky této metody na výpočetní výkon rostou s délkou klíče exponenciálně: například 40bitový klíč vyžaduje 2^{32} kroků, které dnešní osobní počítače dokáží provést prakticky v reálném čase. Obecně se předpokládá, že 128bitové klíče nebude možné zlomit v několika nejbližších letech. Přitom však počítáme výhradně s přístupem typu *hrubou silou*, ačkoli se může objevit nějaký matematický génius, který přijde s efektivnější metodou. Není však důvod propadat pesimizmu, protože mnoho systémů se v současnosti konstruuje tak, že klíče mají dynamický charakter a poměrně rychle ztrácejí platnost. Kromě toho technologický postup může opravdu zpochybnit dnes využívané metody. Od takzvaných kvantových počítačů (v době psaní této knihy existují výhradně na papíře) se očekává, že přinesou revoluční nárůst výpočetního výkonu (oproti současným počítačům o několik rádů). Metoda typu *hrubou silou* by se pak dala nasadit i na velmi dlouhé klíče.
- *Dedukce na základě celé zprávy nebo její části* – útočník může odhadnout, že přenášená komunikace obsahuje známé prvky, např. adresy a názvy firem v hlavičkách, a z tohoto hlediska se může pokusit o rychlejší dešifrování algoritmu nebo o narušení jeho struktury.

Čtenářům, kteří se zajímají o kryptografické algoritmy a chtěli by s nimi experimentovat, doporučují, aby si prohlédli knihovnu **CryptPak406.zip** Markuse Hahna, která je součástí archivu ZIP dostupného ke stažení. Je distribuována na principu open source a připravena ke komplikaci v prostředí Win32 a Linux.

Techniky komprese dat

Mezi historicky první příklady komprese dat patří *Morseova abeceda*. Samuel Morse si povšiml, že některá písmena abecedy se v textech vyskytují častěji než jiná. Tento poznatek pak využil ve svém kódovacím systému, který byl založen na dvou znacích: čárce (–) a tečce (.), které bylo možné snadno přenášet pomocí telegrafu. Vysílané texty bylo možné zkrátit díky tomu, že častěji se vyskytující znaky byly nahrazeny kratšími kódovými sekvencemi a vzácnější znaky naopak delšími¹¹ (viz tabulka 13.2).

¹¹ O autorství tohoto kódovacího systému se vedou spory, ale do historie se zapsal Samuel Morse, který první depeši sestavenou pomocí této abecedy vyslal z Washingtonu do Baltimoru v roce 1844.

Tabulka 13.2: Morseova abeceda

Znak	Kód	Znak	Kód
A	- .	N	- - .
B	- ...	O	- ---
C	- . - .	P	. - - .
D	- ..	Q	- - . -
E	.	R	. - .
F	.. - .	S	...
G	- - .	T	-
H	U	. . -
I	..	V	... -
J	. - - -	W	. - -
K	- . =	X	- . . -
L	. - ..	Y	- . - -
M	- -	Z	- - ..

Koncepcně podobný systém kódování se využívá v Braillově písmu, jehož znaky (výrazy) mají podobu dvourozměrné mřížky velikosti 2×3 , která obsahuje vypouklá nebo plochá políčka. Vzhledem k tomu, že lze vytvořit $2^6 = 64$ odlišných znaků, což je více než písmen latinské abecedy (26), zbývající kódy mohou reprezentovat často používaná slova – jeden speciální znak proto informuje o tom, že za ním následuje celé slovo, a nikoli pouze znak¹².

Obecný matematický model komprese dat – podobně jako u šifrování a dešifrování – obsahuje dva typy algoritmů: kompresní a dekompressní. Co se týče samotných algoritmů, nejčastěji se kompresní algoritmy řadí do dvou skupin:

- *bezeztrátové* (po zakódování a rekonstrukci dat získáme výsledek, který na 100 % odpovídá originálu),
- *ztrátové* (výsledek komprese a dekomprese se může od předlohy poněkud lišit).

U ztrátové komprese připouštíme jistý úbytek informací, samozřejmě v aplikacích, kde je to možné, například při komprezi zvuku či obrazu. Ztrátovou kompresi jistě nelze použít v systémech, kde záleží na stoprocentní věrohodnosti dat, například v bankovních systémech nebo systémech podobné kategorie.

Kompresních algoritmů je velmi mnoho a celá oblast se navíc neustále rozvíjí, což je způsobeno současnými technologickými potřebami. Aniž bychom zabíhali do klasifikačních podrobností, můžeme říci, že kompresní algoritmy lze hodnotit podle těchto hlavních kritérií:

- rychlosť fungování,
- stupeň komprese, což je koeficient zmenšení objemu vstupních dat po jejich zpracování kompresním algoritmem.

Všechny kompresní algoritmy využívají toho, že datové soubory obsahují více dat, než je fakticky potřeba k uchování stejné informační hodnoty. Tuto vlastnost datového souboru odborně označujeme jako redundanci – úplnou matematickou definici si odpustíme, abychom z této knihy neudělali

12 Abecedu publikoval roku 1837 francouzský pedagog Louis Braille. Sám byl od dětství slepý, a protože učil v instituci pro nevidomé, chtěl postiženým dětem umožnit čtení.

KAPITOLA 13 Kódování a komprese dat

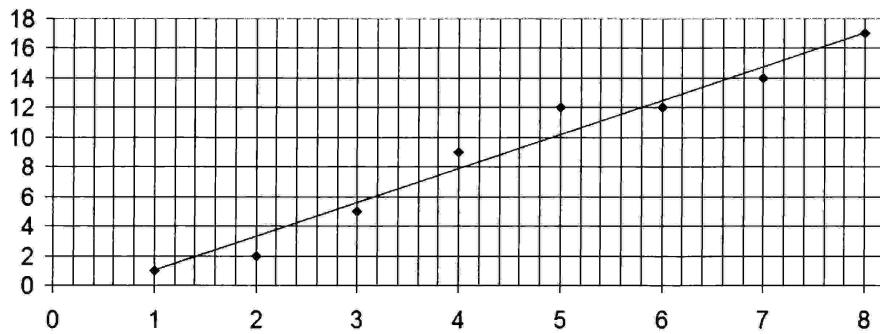
příručku teorie informace. Samotný pojem ostatně není nijak exotický, například v jazykovědě se jedná o výskyt jazykových prvků, které z funkčního hlediska nejsou potřebné.

Komprese pomocí matematického modelování

Při kompresi založené na matematickém modelování se snažíme obnovit redundanci datového souboru. Tuto metodu lze poměrně snadno vysvětlit na jednoduchém příkladu, i když v praxi není příliš užitečný. Má však výukovou hodnotu a umožňuje popsat typické schéma, podle kterého algoritmy tohoto typu fungují.

Předpokládejme, že chceme přenést následující číselnou posloupnost: 1, 2, 5, 9, 12, 12, 14 a 17. Jedná se o část celé zprávy. Máme za úkol zjistit, zda kvůli předání této posloupnosti musíme skutečně vyslat až $8 \cdot 5 = 40$ bitů (každé z čísel od 1 do 17 lze binárně zakódovat pomocí 5 bitů). Ne-mohli bychom snad stejnou informaci přenést poněkud úsporněji?

Jedno řešení spočívá v odvození matematické funkce, která na základě sekvenčního čísla vzorku umožní vypočítat hodnotu přenášeného znaku. Abychom dokázali tuto funkci definovat, pokusme se data vynést do jednoduchého grafu (obrázek 13.3).



Obrázek 13.3: Komprese, která využívá matematické modelování datového souboru

Už na první pohled vidíme, že se usporádání dat velmi dobře blíží lineární závislosti (znázorněné v grafu). Řekněme, že budeme tuto datovou posloupnost modelovat pomocí funkce $(7n-4)/3$, kde n označuje číslo vzorku. Porovnejme nyní kódovou posloupnost s navrženým matematickým modelem:

n	1	2	3	4	5	6	7	8
Hodnota	1	2	5	9	12	12	14	17
$(7n-4)/3$	1	3	6	8	10	13	15	17
Odchylka	0	1	1	-1	-2	1	1	0

Modelovací funkci můžeme zakódovat nastálo (případně můžeme její parametry předat na začátku přenosu dat). Díky znalosti této funkce pak stačí vysílat pouze informace o odchylkách: 0, 1, 1, -1, -2, 1, 1, 0. V řadě odchylek se vyskytují pouze čtyři hodnoty:

Odchylka	Kód
-2	00
-1	01
0	10
1	11

Můžeme je tedy zakódovat pomocí pouhých 2 bitů! Když odhlédneme od modelovací funkce, informace o samotných datech nyní nemusí zabrat 40 bitů, ale jen $8 \cdot 2 = 16$.

Komprese metodou RLE

Kompresní metoda *RLE* (ang. *Run Length Encoding*) je založena na jednoduchém bezetrátovém algoritmu, který se ideálně hodí ke komprezi datových bloků, kde se po sobě mnohokrát opakuje stejné znaky. Nejjednodušším příkladem mohou být grafické soubory, které obsahují obrázky s jednobarevnými plochami. Textové soubory lze oproti tomu komprimovat mnohem hůře.

Předpokládejme, že v grafickém souboru kódujeme data o barvách (a případně jiné informace) pomocí jedné osmibitové hodnoty. V našich příkladech budeme tuto hodnotu symbolizovat písmeny A, B, C atd. V rámci příkladu se podívejme na fragment grafického souboru:

AABBBBBBCCDDABBBDDDDDDDDDDAAAAAA (30 znaků)

Datový blok obsahuje 30 bajtů, ale obsažené informace můžeme stejně dobře zakódovat ve tvaru:

AA*4BCDDA*3B*9D*6A (17 znaků),

kde sekvence $\{*\ N\ Z\}$ označuje N -násobný výskyt znaku Z . Symbol $*$ slouží jako oddělovač, který umožňuje rozlišit nekódovaná data od dat, která jsme se rozhodli nahradit kódovou sekvensí.

Zamysleme se nyní, jak bychom měli definovat kritérium, zda data kódovat či nikoli.

Otzáka není tak složitá, jak vypadá. Stačí si uvědomit, co je obecným cílem komprese. V záplavě uváděných algoritmů bychom na to mohli zapomenout, takže si tedy připomeňme: jde nám o zmenšení rozměru datového bloku, aniž bychom ztratili část jeho informační hodnoty.

Stojí za to kódovat jeden znak Z ? Samozřejmě že nikoli, místo efektu komprese bychom tím velikost datového souboru naopak zvětšili:

$Z \rightarrow *1Z$ (3 znaky místo jednoho)

Totéž platí pro dva, a dokonce i pro tři znaky. V posledním případě sice objem datového souboru neroste, ale nedosáhneme ani jeho komprese.

$ZZZ \rightarrow *3Z$ (3 znaky místo... tří).

Komprese RLE tedy začíná být efektivní, když ponecháváme beze změny sekvence od 1 do 3 znaků a zaměříme se na komprezi bloků, kde se opakuje 4 a více znaků.

V archivu ZIP dostupném ke stažení se nachází soubor **rle8_sc.zip**, který obsahuje kódovací a dekódovací procedury *RLE*, jejichž autorem je Shaun Case. Tyto programy stažené z Internetu jsou dostupné na základě licence *public domain*. Soubory jsou převzaty beze změn a mohou vyžadovat přizpůsobení pro konkrétní komplilátor jazyka C++.

Původně byly napsány v jazyce C pro komplilátor Borland C++ 2.0, ale lze je kompilovat bez problémů i pomocí komplilátoru Borland C++ Builder 5.0. Chcete-li použít prostředí Visual C++, přečtěte si soubor **rle.txt**.

Komprese dat Huffmanovou metodou

Jako první podrobně popíšeme *Huffmanův* algoritmus. Jedná se přímo o učebnicový příklad toho, jak lze vytvořit dobrý algoritmus, který je možné snadno implementovat pomocí moderních

KAPITOLA 13 Kódování a komprese dat

programovacích jazyků. Samotný algoritmus patří do početné třídy tzv. *prefixových algoritmů*. Než však přejdeme k jeho popisu, vysvětlíme nejdříve hlavní myšlenku, na které je založen.

Kód, který se rozhodněm použít, se může značně lišit od známého kódu ASCII. Jak si pamatuji, kód ASCII je tvořen tabulkou osmibitových textových znaků (některé z nich se sice v češtině nepoužívají, ale to zde není příliš důležité). Základní vlastností tohoto kódování je stejná délka každého kódového slova, které odpovídá danému znaku: 8 bitů. Je to povinné? Samozřejmě že nikoli. Podívejme se na příklad kódování znaků jisté abecedy s pěti znaky (tabulka 13.3).

Tabulka 13.3: Příklad kódování znaků jisté abecedy s pěti znaky

Znak	Bitový kód
	000
	001
	01
	10
	11

Někde daleko v džungli žije kmen, který kombinací těchto pěti znaků dokáže vyjádřit úplně vše: vypovědět válku, uzavřít příměří, poprosit o jídlo, ohlásit předpověď počasí... Texty se zapisují na listy jisté rostliny, které se vyznačují vysokou trvanlivostí. Kvůli efektivnější komunikaci lidé vymysleli systém rychlého předávání zpráv pomocí signálů trubek, které jsou slyšitelné na značné vzdálenosti.

Dva krátké signály znamenají znak , krátký a dlouhý signál zastupují znak atd., jak je patrné z následující tabulky (0 – dlouhý signál, 1 – krátký signál). Nepochybň se jedná o binární kód (i když si to tajemní domorodci sami neuvědomují).

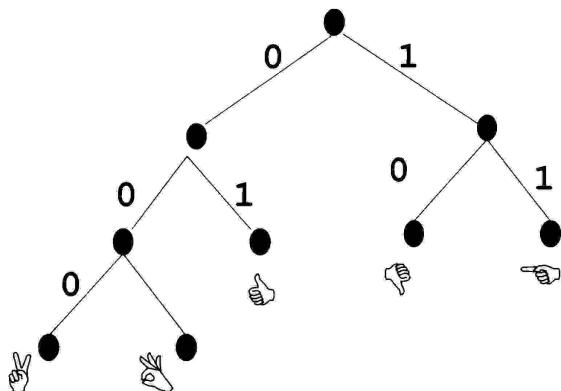
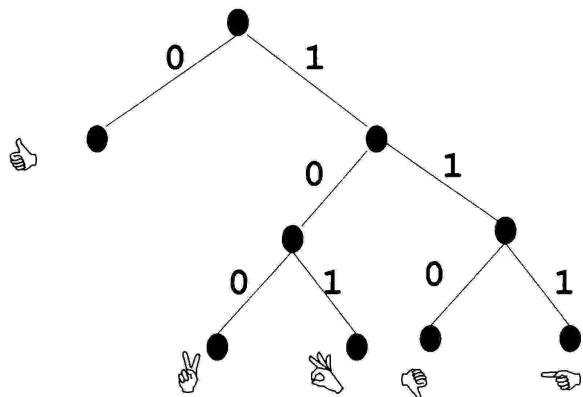
Řekněme, že jsme jistého dne zachytili následující signály: 011110000001 (odesilatel předal zprávu: , tj. „pošlete ještě čerstvé melouny“). Je možné, abychom zprávu přečetli nesprávným způsobem – neboli může se stát, že některé znaky zaměníme? Zkusme to:

- 0 – znakem může být: nebo nebo .
- 01 – už víme, že to je !
- 01 1 – znakem může být: nebo .
- 01 11 – už víme, že to je !
- 01 11 1 – znakem může být: nebo
- ...
- (atd.)

Jasně vidíme, že k omylemu nemůže dojít, protože žádný kódový znak není *předponou (prefixem)* jiného kódového znaku. Našli jsme důležitou vlastnost kódu: musí být *jednoznačný*, tzn. nemohou nastat pochybnosti, zda daná sekvence patří ke znaku X nebo snad ke znaku Y.

Kód s touto vlastností lze sestavit poměrně snadno, když jeho abecedu reprezentujeme formou tzv. kódového stromu. V našem příkladu vypadá strom tak jako na obrázku 13.4.

Při procházení stromu (od jeho kořene až k listům) postupně navštěvujeme větve označené etiketami 0 (levé) nebo 1 (pravé). Trasa, kterou jsme došli k danému listu, představuje jeho binární kódové slovo. Zásadním problémem kódových stromů je jejich... nadbytek. Pro danou abecedu můžeme sestavit celý les kódových stromů, jak je patrné z obrázku 13.5.

**Obrázek 13.4:** Příklad kódového stromu (1)**Obrázek 13.5:** Příklad kódového stromu (2)

Nabízí se tedy otázka: Který strom je nejlepší? Kritérium jakosti kódového stromu samozřejmě souvisí s našim hlavním cílem – kompresí. Za nejlepší zvolíme takový kód, který nám zajistí nejvyšší stupeň komprese. Všimněme si, že kódová slova nejsou stejně dlouhá (v našem příkladu obsahovala 2 či 3 znaky). Pokud nějakým kouzelným způsobem dosáhneme toho, aby znaky vyskytující se v kódovaném textu nejčastěji měly nejkratší kódová slova a znaky, které se objevují sporadicky, zase slova nejdelší, bude získaná bitová reprezentace kratší než kterýkoli jiný binární kód.

Na tomto postřehu je založen *Huffmanův kód*, který umožňuje získat optimální kódový strom. Snadno si domyslíme, že tento kód potřebuje údaje o frekvenci výskytu znaků v textu. Může se jednat o výsledky lingvistické analýzy, která udává pravděpodobnost výskytu určitých znaků v daném jazyce, nebo můžeme vycházet z vlastního předběžného zpracování textu, který chceme zakódovat. Konkrétní postup závisí na tom, co chceme kódovat (a případně vysílat): texty přirozeného jazyka, pro který známe pravděpodobnost výskytu jednotlivých písmen, nebo z hlediska obsahu náhodné „binární“ soubory (např. digitální fotografie, počítačové programy atp.).

Pro zájemce uvádíme tabulku, která obsahuje data týkající se českého jazyka¹³ (viz tabulku 13.4).

¹³ <http://www.algoritmy.net/article/40/Cetnost-znaku-CJ>

Tabulka 13.4: Pravděpodobnost výskytu znaků v českém textu

Znak	Četnost (%)	Znak	Četnost (%)	Znak	Četnost (%)
a	6,2193 %	í	3,2699 %	t	5,7268 %
á	2,2355 %	j	2,1194 %	ť	0,0426 %
b	1,5582 %	k	3,7367 %	u	3,1443 %
c	1,6067 %	l	3,8424 %	ú	0,1031 %
č	0,9490 %	m	3,2267 %	ű	0,6948 %
d	3,6019 %	n	6,5353 %	v	4,6616 %
ď	0,0222 %	ń	0,0814 %	w	0,0088 %
e	7,6952 %	o	8,6664 %	x	0,0755 %
é	1,3346 %	ó	0,0313 %	y	1,9093 %
ě	1,6453 %	p	3,4127 %	ý	1,0721 %
f	0,2732 %	q	0,0013 %	z	2,1987 %
g	0,2729 %	r	3,6970 %	ž	0,9952 %
h	1,2712 %	ř	1,2166 %		
ch	1,1709 %	s	4,5160 %		
i	4,3528 %	š	0,8052 %		

Huffmanův algoritmus bezprostředně využívá právě takové tabulky. Vyhledává a seskupuje méně časté znaky, aby jim následně přiřadil nejdélší binární slova, zatímco častěji se vyskytujícím znakům přiděluje naopak slova nejkratší. Algoritmus může pracovat s pravděpodobnostmi či frekvencemi výskytu znaků. Dále uvádíme klasický algoritmus na konstrukci Huffmanova kódu, který budeme dále analyzovat na konkrétním početním příkladu.

FÁZE REDUKCE (směr: dolů)

1. Seřadíme znaky kódované abecedy podle klesající pravděpodobnosti jejich výskytu.
2. Redukujeme abecedu tak, že dva nejméně pravděpodobné znaky spojíme v jeden zástupný znak, jehož pravděpodobnost se rovná součtu pravděpodobností kombinovaných znaků.
3. Jestliže redukovaná abeceda obsahuje 2 znaky (zástupné), přejdeme k bodu 4. V opačném případě se vracíme k bodu 2.

FÁZE KONSTRUKCE KÓDU (směr: nahoru)

4. Dvěma znakům redukované abecedy přiřadíme kódová slova 0 a 1.
5. Na nejnižší pozice kódových slov, která odpovídají dvěma nejméně pravděpodobným znakům redukované abecedy, zapíšeme znaky 0 a 1.
6. Pokud jsme obnovili počáteční abecedu, algoritmus končí. V opačném případě se vracíme k bodu 5.

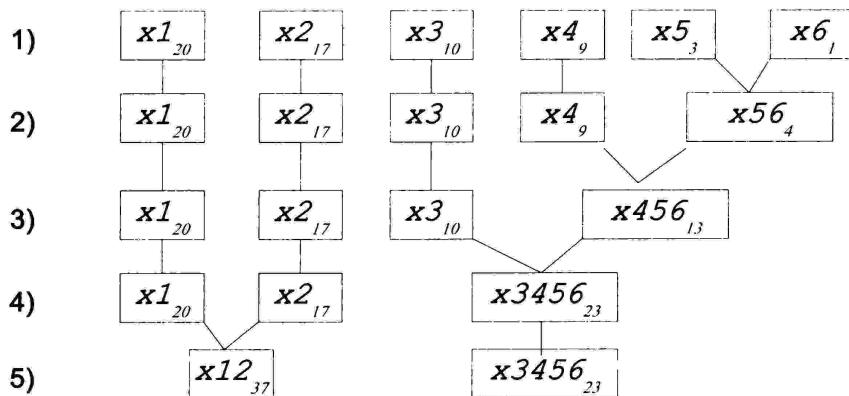
Algoritmus možná na první pohled není příliš srozumitelný, ale všechny nejasnosti by měl vysvětlit konkrétní příklad, který zakrátko budeme analyzovat.

Předpokládejme, že máme k dispozici abecedu tvořenou šesti znaky: x_1, x_2, x_3, x_4, x_5 a x_6 . K zakódování dostaneme test o délce 60 znaků, které se v textu vyskytují v následujících počtech: 20,

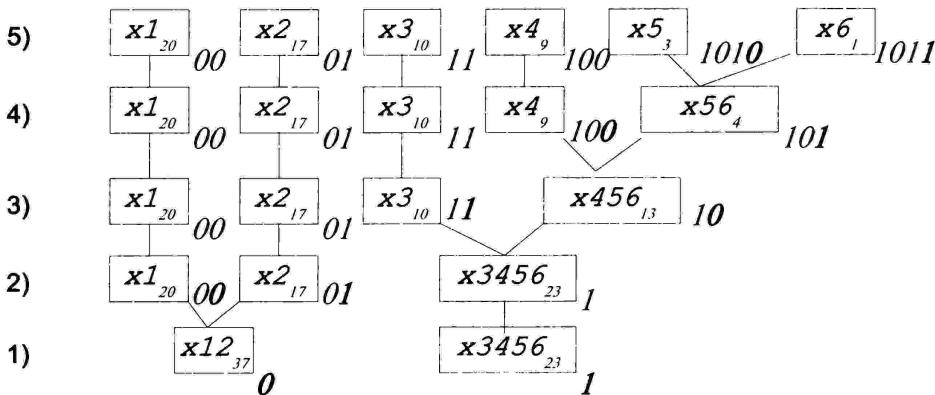
17, 10, 9, 3 a 1. Chceme-li zakódovat šest různých znaků, potřebujeme alespoň 3 bity ($6 < 2^3$). Zakódovaný text by měl tedy délku $3 \cdot 60 = 180$ bitů. Podívejme se nyní, jak se projeví *Huffmanův* algoritmus, který nám umožní získat optimální binární kód.

Když budeme postupovat podle pravidel uvedených v předchozím popisu algoritmu, získáme následující redukci (viz obrázek 13.6).

Obrázek znázorňuje šest kroků redukce kódované abecedy (nenechejme se ovlivnit podobou schématu, nejde zatím o binární strom). Znaky x_5 a x_6 se vyskytují nejméně často, takže je redukujeme na zástupný znak, který označíme jako x_{56} . Obdobnou operaci provedeme v každém následném kroku, až se dostaneme do stavu, kde zbývají pouze dva (zástupné) znaky abecedy. Fáze *redukce* je u konce a můžeme přejít do fáze *konstrukce kódu*. Znázorňuje ji obrázek 13.7.



Obrázek 13.6: Konstrukce Huffmanova kódu – fáze redukce



Obrázek 13.7: Konstrukce Huffmanova kódu – fáze tvorby kódu

Bity 0 a 1, které v dané etapě přidáváme k redukovaným znakům abecedy, jsou zvýrazněny **tučně**. Pochopení principu konstrukce kódu podle obrázku by nemělo činit potíže, tím spíše, že příklad vychází z krátké abecedy.

Pomocí klasické metody binárního kódování bychom zprávu s délkou 60 znaků (sestavenou z abecedy se 6 znaky) zakódovali pomocí $60 \cdot 3 = 180$ bitů. Při použití *Huffmanova* kódu by stejná zpráva vyžadovala $20 \cdot 2 + 17 \cdot 2 + 10 \cdot 2 + 9 \cdot 3 + 3 \cdot 4 + 1 \cdot 4 = 137$ znaků (získáme asi 23 % původní délky).

Už víme, jak konstruovat kód, takže bychom se měli zamyslet nad jeho programovou implementací v jazyce C++. Nebudeme prezentovat hotový kódovací program, protože by nám zabral příliš

KAPITOLA 13 Kódování a komprese dat

mnoho místa. Vhodnou metodou by bylo zkopirovat grafickou strukturu, která je znázorněna na dvou předchozích obrázcích. Je to přeci dvourozměrné pole s maximálními rozměry 6×5 . V jeho buňkách bychom museli uchovávat poměrně složité údaje: kód znaku, frekvenci jeho výskytu a binární kód. Zapamatování poslední položky by nemělo být složité, protože jazyk C++ nabízí různé možnosti: pole 0-1, celé číslo, jehož binární reprezentace by odpovídala vytvářenému kódu, atp. Neměli bychom zapomenout, že spolu s kódovanou zprávou musíme poslat také její... Huffmanův kód. (Příjemce by jinak zprávu nedokázal přečíst.)

Musíme tedy řešit docela hodně technických problémů. Výše navržený způsob samozřejmě není povinný. Velmi zajímavý přístup (spolu s hotovým kódem C++) předkládá [Sed92]. Autor vyhledává znaky s nejmenšími pravděpodobnostmi pomocí prioritní fronty, ale tento přístup zase poněkud komplikuje tvorbu binárního kódu na základě redukováné abecedy. K výhodám algoritmů založených na „haldových“ strukturách však nepochybňě patří jejich efektivita: operace s haldou totiž patří do třídy $\log N$, což může mít viditelný praktický efekt. Podívejme se tedy, jak lze vyjádřit Huffmanův algoritmus tvorby kódového stromu právě při použití těchto datových struktur:

```
Huffman(s, f)
// s - kódovaná binární posloupnost znaků
// f - tabulka četnosti výskytu znaků v abecedě
{
    vlož všechny znaky posloupnosti s do haldy H
    podle jejich četnosti;
    dokud halda H není prázdná proved
    {
        jestliže halda H obsahuje pouze jeden znak X pak
            X se stává kořenem Huffmanova stromu T;
        v opačném případě
        {
            - vezmi dva znaky X a Y s nejmenšími četnostmi  $f_1$  a  $f_2$ 
            a odstraň je z haldy H;
            - nahraď znaky X a Y zástupným znakem Z,
            jehož četnost výskytu  $f=f_1 + f_2$ ;
            - vlož znak Z do haldy H;
            - vlož znaky X a Y do stromu T jako následníky Z;
        }
    }
    vrát strom T;
}
```

Tento algoritmus samozřejmě odpovídá výše uvedenému a liší se pouze formou zápisu.

Do teorie kódování a informací patří mnoho zajímavých témat značného praktického významu. Mnoho zajímavých otázek jsme v této kapitole nemohli otevřít kvůli nedostatku místa. Některé problémy lze navíc jen těžko převést na snadno srozumitelný kód C++. Kapitolu proto můžeme považovat za úvod do velmi rozsáhlého a atraktivního oboru.

V archivu ZIP dostupném ke stažení se nachází soubor **huf.zip**, který obsahuje kódovací a dekódovací programy založené na Huffmanově algoritmu, jejichž autorem je Shaun Case. Tyto programy stažené z Internetu jsou dostupné na základě licence *public domain*. Soubory jsou převzaty bez změn a mohou vyžadovat přizpůsobení pro konkrétní kompilátor jazyka C++. (Původně byly napsány v jazyce C pro kompilátor Borland C++ 2.0, ale lze je kompilovat bez problémů i pomocí kompilátoru Borland C++ Builder 5.0. Chcete-li použít prostředí Visual C++, přečtěte si soubor **huf.txt**.)

Kódování LZW

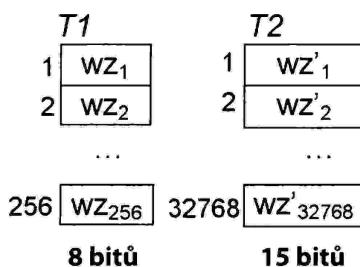
Kód *LZW* představuje dobrý a známý příklad kódování pomocí slovníku, který uchovává často opakované vzory. Výhoda takového slovníku leží v tom, že se přizpůsobuje obsahu textu. Oproti tomu tabulková kódování (viz kód ASCII či Morseova abeceda) používají dosti statický přístup, kde nás více zajímá abeceda daného jazyka než kódovaný text.

Metodu kódování *LZW* vymysleli v letech 1977 a 1978 Abraham Lempel a Jacob Ziv a v roce 1984 ji pak zdokonalil Terry Welch¹⁴. Název metody je odvozen z prvních písmen příjmení jejích autorů. Metoda se používá mj. v systému UNIX (funkce compress) nebo při kompresi obrázků ve formátu *GIF* (ang. *Graphics Interchange Format*). Připomeňme také, že algoritmus *LZW* je od roku 1995 patentově chráněn – vlastníkem patentu je firma *Unisys Corporation*. Producenti, kteří chtějí prodávat programy implementující algoritmus *LZW*, si musí od společnosti *Unisys Corporation* pořídit placenou licenci. Zdarma mohou algoritmus používat jen koncoví uživatelé a nekomerční organizace. Dochází zde k podivné situaci, kdy specifikace formátu *GIF* je sice bezplatná¹⁵, ale vzhledem k tomu, že zahrnuje algoritmus *LZW* (komerční), fakticky se mění na placený standard.

Princip slovníkového kódování na příkladech

Metoda kódování pomocí slovníku není příliš komplikovaná. Vytváříme přitom dva slovníky ke kódování vzorů, které se v kódovaném textu často vyskytují (počet vzorů je konečný a obecně nepříliš vysoký). První slovník zapíšeme do pole T1. Ke kódování zbývajících vzorů (prvků) objevujících se v textu slouží slovník T2 (je to v zásadě překladová funkce, a nikoli skutečný slovník s alokací paměti).

Příklad je znázorněn na obrázku 13.8. Příklad vychází z toho, že kódových znaků je tolik, že na jejich zakódování potřebujeme 15 bitů – všech možných hodnot tedy existuje 2^{15} (viz T2). Když ovšem víme (a nemáme zde místo vysvětlovat, odkud to víme), že v daném textu se nejčastěji vyskytuje 256 kódových posloupností wz_1, \dots, wz_{256} , můžeme je seskupit do reálného pole T1. Každá často se vyskytující posloupnost bude v zakódovaném textu zabírat 8 bitů (plus případně nějaký kontrolní bit, podle kterého tuto posloupnost poznáme). Analogicky platí, že řidce se vyskytující posloupnost bude kódovaná pomocí 15 bitů (eventuálně s dodatečným rozlišovacím bitem).



Obrázek 13.8: Slovníková metoda kódování

Přínos komprese založené na slovníkové metodě závisí na tom, kolik kódových posloupností vzhledem k jejich celkovému počtu je zařazeno mezi často se vyskytující (tyto posloupnosti v kódu za-

¹⁴ Terry A. Welch, „A Technique for High Performance Data Compression“ (Metoda vysoké výkonné komprese dat), *IEEE Computer*, sv. 17, č. 6, 1984, str. 8–19.

¹⁵ Jejím držitelem je společnost CompuServe Incorporated, kterou roku 1997 pohltila společnost AOL (America Online).

KAPITOLA 13 Kódování a komprese dat

bírají méně místa). Musíme tedy nějakým způsobem zajistit, aby algoritmus získal o kódovaném textu dostatečné informace, které umožní znstruovat požadované pole.

Metoda *LZW* elegantně implementuje algoritmus slovníkového kódování, kde algoritmus samocenně vytváří svůj slovník nejčastěji se vyskytujících znaků. Fungování algoritmu kódování *LZW* můžeme ukázat na příkladu jednoduché kódové posloupnosti. (Tento příklad by měl algoritmus vysvětlit lépe než teoretický popis.)

Než přejdeme k samotnému algoritmu, shrňme jeho hlavní vlastnosti:

- Neexistuje předem definovaný slovník – dynamicky se tvoří pro každou vstupní posloupnost (týká se to kódování i dekódování).
- Algoritmus má jen jeden průběh.

Algoritmus kódování metodou *LZW* lze popsát pomocí pseudokódu:

```

z = NULL;
while(čti znak k) // dokud není vstupní datový proud u konce
{
    if (zk ∈ SLOVNÍK) // zk je posloupnost vzniklá zřetězením z a k
        z = zk;
    else
    {
        přidej zk do SLOVNÍKu
        vypiš kód pro z;
        z = k; // posledním znakem začíná nová posloupnost
    }
}

```

Uvedený zápis znamená, že kodér (zařízení – kódující algoritmus) shromažďuje symboly ze vstupní posloupnosti a tvoří s nich jistou posloupnost s tak dlouho, dokud je s stále prvkem slovníku. V opačném případě algoritmus přidá s do slovníku a pokračuje tím, že posloupnost s inicializuje jejím posledním znakem.

Příklad:

Předpokládejme, že základní znaky mají kódy od 0 do 255 (kód ASCII) a pro znaky dodatečných symbolů ve slovníku vyhradíme indexy od 256 nahoru.

Vstupní datový proud: ^MNO^MN^MNN^MNP^MNQ

Z	k	VÝSTUP	index	symbol	Pozice v souboru
NULL	^				
^	M	^	256	^M	^MNO^MN^MNN^MNP^MNQ
M	N	M	257	MN	^MNO^MN^MNN^MNP^MNQ
N	O	N	258	NO	^MNO^MN^MNN^MNP^MNQ
O	^	O	259	O^	^MNO^MN^MNN^MNP^MNQ
^	M				
^M	N	256	260	^MN	^MNO^MN^MNN^MNP^MNQ
N	^	N	261	N^	^MNO^MN^MNN^MNP^MNQ
^	M				
^M	N				

Z	k	VÝSTUP	index	symbol	Pozice v souboru
AMN	N	260	262	^MNN	AMNO^MNA^MNN^MNP^MNQ
N	^				
N^	M	261	263	N^M	AMNO^MNA^MNN^MNP^MNQ
M	N				
MN	P	257	264	MNP	AMNO^MNA^MNN^MNP^MNQ
P	^	P	265	P^	AMNO^MNA^MNN^MNP^MNQ
^	M				
^M	N				
^MN	Q	260	266	^MNQ	AMNO^MNA^MNN^MNP^MNQ
Q	NULL	Q			

Začínáme na začátku souboru (NULL). Následující vstupní znak (^) se nachází ve slovníku, vypisujeme jej (VÝSTUP) a postupujeme dále: aktuální posloupnost $s = ^M$. Tato posloupnost ve slovníku neexistuje, takže ji přidáme na první volný index (256). Bereme poslední znak posloupnosti s , tj. M a připojujeme jej k následujícímu znaku (N). Ani tato posloupnost ve slovníku není, takže ji přidáme. Stejným způsobem do slovníku doplňujeme další kódy. Nejzajímavější fáze činnosti algoritmu nastává tehdy, když se začínají opakovat stále delší texty, které nahrazujeme dříve definovanými kódy a dosahujeme přitom kompresního efektu.

Dekódovací algoritmus není složitý. Vychází z identického vstupního slovníku jako kódující algoritmus.

Dekódování LZW:

```

čti znak k; // první znak
vypiš k;
w = k; // první kód
while (čti k) // dokud není vstupní datový proud u konce
{
    s = obsah slovníku pro k;
    vypiš s;
    přidej w + s[0] do SLOVNÍKu;
    w = s;
}

```

Vstupní datový proud: AMNO<256>N<260><261><257>P<260>Q

w	k	výstup	index	symbol
	^	^		
^	M	M	256	^M
M	N	N	257	MN
N	O	O	258	NO
O	<256>	^M	259	O^
<256>	N	N	260	^MN

KAPITOLA 13 Kódování a komprese dat

w	k	výstup	index	symbol
N	<260>	^MN	261	N^
<260>	<261>	N^	262	^MNN
<261>	<257>	MN	263	N^M
<257>	P	P	264	MNP
P	<260>	^MN	265	P^
<260>	Q	Q	266	^MNQ

Všimněme si, že algoritmus při dekódování zároveň sestavuje stejný slovník jako kódovací algoritmus a tento slovník používá v dalších fázích dekódování.

Popis formátu GIF

V této podkapitole se pokusíme ukázat, jak se v praxi uplatňuje metoda komprese *LZW* v oborovém standardu grafických souborů *GIF*, který se široce používá na webových stránkách.

Formát *GIF* má dvě varianty: *GIF87a* a *GIF89a* (viz prvních 6 znaků libovolného souboru v tomto formátu). Druhá verze formátu umožňuje ukládat animace a nastavovat průhlednost. Měli bychom vědět, že formát *GIF* vychází z předpokladu, že paleta barev¹⁶ obsahuje nejvýše 256 barev (2, 4, 8, 16, 32, 64, 128 nebo 256). Při převodu grafiky s barevnou hloubkou *True Color* (24 bitů) do tohoto formátu s podporou 8 bitů proto dochází ke ztrátě informací¹⁷.

Redukce barevné palety se však často vyplácí, protože umožňuje zmenšit velikost grafických souborů. Jak si můžeme všimnout, internetové stránky ve formátu HTML často odkazují právě na grafické soubory *GIF*.

Komprese formátu *GIF* využívá algoritmu *LZW*. Algoritmus v obraze vyhledává sekvence pixelů, např. posloupnosti bodů stejné barvy. Tyto sekvence nahrazuje vhodnými číselnými kódy, které jsou uloženy v poli. Čím více takových opakovaných sekvencí existuje, tím lepšího výsledku komprese lze dosáhnout. Soubory jsou pak menší, ale úspora místa samozřejmě značně závisí na struktuře obrázku.

Formát souboru *GIF* popisuje tabulka 13.5. Kromě samotných obrazových dat (komprimovaných) formát zahrnuje také všechny dodatečné informace, které jsou potřeba ke zobrazení na fyzickém zařízení (např. na displeji).

Tabulka 13.5: Formát souboru *GIF*

Blok	Obsah
Hlavnička	6 bajtů, „GIF87a“ nebo „GIF89a“
Popis logické obrazovky (popisovač obrazovky) – 7 bajtů	
Šířka obrazovky	2 bajty
Výška obrazovky	2 bajty
Indikátor globální mapy barev	1 bit (1 – je, 0 – není)

¹⁶ Paleta je z hlediska počítače tabulkou kódů, která umožňuje interpretovat znakový kód v grafickém souboru jako barvu viditelnou na obrazovce.

¹⁷ Při konverzi z formátu s barevnou hloubkou *True Color* do formátu *GIF* probíhá vzorkování barev z větší barevné palety a vzniká menší paleta.

Blok	Obsah
Rozlišení barev	3 bity
Značky třídění barev v mapě	1 bit (hodnota 1 znamená rostoucí třídění, nejčastěji se vyskytující barva na prvním místě)
Velikost globální mapy barev	3 bity, velikost = (2^{n+1})
Barva pozadí	1 bajt – index barvy pozadí v globální mapě barev
Koeficient tvaru pixelů	1 bajt, 0 – čtvercové pixely, pokud je hodnota nenulová, pak podíl šířky a výšky závisí na poměru $(n+15)/64$
Globální mapa barev	(volitelné), každá následující barva je kódována pomocí třech složek RGB
Blok popisovače obrazu	
Oddělovač	1 bajt (znak čárky, 0x2c)
Vzdálenost od levé strany obrazu	unsigned (2 bajty)
Vzdálenost od horního okraje obrazu	unsigned (2 bajty)
Šířka	unsigned (2 bajty)
Výška	unsigned (2 bajty)
Indikátor lokální mapy barev	1 bit (1 – ano, 0 – ne)
Indikátor prokládání	1 bit (1 – ano, 0 – ne)
Indikátor třídění barev v mapě	1 bit
<vyhrazeno>	2 bity
Velikost lokální mapy barev	3 bity, velikost = $(2n+1)$
Lokální mapa barev (přepisuje globální mapu, týká se následujícího obrázku)	
Data pixelů	Jednotlivý obraz – jeden soubor GIF může obsahovat od 1 do n obrazů
...	
Indikátor konce	

Měli bychom ještě vysvětlit pojem „prokládání obrazu“. Po sobě následující pixely se zapisují v pořadí zleva doprava a shora dolů. Pokud mají uživatelé pomalejší připojení k Internetu, můžeme využít optického triku. Spočívá v tom, že do obrazu postupně doplňujeme jednotlivé chybějící řádky. Přitom využíváme prokládání (změnu pořadí) zápisu řádků podle tohoto schématu:

- Při *prvním* průchodu zapisujeme každý osmý řádek počínaje nulovým.
- Při *druhém* průchodu zapisujeme každý osmý řádek počínaje čtvrtým.
- Při *třetím* průchodu zapisujeme každý čtvrtý řádek počínaje druhým.
- Při *čtvrtém* průchodu zapisujeme každý druhý řádek počínaje prvním.

Soubory s prokládaným zápisem se mohou zobrazit v relativně čitelné podobě už po načtení jedné osminy dat!

První bajt v komprimované podobě obrazu určuje minimální počet b bitů na pixel v původním obrazu. Hodnota 2^b je tzv. čisticí kód (ang. *clear code*), který umožňuje nastavit výchozí hodnoty

KAPITOLA 13 Kódování a komprese dat

všech parametrů komprese a dekomprese. Počáteční velikost slovníku se rovná 2^{b+1} . Velikost se v případě potřeby zdvojnásobuje, dokud není dosažena maximální hodnota 4096.

Kódová slova generovaná algoritmem *LZW* jsou v souboru *GIF* uložena ve formě znakových bloků délky 8 bitů. Maximální rozměr bloku se rovná 255 – na začátku každého bloku se nachází hlavička, která obsahuje informace o jeho velikosti. Pokud má bajt délky podbloku hodnotu 0x00, označuje to konec datového bloku (tzv. *terminátor bloku*).



Poznámka: Podrobný popis formátu *GIF* naleznete v souboru *Gif89a.txt*, který je součástí archivu ke stažení.

KAPITOLA 14

Různé úlohy

V této kapitole:

- Texty úloh
- Řešení úloh

V této kapitole jsou umístěny úlohy, které se nevešly do předchozích kapitol. Jsou to prostá programátorská cvičení na různá atraktivní téma. Při jejich řešení si můžete ověřit, zda dokážete efektivně zvládnout každodenní programátorské úkoly. Některá zadání neobsahují řešení, ale vzhledem k jejich jednoduchosti by to nemělo vadit.

Texty úloh

Úloha 1

Tzv. *Eratosthenovo síto* patří ke starším metodám hledání prvočísel (tzn. čísel, která jsou dělitelná pouze sebou samými a číslem 1). Algoritmus spočívá na následující redukci čísel:

vypíšeme posloupnost přirozených čísel:

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ..., N
odstraníme z ní čísla větší než 2 a dělitelná číslem 2;
- 1, 2, 3, *, 5, *, 7, *, 9, *, 11, *, 13, *, 15... N
odstraníme z ní čísla větší než 3 a dělitelná číslem 3;
- 1, 2, 3, *, 5, *, 7, *, *, 11, *, 13, *, ... N
odstraníme z ní násobky čísla 5 (další nejmenší prvočíslo), 7...

Napište program, který:

- metodou založenou na *hrubé síle* zkонтroluje, zda dané číslo patří mezi prvočísla.
- při využití *Eratosthenova síta* najde všechna prvočísla menší než 100.

Úloha 2

Napište funkci, která na vstupu přijímá datum zakódované jako celé číslo (např. 120461) a vypíše slovně jeho význam (zde: „12. dubna 1961“).

Úloha 3

V operacích s maticemi se často vyskytují pole s mnoha nulovými prvky. Jejich dvourozměrná reprezentace není z paměťového hlediska příliš efektivní. Zkuste navrhнуть datovou strukturu, která bude obsahovat pouze informace o „souřadných“ nenulových prvcích. Předpokládáme přitom, že všechny zbývající hodnoty, které struktura nezahrnuje, jsou nulové. Navrhňte funkce, které budou takovou datovou strukturu obsluhovat: vypisovat matici v dekódované formě, sčítat a násobit dvě matice atd.

KAPITOLA 14 Různé úlohy

Zkuste přibližně odhadnout, do jakého stupně zaplnění pole nulami se taková datová struktura vyplatí z hlediska využití paměti.

Úloha 4

Navrhněte dvě verze rekurzivního algoritmu na výpočet funkce x^n (s „přirozenou“ rekurzí a rekurzí „s pomocným parametrem“).

Úloha 5

Zkuste vytvořit *nerekurzivní* funkci, která na základě dvou setříděných seznamů vrátí jako výsledek setříděný seznam, který bude obsahovat všechny prvky původních seznamů. Požadavek: Nelze vytvářet nové paměťové buňky, povolena je pouze manipulace s ukazateli.

Úloha 6

Napište funkci, která přijímá cenu uvedenou v podobě celého čísla (např. typu `long`) a vypíše ji ve slovní podobě. Příklad: Volání `cena_slovne(12304)` by mělo zobrazit text: „dvanáct tisíc tři sta čtyři Kč“.

Úloha 7

Napište program, který provede cyklickou permutaci daného vstupního pole o zadáný počet pozic. Pokuste se problém podrobně analyzovat a vybrat programovací techniku: iterativní či rekurzivní řešení – v druhém případě také jeho typ.

Úloha 8

Napište program, který bude nejjednodušším způsobem počítat, kolikrát se dané slovo vyskytuje ve vstupním textu.

Úloha 9

Napište program, který bude nejjednodušším způsobem zjišťovat statistické údaje o vstupním textu: počet výskytů každého písmene, slova atd.

Úloha 10

Napište program, který bude kontrolovat, zda je vstupní věta palindromem (tzn. zda se dá přečíst stejným způsobem zleva doprava jako zprava doleva). Příklad takové věty: „jelenovi pivo nelej“.

Úloha 11

Napište program, který ověří, zda vstupní věta obsahuje skryté slovo, např. text „Franta ochotně nese tašku“ ukrývá slovo „fronta“.

Úloha 12

Napište funkci, která vypočítá výrazy ve tvaru: $2+2+1$, $1+2*3$ atd., které jsou přístupné pomocí ukazatele typu `char*`. (Knihovní funkce jazyka C++, která nahrazuje znakovou posloupnost číslem s plovoucí řádovou čárkou, se nazývá `atof`, ale její činnost se zastaví na prvním nečíselném znaku textu.)

Řešení úloh

Úloha 1

Při řešení úlohy (a) budeme potřebovat funkci, která vrátí první celé číslo x splňující podmínu $x^2 < n$:

```
erastot.cpp
inline int sqrt_int(int n)
{
    return (int)sqrt((double)n)/1;
}
```

(Využíváme toho, že celočíselné dělení v jazyce C++ poskytuje výsledek bez desetinné části.)

Chceme-li odpovědět na otázku, zda n je prvočíslo, stačí zkонтrolovat, zda skutečně nemá jiné dělitele než číslo 1 a sebe samo:

```
bool prvocislo(int n) // je n prvočíslo?
{
    int i, limes=sqrt_int(n);
    for(i=2; n!=(n/i)*i && i<=limes; i++)
        if (i>limes)
            return true; // ano, jde o prvočíslo
        else
            return false; // ne, je to "obyčejné" číslo
}
```

Poněkud komplikovanější je implementace *Eratosthenova síta* (b).

Musíme bohužel deklarovat velká pole, ale to je jediným nedostatkem této prosté metody:

```
void sito(int n) // vypsání všech prvočísel < n
{
    int cpt = 1, i, *tp=new int[n+1];

    for(i=1; i<=n; i++)
        p[i]=i; // označení všech přirozených čísel od 1 do n

    while(cpt<n)
    { // hledání prvního nenulového prvku pole tp:
        for(cpt++; (tp[cpt]==0) && (cpt!=100); cpt++);
            int k=2; // nulování násobků tohoto prvku (cpt) v poli tp
        while(cpt*k<=n)
        {
            tp[cpt*k]=0;
            k++;
        }
    }
    for(i=1;i<=n;i++)
        if (tp[i]!=0) cout << "Prvočíslo:" << tp[i] << endl;
        delete tp; // odstranění paměťového pole
}
```

Úloha 3

Datová struktura na obsluhu „pole s nulami“ může mít formu následujícího seznamu:

```
struct nulovy_seznam
{
    int x,y;
    int val; // nebo libovolný jiný datový typ
    struct nulovy_seznam *dalsi;
}
```

Za předpokladu, že ukazatel `dalsi` zabírá v paměti dva bajty (podobně jako proměnné typu `int`), libovolný prvek seznamu obsadí $p = 2+2+2+2 = 8$ bajtů paměti. Na druhou stranu zvolíme-li klasické pole, budou jednotlivé prvky pole vyžadovat pouze 2 bajty (jedná se o proměnnou typu `int`), avšak musíme předem přidělit paměť na celé pole.

Označme velikost pole písmenem N . K uložení celého pole do paměti potřebujeme $2N^2$ bajtů. „Magickou“ hranici k , za kterou přestává mít nasazení seznamů smysl, můžeme snadno vypočítat pomocí rovnice: $k \cdot p = 2N^2$. Například pro $p = 8$ a $N = 10$ je tento limit překročen již u dvacátého šestého nenulového prvku. Z praktického pohledu by nenulových prvků mělo být *mnohem* méně než $2N^2/p$ – o přesném významu slova „mnohem“ zde rozhoduje programátor.

Úloha 4

Následují dvě verze rekurzivních programů, které umožňují vypočítat hodnotu x^n :

```
pot.cpp
int moc1(int x, int n)
{
    if (n==0)
        return 1;
    else
        return (moc1(x,n-1)*x);
}

int moc2(int x,int n,int temp=1)
{
    if (n==0)
        return temp;
    else
        return (moc2(x,n-1,temp*x));
}

int main()
{
    cout << "Dvě na třetí\n";
    cout << "Metoda 1\t" << moc1(2,3)<< "\n";
    cout << "Metoda 2\t" << moc2(2,3)<< "\n";
}
```

Úloha 10

Úloha patří mezi elementární, takže bychom měli bez problémů dojít k následujícímu řešení:

```
palindro.cpp
void palindrom(char *s)
{
    int dl=strlen(s), cpt=0;
    bool test=true; // 's' je (zatím) palindrom
    while( (cpt<=dl/2) && (test==true) )
        if(s[cpt]==s[dl-cpt-1])
            cpt++;
        else
            test=false;
    cout << s;
    if(test==true)
        cout << "...je palindrom\n";
    else
        cout << "...je obyčejné slovo...\n";
}
```

Úloha 12

S úkolem vypočítat hodnoty výrazů zapsaných ve slovním tvaru se programátoři v praxi setkávají dosti často. Zadání je obecně dosti náročné a postup bychom měli dobře promyslet. Řešení, které je uvedeno níže, bychom tedy měli brát spíše jako příspěvek do diskuze než jako hotový vzor. Tím spíše, že pro jisté konfigurace vstupních dat jsou výrazy vypočítány špatně!

```
oblicz.cpp
double transl(char *s)
// nahrazuje posloupnosti znaků typu 1+1 či 1+1+2*5 hodnotou (zde 2, resp. 12)
// pozor: funkce neanalyzuje dělení a nekontroluje případ
// dělení nulou!
{
    int i,n;
    char *s1;
    n=strlen(s);
    s1= new char[n+1]; // pracovní kopie vstupního textu
    strcpy(s1,s); // kopie vstupní posloupnosti

    for(i=0;i<n;i++) // hledání znaků + a *
        if(s[i]=='+'||s[i]=='*')
        {
            s1[i]='\0';
            if(s[i]=='+')
                return transl(s1)+transl(s+i+1);
            else
                return transl(s1)*transl(s+i+1);
        }
    // elementární případ:
    delete s1;
    return atof(s); // atof= "ascii to float"
}
int main()
{
    cout << "1+1=<< transl("1+1")           << endl;      // 2   OK
```

KAPITOLA 14 Různé úlohy

```
cout << "2*2*3=" << transl("2*2*3")      << endl;    // 12 OK
cout << "2+2*3=" << transl("2+2*3")      << endl;    // 8 OK
cout << "2+2+3=" << transl("2+2+3")      << endl;    // 7 OK
cout << "2+2*0=" << transl("2+2*0")      << endl;    // 2 OK
cout << "2*3+4*5=" << transl("2*3+4*5") << endl;    // 46 špatně!
}
```

Zamyslete se nad tím, proč funkce `transl` nesprávně vypočítala poslední výraz. (Návod: Postupujte podle směru analýzy výrazu.)

Pro pokročilé programátory v C++: Analyzujte, jakým způsobem funkce `transl` pracuje s pamětí. Je v tomto případě skutečně vhodné používat klíčová slova `new` a `delete`?

PŘÍLOHA A

Seznámení s jazykem C++

Tato příloha by měla posloužit jako pomůcka programátorům v jazyce Pascal (Delphi), kteří se chtějí rychle a bez námahy seznámit se základními prvky jazyka C++, aby jim syntaktické odlišnosti obou jazyků nekomplikovaly čtení kódů v knize. Tento materiál je záměrně umístěn v příloze, protože nepatří k probírané tematice.

Příloha samozřejmě nenahradí specializovanou příručku věnovanou jazyku C++, ale to ani není jejím cílem. Následující rychlokurz jazyka C++ zahrnuje pouze prvky, které jsou nezbytné k pochopení prezentovaných výpisů. Jedná se o naprostě minimální úvod, který se zaměřuje výhradně na syntaxi. Vážní zájemci o programování v jazyce C++ by si měli pořídit dobrého průvodce, např. [Eck00] nebo [Str97].

V této kapitole:

- Prvky jazyka C++ na příkladech
- Podprogramy
- Aritmetické operace
- Odkazy
- Složené typy
- Iterace
- Rekurzivní struktury
- Parametry programu main()
- Operace se soubory v jazyce C++
- Objektové programování v jazyce C++
- Podmíněný kód v jazyce C++

Prvky jazyka C++ na příkladech

Na dalších stranách se nachází řada příkladů, které programátorům zvyklým na jazyk Pascal umožňují, aby se na základě analogie seznámili se základními pravidly zápisu algoritmů v jazyce C++.

První program

Podívejme se na následující program, který patří do dobře známé kategorie `hello world`:

```
program pr1; {komentář}
begin
  writeln('Ahoj lidi!')
end.
```

```
#include <iostream>
using namespace std;
int main() // komentář
{
    cout << "Ahoj lidi!\n";
}
```

- Blok v C++ je uzavřen do složených závorek { }.
- Činnost programu začíná od funkce s názvem `main`. V jazyce C++ tato funkce vždy vraci hodnoty typu `int`, a proto je nutné před název funkce uvést právě tento typ. Ve starším kódu jsme se na tomto místě mohli občas setkat s typem `void`. Ten označuje funkce, které nevracejí hodnoty v aritmetickém smyslu.
- Chceme-li v C++ vypsat nějaký text, můžeme na standardní výstup (`cout`) odeslat posloupnost znaků uzavřenou do dvojitých uvozovek („text“). Sekvence \x označuje speciální znak, např. \n znamená přechod na další řádek při vypisování textu na obrazovce, \t – znak tabulátoru atd.
- Komentář // v jazyce C++ platí do konce řádku. Chceme-li napsat delší komentář na několik řádků, můžeme místo zápisu // komentář použít raději /* komentář */.

Direktiva #include

Symbol # v kódu C++ označuje direktivy neboli příkazy pro kompilátor. Jazyk C++ poskytuje pokročilé manipulace s obsahem zdrojového kódu, ale většině programátorů zpočátku stačí vědět, že musí použít direktivu `#include <iostream>`, která znamená, že ještě před samotnou kompliací je k souboru s programem připojen celý obsah souboru `iostream`. Tento soubor je nezbytný, chceme-li pracovat s datovými proudy `cout`, `cin` a `cerr`, které odpovídají standardnímu výstupu (např. na obrazovku), vstupu (např. z klávesnice) a nakonec místu, kam program odesílá zprávy o chybách. Obvykle se chyby vypisují na obrazovku.

Doprovodná direktiva: `using namespace std` zajistí zpřístupnění deklarace a názvů, které jsou součástí standardní knihovny C++. Znalosti principů „oborů názvů“ lze využít na poněkud pokročilejší úrovni programování v C++, která přesahuje jednoduché algoritmy obsažené v této knize. Zatím stačí vědět, že bez uvedené instrukce programy nebudou fungovat.



Poznámka: V dalších příkladech budeme obě direktivy kvůli úspoře místa vynechávat, ale chceme-li program spustit, nesmíme na tyto direktivy zapomenout. Je také dobré vědět, že starší programy jazyka C++ obsahovaly pouze instrukci `include <iostream.h>`, ale současné kompilátory už tuto formu nepřijímají.

V připojených souborech se obvykle nacházejí deklarace často používaných konstant a typů. Připojený soubor uvedený ve špičatých závorkách <> patří mezi standardní soubory z knihovny kompilátoru. Soubory, jejichž název je uzavřen do dvojitých uvozovek "", vytváří sám programátor. Obvykle jsou uloženy ve stejně složce jako soubory s kódem C++.

Podprogramy

V jazyce C++ se podobně jako v klasickém jazyce C všechny podprogramy označují jako funkce. Protějškem *procedury* známé z jazyka Pascal je funkce, která nevrací žádnou hodnotu – uvádíme ji klíčovým slovem `void`.

Procedure

Následuje příklad definice a použití procedur v jazycích Pascal a C++:

```
program pr9;                                     void proc1(int a,
procedure proc1(a,b:integer;                  int b,
  var m:integer                                int& m.
  )                                              )
{  lokální proměnná:}                         { // lokální proměnná:
var c:integer;                                 int c;
begin                                         c=a+b;
  c:=a+b;                                     cout << c << endl;
  writeln(c);                               m=c*a*b;
  m:=c*a*b                                 }
end;                                         int i,j,k;
var i,j,k:integer;                           int main()
begin                                         {
  i:=10;                                       i=10;
  j:=20;                                       j=20;
  proc1(i,j,k)                             proc1(i,j,k);
end.                                         }
```

- Jazyk C++ neumožnuje vytvářet lokální procedury a funkce.
- Definované funkce a procedury jsou obecně dostupné v celém programu.
- Deklaraci typu var v hlavičce funkce odpovídá v jazyce C++ v zásadě tzv. odkaz (&), např. zápis Fun(var i:integer;...) je funkčně ekvivalentní s formou Fun(int& i,...).
- Na pole jazyka C++ z principu odkazujeme pomocí adresy. Například zápis Fun(int tab[3]) znamená, že chceme jako vstupní parametr použít pole prvků typu int. Při volání funkce Fun předáváme pole použité jako parametr pomocí jeho adresy a obsah pole se může fyzicky změnit (viz také stranu 324).

Funkce

Zásadní rozdíl mezi funkcemi v jazycích C++ a Pascal se týká způsobu, jakým vracejí hodnotu:

```
program pr10;                                     int plus2(int a)
function plus2(a:integer):integer;           {
begin                                             return a+2;
  plus2:=a+2                                }
end;                                         int i;
var i:integer;                               int main()
begin                                         {
  i:=10;                                       i=10;
  writeln(plus2(i))                         cout<<plus2(i)<<endl;
end.                                         }
```

PŘÍLOHA A Seznámení s jazykem C++

- Instrukce `return(v)` v jazyce C++ způsobí okamžitý návrat z funkce s hodnotou `v`. Například po instrukci `if(v) return(val)` není nutné použít instrukci `else1` – pokud je podmínka v splněna, případná další část procedury se již neproveze.
- K dobrým programátorským návykům patří používání tzv. *hlaviček funkcí*, které komplilátor informují o typu funkce a jejích vstupních a výstupních parametrů. Hlavička funkce je to, co z funkce zůstane, když odstraníme její definici a *názvy* vstupních parametrů. Například je-li někde v programu definována funkce:

```
void f(int k, char* s[3]) { ... následuje kód ... },
```

můžeme bezprostředně za direktivy `#include` doplnit rádek:

```
void f(int, char*[]); // na konci středník!
```

Hlavičkové deklarace se často sdružují do souborů typu `.h`, které jsou připojeny direktivou `#include`. Tento postup se uplatňuje zvláště u definic tříd (viz stranu 330). Hlavičky jsou velmi důležité, protože již ve fázi předběžné komplikace umožňují odstranit mnoho chyb, k nimž dochází při volání funkce se špatnými parametry. Některé komplilátory mimoře už z principu chybějící hlavičky netolerují a neumožní zkompilovat kód s neznámou funkcí (tj. s takovou funkcí, jejíž definice nebo deklarace hlavičky nebyla nalezena při prvním „průchodu“ kompilačního procesu).

Uvedeme příklad chybného (bez použití hlavičky funkce) a správného kódu C++:

```
// chybný kód:  
void fun() // definice  
{...} //  
int main()  
{  
    fun();  
    fun2(5); // komplilátor nezná funkci 'fun2'!  
}  
void fun2(int a) // definice  
{...}
```

```
// správný kód:  
void fun2(int a); // hlavička  
void fun()  
{...}  
int main()  
{  
    fun();  
    fun2(5);  
}  
void fun2(int a) // definice  
{...}
```

Aritmetické operace

Nevelké rozdíly najdeme u jistých operátorů, které se v jazyce Pascal nazývají poněkud odlišně než v C++. U programů v jazyce C++ nás na první pohled může zarazit, že v zápisech aritmetických operací obsahují mnoho zkrátek. Tyto zkratky zdánlivě zhoršují čitelnost, ale když se seznámíme se syntaxí jazyka, tento dojem se brzy ztratí. Máme zde na mysli hlavně operátory `++`, `--` a celou skupinu výrazů typu:

`proměnná OPERÁTOR = výraz;`

Je potřeba zdůraznit, že tyto formy sice nejsou povinné, nicméně jsou vhodné – výsledný kód programu je díky nim o něco efektivnější.

1 Není to samozřejmě zakázané.

```

const pi=3.14;
program pr2;
var a,b,c:integer;{globální}
begin
    a:=1;
    b:=1;
    a:=a+1; {inkrementace}
    b:=b-2
end.

```

```

const float pi=3.14;
// nebo double kvůli vyšší přesnosti
int a,b,c;
int main()
{
    a=1;
    b=1;
    a++; // inkrementace
    b-=2; // STŘEDNÍK!
}

```

- Proměnné lze v jazyce C++ deklarovat na libovolném místě. Můžeme to provést *před* některými instrukcemi, za nimi i *v jejich těle*.
- K přiřazení hodnoty proměnné slouží symbol =, nikoli :=.
- Funkcí **div** a **mod**, které znají programátoři v jazyce Pascal, odpovídají v jazyce C++ operátory /, resp. %.

Všimněme si nyní operátorů zvětšení o 1 a zmenšení o 1 (++ , --), které se v C++ používají velmi často. Jestliže je uvedeme *prefixově*, uplatní se jako první, zatímco v případě *postfixového* zápisu má prioritu příslušný výraz.

Příklad:

```

int a = 2;
int b = 5;
int n = a + b++; // n = 7 (prioritu má sčítání)
cout << n << endl; // vypíše 7
cout << b << endl; // vypíše 6
cout << a + ++b << endl; // vypíše 9 (prioritu má inkrementace)

```

- Zápis proměnná *op* = výraz odpovídá klasickému zápisu:

proměnná = *proměnná op výraz*,

kde *op* označuje jistý operátor přijímající dva argumenty.

Logické operace

Podobně jako aritmetické operace mají také logické operace svá specifika. Naštěstí jich není příliš mnoho. Programátoři v jazyce Pascal by měli věnovat zvláštní pozornost rozdílům mezi operátorem = v jazyce Pascal a operátorem == v C++. Kompilátor bohužel neohlásí chybu, pokud se v jazyce C++ pokusíme zkompilovat instrukci: **if** (a=1) a=a-3 místo instrukce **if** (a==1) a=a-3.

Příklad správných logických instrukcí:

```

Program pr3;
var a:boolean;
begin
    a:=true;
    if a=true then
        writeln('true')
    else
        writeln('false')
end.

```

```

int main()
{
    bool a;
    a=(2>3);
    if (a==true)
        cout << "Pravda!\n";
    else
        cout << "Nepravda!\n";
}

```

PŘÍLOHA A Seznámení s jazykem C++

Výrazy `bool`, `true` a `false` patří mezi klíčová slova jazyka C++.

Všimněme si, jakou roli plní v jazyce C++ středník, který označuje *konec* dané instrukce. Z toho-důvodu je nutné středníkem uzavřít dokonce i instrukci, která se nachází před instrukcí `else!` Některé logické operátory používané ve výrazech se v obou jazycích liší. Jejich souhrn obsahuje tabulka A.1.

Tabulka A.1: Porovnání operátorů v jazycích Pascal a C++

Pascal	C++
=	==
not	!
<>	!=
OR	
AND	&&

Ukazatele a dynamické proměnné

Jazyk C++ umožnuje uplatňovat různá paradigmata programování s vysokou úrovní abstrakce (patří přece mezi tzv. strukturální jazyky) a zároveň poskytuje možnosti, které jej přibližují k jazyku symbolických adres (assembler). Naučíme-li se využívat obou aspektů jazyka, dokážeme snadno programovat efektivní aplikace. Dynamické proměnné, adresy a ukazatele představují klíč k dobrému zvládnutí jazyka C++ a musíme se s nimi naučit pracovat. Následující příklad ukazuje, jakým způsobem se tvoří dynamické proměnné a jak lze s nimi operovat.

```
program pr4;
type example=^real;
var p:example;
begin
  new(p);
  p^:=3.13;
  dispose(p)
end.
int main()
{
    float *p, q=3.14,
    p=new float;
    *p=q;
    delete p;
}
```

- Operace s ukazateli (na adresách) se v jazyce C++ neomezují na dynamické proměnné.
- Chceme-li zjistit *hodnotu*, na kterou ukazatlová proměnná odkazuje, musíme před ni uvést symbol hvězdičky (například `p` obsahuje adresu a `*p` příslušnou hodnotu – samozřejmě za předpokladu, že jsme proměnnou `p` dříve inicializovali).
- Operátor `&` zjišťuje adresu proměnné. Je tedy patrné, že se jedná o protějšek operátoru `*` (např. výraz `int *p=&m` zajistí, že proměnná `p` bude odkazovat na paměťovou adresu, kde je uložena proměnná `m`).
- Chceme-li zjistit adresu libovolné proměnné jazyka C++, stačí před název proměnné uvést operátor `&`.

Příklad:

```
int k=12;
int *ukz=&k;
cout << *ukz << endl; // program vypíše 12
```

- Jazyk C++ při dodržení syntaktických pravidel nijak neomezuje operace s adresami, ukazateli a proměnnými, dynamickým přidělováním paměti atd.

Odkazy

Programátorům, kteří zatím jazyk C++ dostatečně neznají, lze doporučit, aby se seznámili s pojmem *odkazu*. Uvedený mechanizmus sice v této knize nepoužíváme, ale často se s ním v praxi můžeme setkat při předávání objektů pomocí jejich adresy, například funkcím.

Tento mechanizmus spočívá v tom, že funkce namísto lokální kopie proměnné dostane její *adresu*. Všechny operace s odkazovanou proměnnou pak nebudou modifikovat lokální kopii, ale originál.

Podívejme se na příklad volání funkce pomocí odkazu:

```
void fun(int& x)
{
    x=5;
}

int main()
{
    int m=6;
    fun(m);
    cout << m << endl;
}
```

Tento poněkud umělý příklad ukazuje, jak se přirozeně používají proměnné předávané odkazem – jediný syntaktický rozdíl spočívá v hlavičce funkce (znak &). Volání funkce fun změní hodnotu proměnné m a program zobrazí číslo 5. Odkazy jsou výhodné, protože díky nim není nutné pracovat s operátorem zjišťování hodnoty ukazatele (*), který pochází z jazyka C.

Složené typy

Jazyk C++ zahrnuje mnoho jednoduchých i složených typů, které dobře známe z jiných strukturálních jazyků. Patří k nim mimo jiné pole a záznamy. V porovnání s jazykem Pascal se zdá, že jazyk C++ neposkytuje tolik možností. Základní omezení polí souvisí s rozsahem indexů: vždy začínají od nuly. Nelze ani deklarovat záznamy „s variantami“. Tyto nevýhody je samozřejmě možné obejít, ale nikoli přímo.

Pole

Indexy v polích deklarovaných v jazyce C++ vždy začínají od nuly. Deklarujeme-li tedy pole t délky 4, ve skutečnosti vytváříme 4 proměnné: t[0], t[1], t[2] a t[3]. Pokud chceme zajistit, aby se indexy v programech jazyků Pascal a C++ rovnaly, musíme přitom jejich převod zajistit sami.

<pre>Program pr5; type tab=array[3..5] of integer; var t:tab; begin t[3]:=11; t[4]:=t[3]+1; end.</pre>	<pre>typedef int tab[3]; tab t; int main() { t[0]=11; *(t+1)=t[0]+1; }</pre>
---	--

PŘÍLOHA A Seznámení s jazykem C++

- Jazyk C++ v zásadě nekontroluje překročení hranic pole, k jehož prvkům přistupujeme pomocí indexů, a spolehlá přitom na programátora. Tuto nevýhodu můžeme ošetřit pomocí objektových mechanizmů, ale při práci se základními nástroji jazyka musíme prostě hlídat, abychom náhodou nešlápli vedle².
- Název pole v C++ zároveň slouží jako ukazatel na něj. Například: `t` odkazuje na první prvek pole a `(t+3)` na čtvrtý. Zápis `*(t+1)` odpovídá zápisu `t[1]`.
- Deklarace `int *x` je ekvivalentní deklaraci `int x[]`.

Záznamy

Základní operace se záznamy můžeme předvést na jednoduchém příkladu:

```
program pr6;
type Bunka=
  record
    znak:char;
    a,b,d:integer;
  end;
var x:Bunka;
begin
  x.znak:= 'a';
  x.a:=1
end.
```

```
struct Bunka
{
  char znak;
  int a,b,d;
};

Bunka x;
int main()
{
  x.znak= 'a';
  x.a=1;
}
```

- V jazyce C++ se záznamy označují jako *struktury*. Přistupovat k nim můžeme podobným způsobem jako v případě jazyka Pascal (pomocí tečkové notace).
- Záznam s variantami nelze deklarovat přímo.
- Podobně jako v jazyce Pascal lze vložit pole do záznamu a naopak.
- Položka `nazev_položky` ukazatele nebo dynamického záznamu, na kterou odkazuje proměnná `y`, není přístupná výrazem `y.nazev_položky`, ale pomocí konstrukce: `x->nazev_položky`.
Příklad:

```
Bunka x, *y; // proměnná typu Bunka a ukazatel na ni
x.znak='a';
y=&x;          // zjištění adresy proměnné x
y->a=1;        // tečková notace byla chybná!
```

Příkaz switch

Příkaz `switch` v jazyce C++ se od svého protějšku v jazyce Pascal liší v několika zrádných detailech. Uvedený příklad je proto potřeba analyzovat pozorně.

Nejdůležitější informace, kterou bychom si měli zapamatovat, souvisí s klíčovým slovem `break` (*přerušení*). Pokud bychom toto slovo vynechali, program by pokračoval v provádění následujících instrukcí, dokud by nenarazil na jiný výskyt slova `break` nebo na konec příkazu `switch`.

² V tomto případě bychom se mohli dostat na „špatnou adresu“.

```

program pr7;
var w:integer;
begin
  w:=2;
  case w of
    1: writeln('1');
    2: writeln('2');
    otherwise:
      writeln('?');
  end
end.

```

```

int w;
int main()
{
  w=2;
  switch(w)
  {
    case 1:cout<< "1\n";break;
    case 2:cout<< "2\n";break;
    default:
      cout<<"?\n"; break;
  }
}

```

- Klíčové slovo **break** v jazyce C++ slouží jako oddělovač případů.

Iterace

Iterační příkazy jsou v obou jazycích podobné:

```

program pr8;
var i,j:integer;
begin
  j:=1;
  for i:=1 to 5 do
    begin
      writeln(i*j);
      j:=j+1
    end;
  i:=1;
  j:=10;
  while j>i do
    begin
      i:=i+1;
      writeln(i)
    end
end.

```

```

int i,j;
int main()
{
  j=1;
  for(i=1;i<=5;i++)
  {
    cout << i*j << endl;
    j++;
  }
  i=1;
  j=10;
  while (j>i++)
    cout << i << endl;
}

```

- Klíčové slovo **endl** označuje přechod na nový řádek.
- Zde neuvedený příkaz **do{... }while(v)** se v jazyce C++ provádí tak dlouho, dokud je výraz v různý od nuly³.
- Jednotlivé prvky příkazu **for(e1; e2; e3)** mají tento význam:
 - **e1**: iniciace cyklu,
 - **e2**: podmínek provádění cyklu,
 - **e3**: modifikátor řídící proměnné (může se jednat o funkci nebo skupinu příkazů oddělených čárkou – pak se provádějí zleva doprava).

Příklad:

```
for(int i=6; i<100; Insert(tab[i++]), Pis(i));
(Pis a Insert jsou funkce, tab označuje jisté pole.)
```

³ Srovnejte např. s cyklem repeat... until.

Rekurzivní struktury

Následující příklad předvádí, jakým způsobem lze deklarovat rekurzivní datové struktury (struktury, které odkazují na sebe samy).

```
Program pr11;
type ukz=^prvek;
prvek=record
  hodnota : integer;
  dalsi:ukz
end;
var p:ukz;
begin
  new(p);
  read(p^.hodnota);
  p^.hodnota=nil
end.
```

```
typedef struct x
{
  int hodnota;
  struct x* dalsi;
} PRVEK;

int main()
{
  PRVEK *p;
  p=new PRVEK;
  cin >> p->hodnota;
  p->dalsi=NULL;
}
```

- Klíčovému slovu **nil** odpovídá v jazyce C++ slovo **NULL**.

Parametry programu main()

Programu v C++ občas při spuštění potřebujeme předat vstupní parametry, např. možnosti volání, názvy souborů se zdrojovými daty atp. Parametry volání jsou znakové řetězce a můžeme se k nim snadno dostat, ačkoli syntaxe pocházející ještě z jazyka C je poněkud bizarní.

Následující program vypisuje parametry předané při spuštění (zkompilovaného spustitelného kódů) na příkazovém rádku:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
  for (int i=0; i< argc; i++)
    cout << "Parametr č. " << i << ":" << argv[i] << endl;
}
```

Parametr **argc** je čítačem parametrů, hodnota **argv[0]** udává název samotného spustitelného programu a poté případně následují běžné parametry volání.

Operace se soubory v jazyce C++

Operace se soubory v jazyce C++ představují oproti jazyku C úplnou hračku. Podívejme se na příklad programu, který načítá jistý vstupní soubor, kopíruje ho (řádek po řádku) do výstupního souboru a přitom vstupní soubor po jednotlivých znacích vypisuje spolu s kódy těchto znaků (desítkovými i šestnáctkovými). Program pracuje s datovými proudy, a proto se vyznačuje podobnou koncepcí jako jiné datové proudy, s nimiž jsme se již setkali (např. **cout**).

pliki.cpp

```
#include <string>
#include <iostream>
```

```

#include <iostream>
using namespace std;

int main()
{
    ifstream soubor_VST ("input.txt"); // vstupní soubor
    ifstream soubor_BIN ("input.txt"); // stejný vstupní soubor
    ofstream soubor_VYS ("output.txt");// výstupní soubor

    string s;

    while (getline(soubor_VST,s))      // kopírování řádek po řádku
        soubor_VYS << s << endl;       // souboru input.txt do souboru output.txt

    char c;                           // vypisování vstupního souboru
    // po jednotlivých znacích:
    while ( soubor_BIN.read(&c,1) )
        cout << "Znak: "<< c << ", dec:"<< dec << (int)c << ", hex:"
            << hex << (int)c << endl;
        cout << endl;
}

```

Uveďme příklad výstupu programu (vstupní soubor `input.txt` obsahoval dva řádky s řetězci: 123 a abA):

```

Znak: 1, dec:49, hex: 31
Znak: 2, dec:50, hex: 32
Znak: 3, dec:51, hex: 33
Znak:
, dec:10, hex: a
Znak: a, dec:97, hex: 61
Znak: b, dec:98, hex: 62
Znak: A, dec:65, hex: 41

```

atp.

Objektové programování v jazyce C++

Celá síla a půvab jazyka C++ nespočívá ve vlastnostech, které zdědil od svého předchůdce⁴, ale v nových možnostech, které vycházejí z objektového přístupu. Jazyk C++ zaujal více lidí, než kterýkoli jiný programovací jazyk v historii informatiky. Z někdejší módy se už pomalu stává standardní nástroj současnosti. Vzhledem k tomu, jak je tento jazyk efektivní, programátorům neobeznámeným s tímto jazykem hrozí, že zůstanou stát na místě, zatímco celý obor spěchá stále rychleji vpřed.

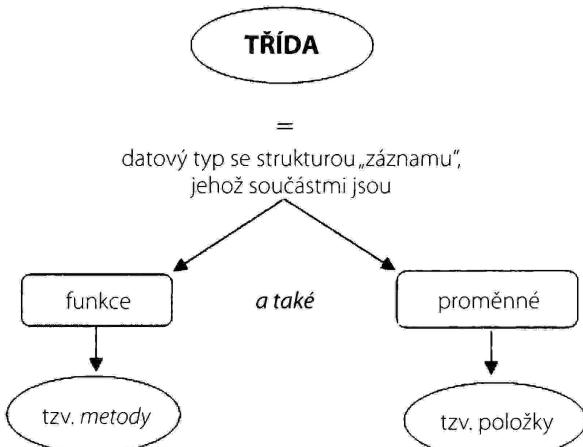
Jen pro formálnost připomeňme ještě upozornění z úvodu: Cílem této kapitoly je výhradně naučit programátory v jazyce Pascal, aby dokázali číst a porozumět výpisům kódu C++. Kvůli omezenému rozsahu knihy a značné šíři tématu se nemůžeme zmínit o všem. Citované příklady byly voleny s ohledem na svou reprezentativnost. Programátoři, kteří se o objektové programování v C++ více zajímají, si mohou své znalosti rozšířit například v publikacích [Poh99] či [Eck00].

⁴ Kterým je samozřejmě jazyk C.

Terminologie

Typické pojmy spojené s objektovým programováním jsou názorně shrnutы na obrázku A.1.

- Proměnná tohoto nového datového typu se nazývá *objekt*.
- *Metody* jsou běžné funkce nebo procedury pracující s položkami, které však patří do určité třídy⁵.



Obrázek A.1: Terminologie v objektovém programování

Existují dvě speciální metody:

- *Konstruktor*, který tvoří a inicializuje objekty (např. přiděluje potřebnou paměť, inicializuje požadovaným způsobem jisté položky atd.). V deklaraci třídy můžeme tuto metodu snadno poznat podle jejího názvu – ten se shoduje s názvem třídy. Konstruktor kromě toho nevrací žádnou hodnotu, dokonce ani v případě, že je jiného typu než void.
- *Destruktor*, který likviduje objekty (uvolňuje paměť, kterou zabíraly). Podobně jako konstruktor se také vyznačuje speciálním názvem, který je v tomto případě shodný s názvem třídy, ale začíná znakem vlnovky (~).

Každá metoda může pomocí příslušných názvů přistupovat k položkám objektu, který ji aktivoval. Jiný způsob přístupu je založen na ukazateli s názvem *this* (klíčové slovo jazyka C++), který odkazuje na samotný objekt. K atributu *x* lze tedy přistupovat buď pomocí výrazu *.x*, nebo *this->x*. Ukazatel *this* se však typicky používá v situacích, kdy metoda potřebuje jako výsledek vrátit předtím modifikovaný objekt (např.: *return *this;*).

Objekty na příkladech

Speciální datový typ třída svou konstrukcí připomíná záznam, který navíc dokáže volat funkce. Definice třídy se může skládat z několika sekcí, které se liší různou úrovní přístupu pro zbývající části programu. Obvykle se používají dva typy sekcí: *soukromé* a *veřejné*. Do soukromé části se obecně umisťují informace, které souvisejí s organizací dat (např. deklarace typů a proměnných) a veřejná část informuje o povolených operacích, které lze s daty provádět. Tyto operace mají samozřejmě formu funkcí neboli – abychom začali používat správnou terminologii – metod třídy.

Podívejme se, jakým způsobem lze deklarovat třídu, která dosti zjednodušeným způsobem obsluhuje tzv. *komplexní čísla*:

⁵ Tzn. mohou je využívat objekty dané třídy, ale jiné vnější funkce programu už nikoli!

```
complex.h
class Complex
{
    public: // začátek veřejné sekce
        Complex(double x,double y)
        // konstruktor třídy má stejný název jako sama třída
    {
        Re=x;
        Im=y;
    }
    void vypis(); // hlavička funkce, která vypisuje imaginární číslo
    double Cast_Real() // vrací reálnou část
    {
        return Re;
    }
    double Cast_Img () // vrací imaginární část
    {
        return Im;
    }
    // hlavička funkce, která předefinuje operátor + (plus), aby
    // bylo možné sčítat komplexní čísla:
    friend Complex& operator +(Complex,Complex);
    // hlavička funkce, která předefinuje operátor <<
    // aby bylo možné vypisovat komplexní čísla:
    friend ostream& operator << (ostream&,Complex);
    private: // začátek soukromé sekce
        double Re,Im; // reprezentace jako Re+j*Im
    }; // konec deklarace (a částečné definice)
        // třídy Complex
```

Konstrukce třídy Complex informuje o našich záměrech:

- Víme, že komplexní čísla se vnitřně skládají z reálné a imaginární části. Vzhledem k tomu, že způsob implementace třídy je její vnitřní záležitost, umisťujeme tyto informace do soukromé sekce, která se v našem případě omezuje na deklarace proměnných Re a Im.
- Z hlediska vnějšího pozorovatele (neboli prostě uživatele třídy) jsou komplexní čísla objekty, které poskytují operaci sčítání⁶ (násobení, dělení atd.) a lze je vypsat v jisté definované formě⁷.
- Abychom umožnili sčítání komplexních čísel, předefinujeme význam standardního operátoru +. Podobně postupujeme i v případě vypisování čísel – tentokrát s operátorem <<.

Definice konstruktoru třídy a dvou jednoduchých metod Cast_Real a Cast_Img se nacházejí již uvnitř deklarace třídy ohrazené složenými závorkami { }. Rozhodnutí o tom, kam definici umístit, nejčastěji závisí na délce kódu: pokud je metoda značně rozsáhlá⁸, obvykle ji přemisťujeme vně deklarace třídy a v rámci deklarace ponecháváme jen hlavičku.

Deklarace ukázkového objektu 20+10*j vypadá v programu takto:

- **případ 1:** (implicitní tvorba objektu pomocí jeho deklarace):

Complex NazevObjektu(20, 10);

⁶ Příklad zahrnuje pouze operaci sčítání – zbývající aritmetické operace můžete snadno napsat samostatně.

⁷ Reprezentaci pomocí modulu a fáze můžete vytvořit sami v rámci jednoduchého programátorského cvičení.

⁸ Obecně se doporučuje pravidlo, že bychom při konstrukci procedury neměli překračovat jednu stránku – díky tomu dokážeme celý kód přehlédnout najednou a nemusíme jím horečně listovat.

PŘÍLOHA A Seznámení s jazykem C++

- **případ 2:** (explicitní tvorba objektu pomocí klíčového slova new):

```
Complex *NazevObjektu_Ukz = new Complex(20, 10);
```

Metody lze volat pomocí standardní „tečkové“ notace:

```
NazevObjektu.NazevMetody(parametry); // případ 1
```

nebo

```
NazevObjektu_Ukz->NazevMetody(parametry); // případ 2
```

Když jsme se seznámili se základními principy fungování tříd, podívejme se, jak realizovat chybějící metody.

Funkce `vypis` je natolik triviální, že bychom ji stejně dobře mohli definovat přímo v těle třídy. Protože se jedná o metodu třídy `Complex`, musíme o tom informovat komplilátor tím, že před jejím názvem uvedeme název třídy zakončený operátorem `:` (jedná se o syntaktické pravidlo). Jako metoda třídy má tato procedura přístup k soukromým položkám objektu, který ji aktivoval. Pokud by k parametrům této metody patřil jiný objekt třídy `Complex` (např. `Complex x`), mohli bychom k jeho položkám přistupovat pomocí tečkové notace. Příklad: `x.Re`.

```
complex.cpp
void Complex::vypis()
{
    cout << Re << "+j*" << Im << endl;
}
```

Jazyk C++ umožňuje snadno *předefinovat význam standardních operátorů*, aby byly operace s objekty co nejprostší. Vzhledem k tomu, že komplexní čísla se sčítají poněkud odlišně než běžná čísla, bude vhodné ukrýt způsob sčítání uvnitř funkce a navenek k tomuto účelu ponechat operátor `+`. Operátor se dvěma argumenty lze nejvhodněji předefinovat pomocí tzv. *spřátné funkce*. Je to speciální funkce, která sice nepatří k metodám⁹ konkrétní třídy, ale může operovat jejími objekty. Týká se to rovněž přístupu k soukromým položkám.

Nová spřátná funkce funguje následovně: jako parametry jsou předány dva objekty `x` a `y`. Po přečtení hodnot jejich položek `Re` a `Im` je možné zkonstruovat nový objekt třídy `Complex` podle jednoduchého vzoru: $(a+j\cdot b)+(c+j\cdot d) = (a+c)+(b+d)\cdot j$. Nově vytvořený objekt je vrácen pomocí odkazu – neboli jako plně adresovatelný objekt. Lze jej připsat jinému objektu, pro který je případně aktivována nějaká metoda třídy `Complex` atd. Odpovídající kód tedy vypadá takto:

```
Complex x(1,2),y(2,3),c; // deklarace objektů
c=x+y;                  // c = (1+2)+j·(2+3)
```

Podívejme se na výpis funkce `+`:

```
Complex& operator +(Complex x,Complex y)
{
    double tmp_Re=x.Cast_Real()+y.Cast_Real();
    double tmp_Im=x.Cast_Img() +y.Cast_Img();
    Complex *NovyObjekt=new Complex(tmp_Re,tmp_Im);
    return (*NovyObjekt);
}
```

⁹ Proto je nelze volat pomocí tečkové notace.

Všimněme si toho, že objekt NovyObjekt je vytvořen *explicitním* způsobem pomocí klíčového slova new. Tento postup zajišťuje, že se vrácený odkaz bude vztahovat k trvalému objektu. (Běžná instrukce Complex NovyObjekt použitá uvnitř bloku by vytvořila dočasný objekt, který by po provedení příkazů v daném bloku zmizel.)

Podobně jako v případě operátoru + je praktické předefinovat operátor <<, který vysílá formátovaná data do výstupního datového proudu. V jazyce C++ k tomu slouží třída s názvem ostream. Aniž bychom zabíhali do detailů¹⁰, zapamatujme si alespoň podstatu následujícího triku:

```
ostream& operator << (ostream &str,Complex x)
{
    str << x.Cast_Real()<< "+j*" << x.Cast_Img();
    return str;
}
```

Podívejme se konečně na ukázkový program, který vytváří objekty a manipuluje s nimi:

```
#include "complex.h"
int main()
{
    Complex c1(1,2),c2(3,4);
    cout << "c1=";
    c1.vypis();
    cout << "c2=";
    c2.vypis();
    cout << "c1+c2=" <<(c1+c2) << endl;
    Complex *c_ukz=new Complex(1,7);
    cout << "a)c_ukz odkazuje na objekt";
    c_ukz->vypis();
    cout << "b) c_ukz odkazuje na objekt"<<*c_ukz<< endl;
}
```

Pro úplnost uvedeme výsledky spuštění programu:

```
c1=1+j*2
c2=3+j*4
c1+c2=4+j*6
c_ukz odkazuje na objekt 1+j*7
c_ukz odkazuje na objekt 1+j*7
```

Statické složky tříd

Každý nově vytvořený objekt se vyznačuje jistými jedinečnými vlastnostmi (hodnotami svých atributů). Občas však potřebujeme disponovat něčím jako globální proměnnou v rámci dané třídy. K tomuto účelu slouží tzv. *statické položky*.

Když v definici třídy C uvedeme před název atributu x klíčové slovo static, vznikne proměnná právě tohoto typu. Takovou položku můžeme inicializovat dokonce ještě před vytvořením prvního objektu třídy C! Přitom stačí napsat

C::x=nějaká_hodnota;

¹⁰ Nemáme zde místo, abychom se zabývali poměrně složitou hierarchií knihoven tříd, které se dodávají s dobrými kompilátory C++. Začínající programátoři v jazyce C++ by takový popis mohli snadno odradit.

PŘÍLOHA A Seznámení s jazykem C++

Principiálně podobné jsou *statické metody*: ty lze také volat ještě před vytvořením libovolného objektu. Statické metody jsou samozřejmě omezeny tím, že nemají přístup k *nestatickým položkám dané třídy*. Kromě toho v jejich případě nemá smysl používat ukazatel `this`. Chceme-li statické metodě umožnit přístup k nestatickým položkám jistého objektu, musíme jej metodě předat jako parametr.

Konečné metody tříd

Programátor může metodu dané třídy označit jako *konečnou* (např. `void fun() const;`). Tento název je poněkud zavádějící. Tato metoda ve skutečnosti deklaruje, že nikdy nemodifikuje položky objektu, pro který byla aktivována.

Dědičnost vlastností

Předpokládejme, že máme k dispozici pečlivě sestavené třídy A a B. Dostali jsme je v podobě *zkompilovaných knihoven*, tzn. kromě spustitelného kódu máme k dispozici jen podrobně komentované hlavičkové soubory s informacemi, jak se metody používají a jaké mají dostupné atributy.

Autor tříd A a B bohužel zvolil několik možností, které nám příliš nevyhovují, a zdá se nám, že bychom to dokázali udělat trochu lépe.

Musíme kvůli tomu napsat vlastní třídy A a B a dostupné knihovny zahodit? Každému je asi jasné, že pokud by odpověď nebyla záporná, pak bychom si tuto řečnickou otázku nekladli. Jazyk C++ umožňuje velmi snadno „opakově používat“ již napsaný (dokonce i zkompilovaný) kód a přitom dovoluje v kódu provést potřebné změny. Jako příklad vezměme deklarace dvou tříd A a B, které jsou součástí následujícího výpisu:

```
dziedzic.h
class C1
{
protected:
    int x;
public:
    C1(int n) // konstruktor
    {
        x = n;
    }
    void pis()
    {
        cout<<"**Stará verze ";
        cout <<"metody 'pis:x=" << x << endl;
    }
};

class C2
{
private:
    int y;
public:
    C2(int n) // konstruktor
    {
        y = n;
    }
};
```

```

int ret_y()
{
    return y;
}

```

Klíčové slovo **protected** (*chráněný*) označuje, že informace v této sekci mají sice z hlediska uživatele třídy soukromý charakter, ale budou předány případné odvozené třídě. To znamená, že odvozená třída je bude moci používat běžným způsobem pomocí názvu, ale uživatel k nim již nebude moci přistupovat např. pomocí „tečkové“ notace. Ještě větší omezení se vztahuje na *soukromé* položky: odvozená třída ve svých metodách nadále nemůže odkazovat na jejich názvy. Toto znepřístupnění můžeme samozřejmě šikovně obejít, když definujeme specializované metody, které poskytují *kontrolovaný* přístup k položkám třídy.

Tento typ ochrany dat skvěle izoluje tzv. *uživatelské rozhraní* od bezprostředního přístupu k datům, ale to je již téma na samostatnou kapitolu.

Pustme se konečně do analýzy konkrétního příkladu programu. Nová třída C dědí vlastnosti od tříd A a B a doplňuje k nim některé vlastní prvky.

```

dziedzic.cpp
#include "dziedzic.h"
class C3:public C1,C2
{
    int z; // soukromá položka
public:
    C3(int n) : C1(n+1),C2(n-1) // nový
    {                               // konstruktor
        z=2*n;
    }
    pis_vse()
    {
        cout << "Všechny položky:\n";
        cout << "\t x=" << x << endl;
        cout << "\t y=" << ret_y() << endl;
        cout << "\t z=" << z << endl;
    }
};

int main()
{
    C3 ob(10);
    ob.pis_vse();
}

// výsledek:
// Všechny položky:
// x = 11
// y = 9
// z = 20

```

Konstruktor třídy C3 inicializuje vlastní proměnnou z a kromě toho ještě volá konstrukty tříd C1 a C2 s takovými parametry, jaké aktuálně potřebuje. Pořadí volání konstruktorů má svou logiku: nejdříve se volají konstruktory bázových tříd (podle svého pořadí na seznamu, který následuje za

PŘÍLOHA A Seznámení s jazykem C++

dvojtečkou) a teprve nakonec konstruktor třídy C3. V našem případě byly parametry n+1 a n-1 zvoleny náhodně.

Kód z předchozích výpisů je názorně vysvětlen na obrázku A.2.

V jazyce C++ může mít stejný název několik funkcí, které se liší svým obsahem. Tato situace se označuje jako *přetížení* a „správná“ funkce se přitom pozná podle typu svých vstupních parameterů. Pokud jsou například v programovém souboru definovány dvě procedury: `void p(char* s)` a `void p(int k)`, pak se volání `p(12)` bude nepochybně týkat té druhé verze.

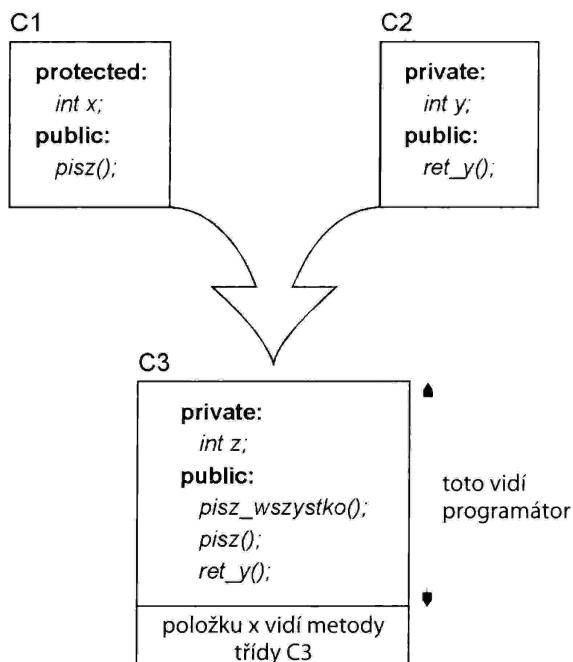
Mechanismus přetížení lze velmi účinně kombinovat s mechanizmy dědění. Řekněme, že se nám nelibí funkce `pis` dostupná ve třídě C3 proto, že ji tato třída zdědila od třídy C1. Na druhou stranu nám vyhovuje název `pis` a chtěli bychom jej používat pro objekty třídy C3, ale k jinému účelu. Doplňme tedy do třídy C3 následující definici¹¹:

```
void C3::pis()
{
    cout << "nová verze metody 'pis'\n";
}
// výsledek použití ob.C1::pis(); ve funkci main:
// ** stará verze metody 'pis': x = 11 **
// výsledek použití ob.pis(); ve funkci main:
// ** nová verze metody 'pis': z = 20 **
```

Příkaz `ob.pis()` nyní vyvolá novou metodu `pis` (ze třídy C3). Pokud bychom však trvali na tom, že chceme použít starou verzi, museli bychom o to výslovně požádat příkazem `ob.C1::pis()`.

Náš příklad obsahuje několik záměrných opomenutí. Problematika dědičnosti vlastností v jazyce C++ totiž zahrnuje mnoho aspektů, které by pro nepoučeného adepta mohly být příliš náročné. Kromě toho informace uvedené v předchozí části kapitoly již postačují k tomu, abychom mohli tvořit poměrně složité objektově orientované aplikace. Jiné mechanizmy, jako jsou např. velmi důležité *virtuální funkce* a tzv. *abstraktní třídy*, již musíme ponechat specializovaným publikacím.

¹¹ Kromě toho je potřeba do veřejné sekce třídy C3 připsat řádek `void pis();`



Obrázek A.2: Příklad dědičnosti vlastností v jazyce C++

Podmíněný kód v jazyce C++

Ve zdrojových souborech (zejména v archivu ke stažení) se můžeme setkat s direktivami, které zajišťují podmíněné provádění kódu programu.

Pokud má fragment kódu strukturu tohoto typu:

```

#define TEST // (*)
// nějaké deklarace a příkazy (1)
#ifndef TEST
// nějaké jiné deklarace a příkazy (2)
#endif

```

kompilátor provede instrukce v části (1). Pokud bychom nyní odstranili řádek se symbolem (*) nebo jej označili jako komentář, program by vykonal příkazy obsažené v části (2).

PŘÍLOHA B

Úvod do číselných soustav

V této kapitole:

- Několik definic
- Dvojková soustava
- Osmičková soustava
- Šestnáctková soustava
- Proměnné v paměti počítače
- Kódování znaků

V této příloze představíme základní principy kódovacích systémů – dvojkové a šestnáctkové soustavy a binární aritmetiky. Programátoři se totiž často nechťejí přiznat k tomu, že jim tato problematika dělá problémy. Případné mezery ve svých znalostech si tedy mohou doplnit právě v této části.

Několik definic

Na úvod zavedeme pojem *poziční soustavy*, ve které čísla zapisujeme pomocí číslic c_1, c_2, \dots, c_n z jisté nevelké množiny. Tento zápis interpretujeme jako součet součinů čísel, která jsou reprezentována jednotlivými číslicemi, a přirozených mocnin čísla n , které označujeme jako základ systému. Jejich exponenty se přitom rovnají pozici číslice v posloupnosti. V informatice se používají zejména systémy se základem 2, 8 a 16 – neboli systém dvojkový, osmičkový a šestnáctkový.

Elektronické obvody, které jsou základem moderních počítačů, provádějí aritmetické operace a řídí se přitom principy tzv. Booleovy algebry. Ta je tvořena sadou pravidel, která pracují pouze se dvěma prvky označenými 0 a 1. Souhrn zásad této algebry najdeme v tabulce B.1.

Tabulka obsahuje tyto logické operace: součet (+), součin (·) a doplněk neboli komplement (čárka nad prvkem nebo před ním).

Dvojková soustava

Dvojková soustava umožňuje zapisovat čísla pomocí dvou dohodnutých znaků: 0 (nula) a 1 (jednička). Přitom se nezabýváme tím, jaký je konkrétní význam obou těchto stavů (např. nula a 5 voltů) pro daný počítač (či zařízení, např. integrovaný obvod). Z našeho hlediska záleží pouze na tom, že pro mnoho aplikací je dvojková reprezentace čísel nejvhodnější. Měli bychom jí dobře porozumět tím spíše, že ji dokonale podporují i programovací jazyky včetně C++.

PŘÍLOHA B Úvod do číselných soustav

Tabulka B.1: Pravidla a tvrzení Booleovy algebry

Pravidla komutativity	Pravidla asociativity	Pravidla distributivity
$x + y = y + x$	$(x + y) + z = x + (y + z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$
Pravidla neutrality 0 a 1	Pravidla komplementarity	Pravidla idempotence
$x \cdot y = y \cdot x$	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
$x + 0 = x$	$x + \neg x = 1$	$x + x = x$
$x \cdot 1 = x$	$x \cdot \neg x = 0$	$x \cdot x = x$
		$x + 1 = 1$
		$x \cdot 0 = 0$
Pravidla absorpcie	De Morganovy zákony	
$(x \cdot y) + x = x$	$\neg(x + y) = \neg x \cdot \neg y$	
$(x + y) \cdot x = x$	$\neg(x \cdot y) = \neg x + \neg y$	

Podívejme se, jak můžeme v desítkové soustavě přirozené pro člověka vyjádřit číslo 1624:

$$1624 = 1 \cdot 1000 + 6 \cdot 100 + 2 \cdot 10 + 4 = 1 \cdot 10^3 + 6 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0$$

Sčítáme „váhy“ (1, 6, 2 a 4) násobené mocninami základu číselné soustavy (zde se jedná o číslo 10) a dostáváme konkrétní hodnotu, kterou chceme znázornit. Nebo naopak: libovolné číslo můžeme zase rozložit pomocí výše uvedené reprezentace.

Desítková soustava bohužel neumožňuje snadno vytvořit elektronická zařízení, která by operovala s čísly této soustavy. Kromě toho, že se komplikuje vnitřní reprezentace (na technické úrovni je potřeba zakódovat až deset stavů, např. hradel a tranzistorů), jsou navíc výpočty v desítkové soustavě dosti paměťově náročné.

Dvojková soustava reprezentuje čísla pomocí dvou znaků: 0 a 1. Základem systému je dvojka.

Podívejme se, jak lze dvojkové číslo snadno převést na jeho desítkovou reprezentaci.

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$$

Skutečné počítače jsou vybaveny vnitřními registry, které umožňují ukládat skupiny bitů a pracovat s nimi. Nejznámější jednotka, pomocí níž se měří délka těchto skupin, se nazývá *bajt* a obsahuje osm bitů.

Bajt může reprezentovat čísla od 00000000 do 11111111 neboli desítkově od 0 do 255.

V případě dvojkové soustavy bylo nutné vyřešit zásadní problém, jak vyjadřovat záporná čísla. Nejdříve se uvažovalo o tom, že by bylo možné znaménko označit pomocí nejvýznamnějšího bitu (bitu na začátku čísla), např. hodnota 1 by udávala záporné číslo a 0 kladné: 0011 = 3, 1011 = -3. Tento systém se z technických důvodů neujal, protože hodně problémů způsobovalo nejednoznačné vyjádření nuly (0000 = +0, 1000 = -0????). Tehdy informatikům pomohli matematici, kteří vymysleli systém *dvojkového doplňku*, kde záporné číslo získáme tak, že všechny bity negujeme (0 na 1, 1 na 0) a k výsledku přičteme jednotku.

Příklad:

$$\begin{array}{r}
 1101 \quad +13 \\
 0010 \quad \text{negujeme bity} \\
 +0001 \quad \text{přičteme číslo 1} \\
 0011 \quad \text{výsledek -13}
 \end{array}$$

Výhoda systému dvojkového doplňku spočívá v tom, že řeší problém znaku nuly a obejde se bez operace odečítání – stačí přičíst číslo s negovanými bity inkrementované o jednotku.

Aritmetické operace s dvojkovými čísly

Aritmetika dvojkových čísel se podobá desítkové aritmetice (využíváme „sloupcová“ pravidla, která známe již ze základní školy). Podívejme se na několik jednoduchých příkladů, které shrnuje tabulka B.2.

Tabulka B.2: Aritmetika dvojkových čísel na příkladech

	Příklad	Poznámka
Sčítání	$\begin{array}{r} A = 11011 = 27_{10} \\ B = 11001 = 25_{10} \\ \hline A+B = 110100 \end{array}$	Nezapomínejme přenést 1 pro (1+1)!
Odčítání	$\begin{array}{r} A = 11010 = 26_{10} \\ B = 10001 = 17_{10} \\ \hline A-B = 1001 \end{array}$	V případě odčítání (0–1) si musíme „půjčit“ 1 na následující pozici čísla.
Násobení	$\begin{array}{r} A = 101 = 5_{10} \\ B = 010 = 2_{10} \\ \hline 000 \\ 101 \\ \hline A*B = 1010 \end{array}$	Stejně jako v desítkové soustavě (násobíme postupně dalšími číslicemi a posunujeme se vlevo).
Dělení	$\begin{array}{r} 110 \\ 1100 : 10 = 12_{10} : 6_{10} = 2_{10} \\ -10 \\ \hline 0100 \\ -10 \\ \hline 000 \end{array}$	Stejně jako v desítkové soustavě (dělitel posunutý vlevo odčítáme od dělence, dokud nezískáme 0 nebo zbytek).

Logické operace s dvojkovými čísly

Základní logické operace, které lze provádět s dvojkovými čísly, uvádí tabulka B.3. V programech se často pomocí bitového kódování často určují parametry volání funkcí – obvykle nízkoúrovňových, např. soubory.

Tabulka B.3: Základní operace s dvojkovými čísly

	Zásada počítání výsledku operací se dvěma bity	Příklad (zapsaný v konvenci C++)
NEBO (ang. OR) – logický součet	Pokud má alespoň jeden z bitů hodnotu jednu, výsledek se rovná 1.	$1101 \mid 0001 = 1101$
A (ang. AND) – logický součin	Pokud má alespoň jeden z bitů hodnotu nula, výsledek se rovná 0.	$1001 \& 1100 = 1000$
Nonekvivalence ¹¹ (ang. XOR)	Pokud se oba bity rovnají, výsledek je nula.	$1100 \wedge 1001 = 0101$
Negace	Z 1 se stává 0 a 0 se mění na 1.	$\sim 1101 = 0010$

PŘÍLOHA B Úvod do číselných soustav

	Zásada počítání výsledku operací se dvěma bity	Příklad (zapsaný v konvenci C++)
Posunutí vlevo o n bitů	-	proměnná $<< n$
Posunutí vpravo o n bitů	-	proměnná $>> n$

```
#define VOLBA_1 1          // 0001
#define VOLBA_2 2          // 0010
#define VOLBA_3 4          // 0100
#define VOLBA_4 8          // 1000
...
int volby = 0;
volby = VOLBA_1 | VOLBA_4; // "zapnutí" voleb 1 a 4, výsledek: 1001
```

S principy bitových operací se můžeme seznámit při analýze následujícího programu, který ukazuje několik typických bitových operací a při té příležitosti představuje jednoduchý způsob, jak zobrazovat číslo v dvojkovém tvaru.

```
bit_operations.cpp
#include <iostream>
using namespace std;
void showbits(unsigned char s)
// funkce zobrazuje binární reprezentaci znaku
{
unsigned char vahy[8]={1,2,4,8,16,32,64,128}; // maska bitu váhy
for(int i=7; i >= 0; i--)
{
    int bit = (vahy[i] & s);
    if (bit !=0 )
        cout << '1';
    else
        cout << '0';
}
// --Ukázkové operace s bity
int main()
{
    cout << "i\Binárně\tPosun vlevo\tNegace\n";
    for (int i=0; i<16; i++)
    {
        cout << i << "\t";    showbits(i); cout << "\t"; // desítkově a binárně
                               // posunutí o 1 bit vlevo:
        int j= i << 1;
        showbits(j);
        cout << "\t";
        int k= ~i;
        showbits(k);
        cout << endl;           // bitová negace
    }
}
```

Výstup programu:

i	Binárně	Posun vlevo	Negace
0	00000000	00000000	11111111
1	00000001	00000010	11111110
2	00000010	00000100	11111101
3	00000011	00000110	11111100
4	00000100	00001000	11111011
5	00000101	00001010	11111010
6	00000110	00001100	11111001
7	00000111	00001110	11111000
8	00001000	00010000	11110111
9	00001001	00010010	11110110
10	00001010	00010100	11110101
11	00001011	00010110	11110100
12	00001100	00011000	11110011
13	00001101	00011010	11110010
14	00001110	00011100	11110001
15	00001111	00011110	11110000

Osmičková soustava

Osmičková čísla se zapisují v osmičkové poziční soustavě, tj. pomocí osmi číslic od 0 do 7 (např. 074, 0322). V jazyce C++ se za osmičkové číslo považuje číselný údaj, který začíná nulou.

Osmičkový zápis lze snadno převést na dvojkový (a naopak): stačí zaměnit osmičkové číslice na tři dvojkové cifry nebo obráceně. Příklad:

$$(78)_{10} = (001001110)_2 = (116)_8$$

Šestnáctková soustava

Šestnáctková soustava umožňuje stručným způsobem zapisovat dlouhá binární čísla. Číslice šestnáctkové soustavy označují čtveřice bajtů (půlabajty), které jsou uvedeny v následující tabulce B.4.

Tabulka B.4: Převodní tabulka šestnáctkových číslic na dvojkové a desítkové

Šestnáctková číslice	Desítková hodnota	Dvojková hodnota
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100

PŘÍLOHA B Úvod do číselných soustav

Šestnáctková číslice	Desítková hodnota	Dvojková hodnota
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

V jazyce C++ se šestnáctková čísla označují předponou 0x.

Příklad převodu dvojkového čísla na šestnáctkové:

$$(78)_{10} = (01001110)_2 = (4E)_{16}$$

Proměnné v paměti počítače

Již víme, že počítač funguje ve dvojkové soustavě. Kvůli přehlednosti se v programech občas využívají číselné zápisy s využitím osmičkové nebo šestnáctkové soustavy, ale jediným důvodem je zjednodušení zápisu čísel a snížení rizika chyb (dvojková čísla jsou příliš dlouhá a pro většinu lidí téměř nečitelná). Na tomto místě se tedy můžeme zeptat, jak zásady binárního zápisu čísel v počítači souvisejí s programovacími jazyky, které přeče obsahují celkem srozumitelné a logické datové typy jako int, char, float atp. Jak počítač uchovává číslo 15 a jak znakový řetězec „15“?

Jak jsme již uvedli v kapitole 5, seznam základních typů spolu s jejich přesností je součástí standardu jazyka C++, ale konkrétní implementace těchto typů (např. počet bitů obsazených proměnnou určitého typu) již závisí na použité technologii (hardwaru) a operačním systému. Zásady adresování dat a implementace kódu závisí na hardwarové architektuře (typu počítačového procesoru a operacích, které podporuje v tzv. registrech – speciálních paměťových buňkách). Procesor a hardwarové prostředky lze bezprostředně programovat pomocí nízkoúrovňového jazyka symbolických adres, který se také nazývá assemblér. Programovat v assembléru však v současnosti neumí dokonce ani většina informatiků.

Operační systém můžeme považovat za svého druhu nadstavbu nad hardwarem, která pomáhá izolovat programy tvořené programátory od vrstvy fyzických prostředků počítače. Od operačního systému mimo jiné požadujeme, aby zajistil bezpečný přístup k hardwarovým prostředkům: uživatelské programy jsou izolovány od systémových programů a teoreticky by nemělo být možné adresovat hardwarová zařízení přímo, například posílat příkazy grafické kartě či vypalovače DVD. V praxi to samozřejmě může být jinak, ale měli bychom si uvědomit, že programy manipulující s hardwarem nejsou přenosné a mívají poměrně krátkou životnost.

Na základě těchto úvah si snadno domyslíme, že právě kompilátor daného programovacího jazyka, který je přizpůsoben určité hardwarové platformě (např. procesorem Intel) a systému (např.

Windows XP/Vista) zajišťuje „překódování“ smluvních logických datových typů na jejich „fyzické“ verze, které se nacházejí „někde v paměti“. Programátor se už nemusí starat o to, jakým způsobem počítač ukládá proměnné do operační paměti – samozřejmě za předpokladu, že se o to nemá důvod zajímat (třeba kvůli optimalizaci kódu či programování hardwarových zařízení).

Kódování znaků

Každý uživatel počítače denně zasedá ke klávesnici a vůbec nepřemýslí nad tím, jak stisknutí klávesy se symbolem třeba písmene A způsobí, že se stejné písmeno zobrazí na displeji. Nikoho by již nemělo překvapit, že počítač do paměti nezapisuje přímo znaky, ale užívá k tomu jisté dohodnuté dvojkové reprezentace. O kódování a dekódování znaků se stará hardware. Např. klávesnice „ví“, že po stisknutí klávesy A musí vyslat jistý kód (obyčejně 65), a podobně displej (grafická karta) dokáže na základě přijatého kódu příslušný znak „nakreslit“.

K nejčastěji používaným standardům kódování znaků patří tzv. kód ASCII (ang. *American Standard Code for Information Interchange*). Tento kód je sedmibitový, takže máme k dispozici čísla z intervalu 0–127. Těmto číslům jsou přiřazena písmena anglické abecedy, číslice, interpunkční znaménka (viz tabulku B.6) a kódy, které řídí způsob zobrazování znaků na tiskárně nebo znakovém terminálu (např. znak přechodu na nový řádek, tabulátor, symbol backspace – viz tabulku B.5).

Tabulka B.5: Kód ASCII – řídicí kódy

Desítkově	Šestnáctkově	Znak	Zkratka
0	00	Null	NUL
1	01	Start Of Heading	SOH
2	02	Start of Text	STX
3	03	End of Text	ETX
4	04	End of Transmission	EOT
5	05	Enquiry	ENQ
6	06	Acknowledge	ACK
7	07	Bell	BEL
8	08	Backspace	BS
9	09	Horizontal Tab	HT
10	0A	Line Feed	LF
11	0B	Vertical Tab	VT
12	0C	Form Feed	FF
13	0D	Carriage Return	CR
14	0E	Shift Out	SO
15	0F	Shift In	SI
16	10	Data Link Escape	DLE
17	11	Device Control 1 (XON)	DC1
18	12	Device Control 2	DC2

PŘÍLOHA B Úvod do číselných soustav

Desítkové	Šestnáctkové	Znak	Zkratka
19	13	Device Control 3 (XOFF)	DC3
20	14	Device Control 4	DC4
21	15	Negative Acknowledge	NAK
22	16	Synchronous Idle	SYN
23	17	End of Transmission Block	ETB
24	18	Cancel	CAN
25	19	End of Medium	EM
26	1A	Substitute	SUB
27	1B	Escape	ESC
28	1C	File Separator	FS
29	1D	Group Separator	GS
30	1E	Record Separator	RS
31	1F	Unit Separator	US
32	20	(mezera)	
127	7F	Delete	DEL

Tabulka B.6: Kód ASCII – číslice, písmena abecedy, interpunkční znaky

Desítkové	Šestnáct-kově	Znak	Desítkové	Šestnáct-kově	Znak	Desítkové	Šestnáct-kově	Znak
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	,	71	47	G	103	67	g
40	28	(72	48	H	104	68	h
41	29)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E		78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p

Desítkově řestnáctkově	Znak	Desítkově řestnáctkově	Znak	Desítkově řestnáctkově	Znak
49	31	1	81	51	Q
50	32	2	82	52	R
51	33	3	83	53	S
52	34	4	84	54	T
53	35	5	85	55	U
54	36	6	86	56	V
55	37	7	87	57	W
56	38	8	88	58	X
57	39	9	89	59	Y
58	3A		90	5A	Z
59	3B		91	5B	[
60	3C	<	92	5C	\
61	3D	=	93	5D]
62	3E	>	94	5E	^
63	3F	?	95	5F	-
64	40	@	96	60	'

Sedm bitů v kódování ASCII představovalo jakýsi historický pozůstatek, a protože počítače již pracovaly s osmibitovými slovy, bylo kódování brzy doplněno dalšími kódy s hodnotami nad 127. V horní části tabulky byly umístěny semigrafické znaky, které slouží ke kreslení rámečků (např. , atp.) a znaky národních abeced. Sada znaků v rozšířené části tabulky ASCII závisí na konkrétní zemi a občas i na dodavateli počítačového hardwaru (tiskárny či znakového terminálu). Kvůli podpoře znaků středoevropských jazyků vznikla varianta kódu ASCII s názvem ISO 8859-2 (tzv. ISO Latin-2).

Ve stejné době se bohužel objevilo několik desítek jiných standardů, jak se občas můžeme přesvědčit, když se v uživatelském rozhraní programů, webových stránkách, e-mailových zprávách či textových dokumentech zobrazují místo českých písmen různé podivné znaky. Informace o nejvýznamnějších standardech kódování českých znaků naleznete v části „Kódování češtiny“ následující stránky na české Wikipedii: https://cs.wikipedia.org/wiki/Znaková_sada.

K nevýhodám standardu ASCII patří zmatky způsobené výrobci programů a hardwaru (jednoznačná je v zásadě jen část kódů v rozmezí 0... 127) a chybějící podpora více jazyků současně (v rozšířené verzi standardu ASCII je k dispozici jen 8 bitů, což dává 256 možností).

Alternativou ke kódování ASCII se pomalu stává 16bitový systém kódování Unicode, který umožňuje jednoznačným způsobem reprezentovat až 65 tisíc znaků. Nepotřebujeme přitom žádné do datečné informace, abychom odlišili různé jazyky, či dokonce jednotlivé znaky. Pro zajímavost můžeme uvést, že v souboru standardu Unicode následují znaky v pořadí jejich faktického čtení, a nikoli zobrazení (představme si například text článku v angličtině, který obsahuje citovaná slova či věty v arabštině).

Kapacita kódování Unicode je zdánlivě nevyčerpatelná, ale historie babylónské věže nás učí, že časem se díky lidské jazykové tvořivosti zaplní i toto kódování.

Kompilování ukázkových programů

Obsah archivu ZIP ke stažení

Všechny ukázkové programy a doplňkové materiály jsou k dispozici ke stažení na stránce knihy: <http://knihy.cpress.cz/K2114>. Programy stáhněte a rozbalte na svůj pevný disk, abyste je mohli upravovat nebo zkoušet, jak fungují.

Archiv se soubory ke knize má následující strukturu:

- Složka *CPP* obsahuje programy ke kompliaci (s příponou *.cpp*) a dva dávkové soubory, které zajišťují hromadnou kompliaci pomocí komplilátoru GCC:
 - *kompiluj_gcc.bat* (pro systém DOS/Windows),
 - *kompiluj_gcc.sh* (pro systém Linux).
- Uvedené dávkové soubory se liší pouze standardem zápisu znaků pro přechod na nový řádek.
- Složka *VISUAL* zahrnuje hotové projekty programů ke kompliaci s použitím komplilátoru Microsoft Visual C++ Express Edition. Projekty uložené v této složce využívají zdrojové soubory ze složky *CPP*.
- Ve složce *DODATKI* jsou umístěny materiály stažené z Internetu včetně dodatečných free-warových programů. Mohly by zajímat zvláště čtenáře, kteří chtějí experimentovat s kódováním a kompresí dat. Kromě těchto programů se ve složce nachází i seznam příkazů GCC a jiné doplňkové zdroje.

Bezplatně dostupné komplilátory C++

Ke komplaci většiny ukázkových programů by mělo stačit jednoduché prostředí typu GCC, které obsahuje komplilátory C, C++ atp. Prostředí GCC (neboli GNU Compiler Collection), dříve známé pod názvem GNU C Compiler, zahrnuje komplilátory různých programovacích jazyků. Jeho vývoj probíhá v rámci projektu GNU a lze je používat na základě licence GPL. V systémech vycházejících z Unixu slouží GCC jako základní komplilátor, ale toto prostředí lze nainstalovat také v systémech Windows či Mac OS (poslední uvedený systém je ostatně odvozen od systému BSD Unix).

PŘÍLOHA C Kompilování ukázkových programů



Poznámka: V systémech typu Linux/Unix již kompilátor C++ často bývá nainstalován standardně. Chcete-li si to ověřit, zkuste v textovém terminálu zadat příkaz `c++` nebo `g++`. Pokud se zobrazí chybové hlášení, přejděte do instalátoru své linuxové distribuce (např. YAST v distribuci Suse) a vyberte programovací sadu pro jazyk C++. Pokud chybí kompilátor v systému Ubuntu Linux, stačí jej nainstalovat z terminálu pomocí příkazu `sudo apt-get install g++`.

Chcete-li stáhnout prostředí GCC, přejděte na adresu <http://gcc.gnu.org>, kde můžete v sekci Download/Binaries vybrat verzi pro svůj systém (např. Windows či Linux). Používáte-li systém Windows, mohu doporučit instalaci programu MinGW (<http://www.mingw.org>) – tento název vznikl jako zkratka výrazu „Minimalist GNU for Windows“. Softwarový balík se standardně nainstaluje do složky C:\MinGW. Pokud chcete kompilovat programy na úrovni příkazového řádku, stačí dodat složku C:\MinGW\bin do proměnné prostředí Path¹.

Jestliže pracujete s počítačem Apple a systémem Mac OS, doporučuji registraci ve službě Apple Developer Connection (<http://connect.apple.com>). Tato služba nabízí například stažení celého grafického prostředí Xcode, které obsahuje komplet kompilátorů včetně nejnovějšího kompilátoru C++. Prostředí Xcode je sice komplikované a generuje hodně velkých souborů, ale nemusíte díky němu používat kompilátor C++ z příkazového řádku.

Uživatelům systému Windows však mohu poradit, aby si nainstalovali skvělý bezplatně dostupný kompilátor Microsoft Visual C++ Express Edition, který si mohou stáhnout na adrese <http://www.microsoft.com/express/download>. Na stránce dodavatele je tento kompilátor k dispozici ve verzi online (Web Install) nebo jako bitová kopie disku DVD-ROM, kterou lze vypálit například programem Nero (Offline Install). V tomto případě musíme stáhnout soubor velikosti přibližně 750 MB, takže potřebujeme rychlejší připojení k Internetu. Pokud někoho velikost této instalace odrazuje, může využít bezplatný kompilátor Dev C++. Jedná se o nadstavbu nad kompilátorem GNU C++ (MinGW), která nabízí i české uživatelské rozhraní. Balíček je k dispozici ke stažení na adrese: <http://orwelldevcpp.blogspot.com/>. Instalace je velmi jednoduchá. V zásadě stačí jen spustit instalační program a potom jako výchozí jazyk vybrat čeština.

Kompilace a spouštění

Jak zkompilovat a spustit program z této knihy?

Nejdříve je potřeba stáhnout a do libovolné pracovní složky rozbalit soubory z oficiální stránky knihy (soubory jsou komprimovány v archivu ZIP). Další postup už závisí na příslušném operačním systému a prostředí. Tato kniha neslouží jako kurz práce s kompilátory, takže popíšeme pouze základní varianty komplikace a spouštění programů, které by měly stačit všem čtenářům bez ohledu na to, jaký operační systém a vývojový nástroj používají.

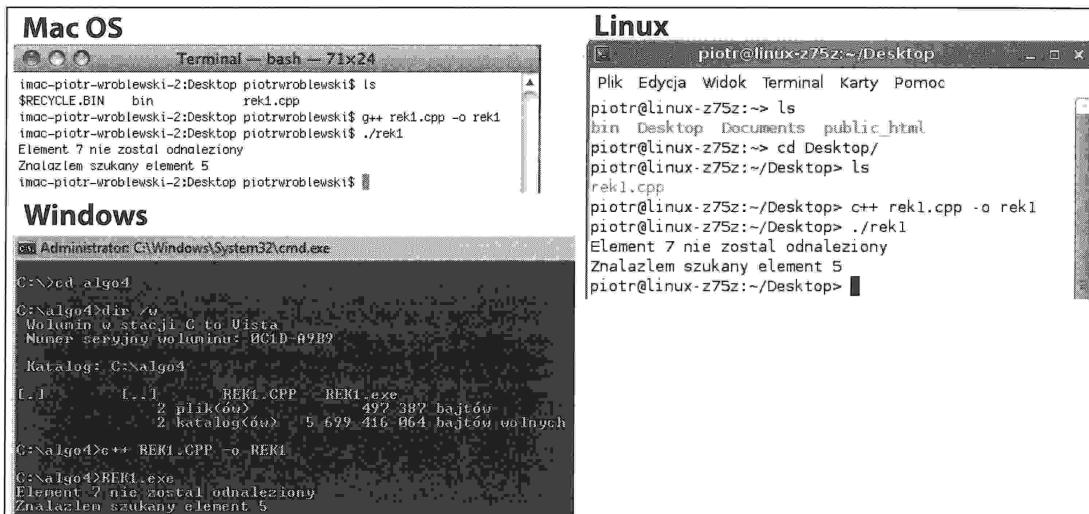
GCC

Způsob komplikace popsán v této části nezávisí na operačním systému. Předpokládáme zároveň, že se při komplikaci omezíme pouze na příkazy zadávané pomocí příkazového řádku (v textovém terminálu), což vzhledem k současnemu rozšíření grafických operačních systémů mnoha osobám činí značné potíže.

Proveďte následující kroky (vycházíme z toho, že kompilátor GCC je již nainstalovaný):

1 Přejděte na Ovládací panely/Systém a údržba/Systém, poté na Pokročilá nastavení systému a Proměnné prostředí, kde můžete za středník připsat novou cestu.

- Spusťte terminál (okno příkazového řádku). V systému Windows lze z nabídky *Start* spustit program s názvem *cmd* (*Start/Všechny programy/Příslušenství/Příkazový řádek*). V systému Mac OS spusťte program *Terminál*. V systému Linux vyhledejte v grafickém prostředí program *Terminál* (např. v distribuci Suse postupně otevřete položky *Počítač/Zobraz všechny programy/Terminál GNOME*).
 - Přejděte do složky, kde je uložen požadovaný soubor se zdrojovým kódem C++, který jste rozbalili z doprovodného archivu ZIP této knihy. K procházení složek pomocí příkazového řádku lze využít příkazů: `cd \`, `cd ..` a `cd název`, které v uvedeném pořadí umožňují přejít do kořenové složky, do nadřazené složky a do podsložky *název*. V systému Ubuntu Linux umístěte ukazatel myši na složku (např. *CPP*), pravým tlačítkem myši otevřete místní nabídku a zvolte příkaz *Otevřít v terminálu*.
 - Na příkazový řádek zadejte příkaz pro komplikaci:
- ```
c++ -o soubor soubor.cpp
```
- Uvedený příkaz zajistí komplikaci příkladu v jazyce C++, který je obsažen v souboru s názvem *soubor.cpp*, a ve stejné složce vytvoří jeho spustitelnou verzi (v závislosti na operačním systému může být k názvu souboru doplněna přípona, např. v systému Windows přípona *.exe*).
  - Zkomplikovaný program v binární formě lze již klasicky spustit dvojím klepnutím myší nebo zadáním jeho názvu ve stejné textové konzoli (příklady na obrázku C.1). Kromě příkazu `cd` lze obsah složky vypsat i příkazem `ls`. Chcete-li spustitelný soubor spustit, musíte před jeho názvem uvést symboly `./` (tečka a lomítko), což je typické pro terminál typu bash. V příkazovém řádku systému Windows stačí zadat název spustitelného souboru spolu s příponou *.exe*.



**Obrázek C.1:** Kompilace a spouštění programu v textové konzoli

## Visual C++ Express Edition

V této části kapitoly popíšeme použití grafického prostředí Visual C++ Express Edition dostupného pro operační systém Windows, které umožňuje kompilovat jak grafické programy, tak i programy fungující v textovém režimu.

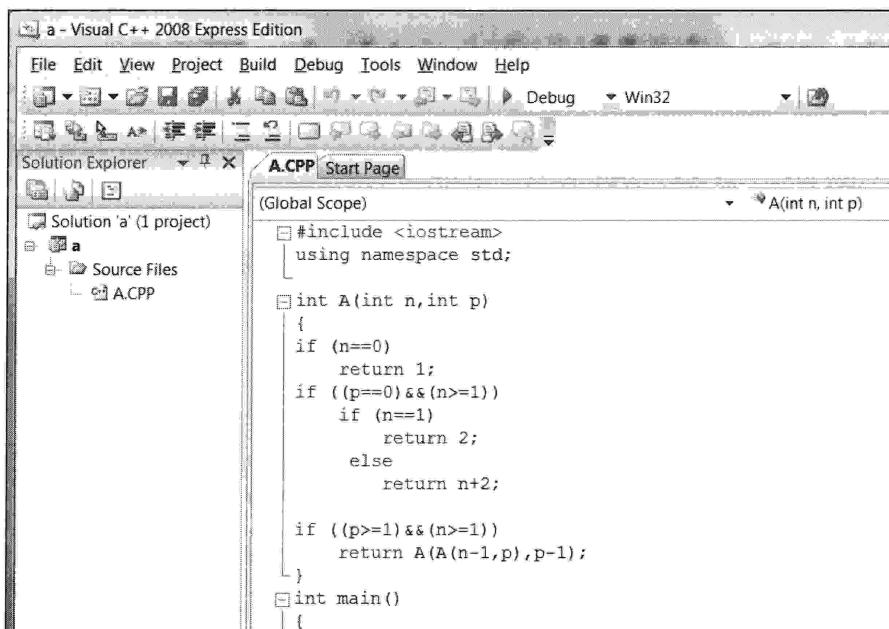
## PŘÍLOHA C Kompilování ukázkových programů

Pro usnadnění naleznete v doprovodném archivu hotové projekty pro toto prostředí. Kromě toho však vysvětlíme, jak lze na základě existujícího souboru CPP vytvořit v tomto kompilátoru nový projekt konzole.

### Otevření hotového projektu

Chcete-li otevřít existující projekt vytvořený v prostředí Visual C++ Express Edition, provedte následující činnosti (obrázek C.2):

- Z nabídky Start spusťte program Visual C++.
- Z nabídky vyberte příkaz *File/Open* (Soubor/Otevřít) a poté zvolte *Project/Solution* (Projekt/Řešení – můžete také použít klávesovou zkratku *Ctrl+Shift+O*).
- Na disku vyberte soubor projektu (s příponou .vcproj) a otevřete jej (klepněte na tlačítko *Otevřít*).
- Kompilace a vytvoření výsledného souboru: klávesa *F7* (také příkaz nabídky *Build/Build Solution* – Sestavit/Sestavit řešení).
- Spuštění kódu: klávesa *F5*.



**Obrázek C.2:** Otevření hotového projektu Visual C++ Express Edition

Po otevření projektu by se na levém panelu prostředí Visual C++ měl zobrazit strom *Solution Explorer* (Průzkumník řešení), ve kterém lze vyhledat zdrojové soubory (ang. *Source Files*) a další dvojím klepnutím je načíst do vizuálního editoru. Prostředí Visual C++ zahrnuje vynikající editor, který zvýrazňuje klíčová slova, pomáhá při formátování (odsazení) a nabízí i nápovědu k syntaxi, což nepochybňě všem programátorem ulehčí práci.

Pokud jste panel *Solution Explorer* náhodou zavřeli a nedokážete bez něj s komplátorem efektivně pracovat, nic vážného se neděje – stačí toto zobrazení znova zapnout v nabídce *View* (Zobrazení). Měli bychom si uvědomit, že konzolový program může po svém spuštění z úrovně prostředí Visual C++ otevřít své okno a po skončení je zase rychle zavřít.

Chceme-li jej na obrazovce „pozdržet“, můžeme například na konec kódu funkce `main` přidat jedenoduchý příkaz na zadávání dat:

```
char klavesa; cin >> klavesa ;
```

nebo

```
cin.get();
```

V systému DOS či Windows lze použít funkci `getch()`:

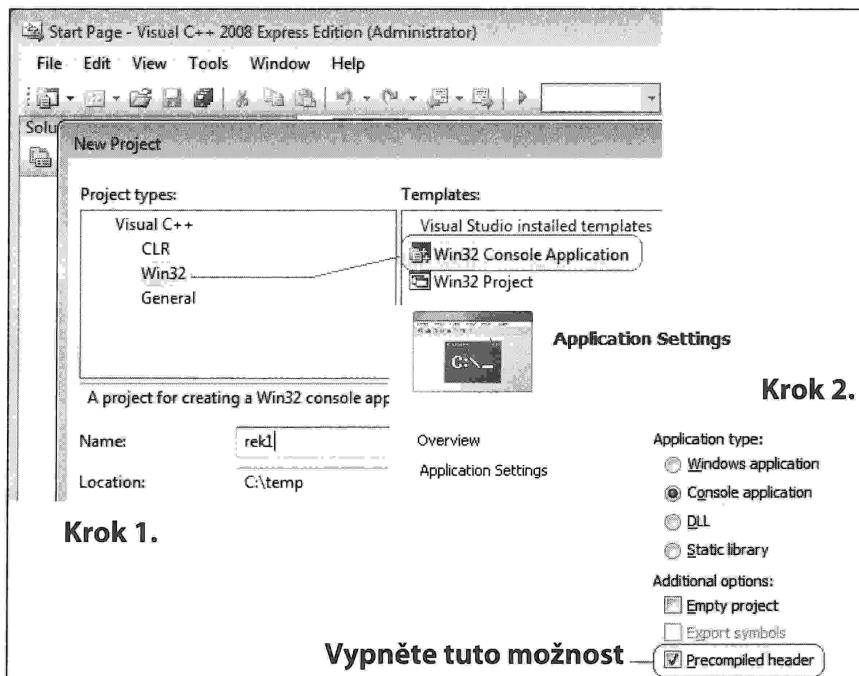
```
#include <conio.h> // kvůli funkci getch
...
getch();
```

### Vytvoření nového konzolového projektu v prostředí Visual C++

Dříve či později se nám určitě bude hodit, když dokážeme v prostředí Visual C++ vytvořit nový projekt. Nemusíme přitom nutně disponovat zdrojovým souborem, ale pro začátek předpokládejme, že takový soubor máme a chceme jej například přizpůsobit svým potřebám.

Chcete-li na základě existujícího souboru `.cpp` vytvořit nový projekt „čistého“ jazyka C++, který bude fungovat v textové konzoli, provedte následující činnosti:

- Z nabídky Start spusťte program Visual C++.
- Z nabídky File/New/Project/Win32 (Soubor/Nový/Projekt/Win32) vyberte typ *Win32 Console Application* (Konzolová aplikace Win32).
- Zadejte název projektu (obrázek C.3).

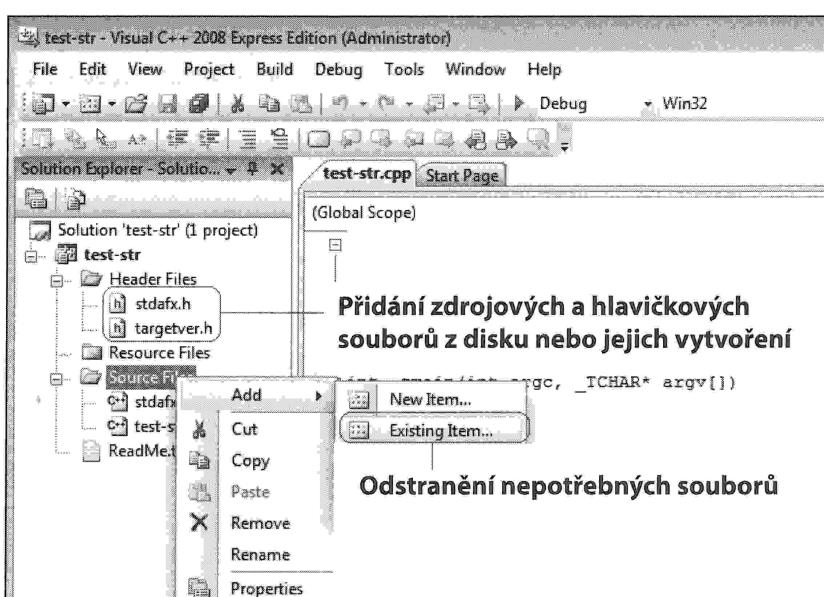


**Obrázek C.3:** Vytvoření tzv. projektu v prostředí Visual C++ Express Edition

## PŘÍLOHA C Kompilování ukázkových programů

(Na obrázku je zachycen okamžik vytvoření projektu s názvem `rek1`, jehož soubory budou uloženy do složky `c:\temp`. Pro zjednodušení je projekt označen stejným názvem jako zdrojový soubor programu, který se v projektu používá, ale samozřejmě to není povinné.)

- V zobrazeném okně *Win 32 Application Wizard* (Průvodce aplikací Win32) stiskněte klávesu *Next* (Další) a vypněte možnost *Precompiled headers* (Předem komplilované hlavičky). Totéž lze provést později v nabídce *Project\<název\_vlastního\_projektu>\Vlastnosti*, v sekci *C/C++/Precompiled Headers* (přístupná teprve po načtení souboru typu `.cpp`).
- V další fázi (obrázek C.4) se již dostanete na známou obrazovku grafického integrovaného vývojového prostředí (IDE) s praktickým průzkumníkem projektových souborů a vynikajícím editorem, který je vybaven funkcemi zvýrazňování klíčových slov jazyka C++ a mechanizmy automatického formátování kódu (například odsazení).
- V průzkumníku projektu přejděte do sekce *Source Files* (Zdrojové soubory) a odstraňte (klávesou *Delete*) soubory, které se tam nacházejí (soubory na pevném disku zůstanou zachovány, ale v případě potřeby je můžete odstranit také).



Obrázek C.4: Kompilace projektu v prostředí Visual C++ Express Edition

- V průzkumníku projektu přejděte do sekce *Header Files* (Hlavičkové soubory) a odstraňte soubory, které se tam nacházejí (soubory na pevném disku zůstanou zachovány, ale v případě potřeby je můžete odstranit také).
- Následně můžete do projektu přidávat vlastní zdrojové soubory: klepněte pravým tlačítkem myši na sekci *Source Files* a z místní nabídky vyberte příkaz *Add/Existing item* (Přidat existující položku). Vyberte požadovaný zdrojový soubor s příponou CPP, který jste stáhli z Internetu nebo vytvořili pomocí běžného textového editoru (stačí systémový Poznámkový blok).
- Stisknutím klávesy *F7* vytvoříte spustitelný soubor, který bude založen na kódu ukázkového programu. Pokud kompilátor oznámí chyby, napravte je<sup>2</sup>.

<sup>2</sup> Autor knihy programy samozřejmě zkontroloval, ale můžete je měnit, což občas vede k chybám.

- Pomocí nabídky *Start* spusťte program *cmd* (Příkazový řádek) a přejděte do složky Debug či Release, jejíž umístění odpovídá hodnotě uvedené v položce Location na obrázku C.3. V tomto příkladu se jedná o cestu *c:\temp\rek1\Debug*.
- Spusťte zkompilovaný program (obrázek C.5).

```

Administrator: Wiersz poleceń
Microsoft Windows [Wersja 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Administrator>cd \temp\rek1\Debug
C:\temp\rek1\Debug>dir /u
Młodź w stacji C to Vista
Numer serwujny voluminu: 0C1D-09B9

Katalog: C:\temp\rek1\Debug

.. 1 rek1.exe rek1_ilk rek1.pdb
 3 plików 1 052 416 bajtów
 2 Katalogów 6 569 467 904 bajtów wolnych

C:\temp\rek1\Debug>rek1
Element ? nie został odnaleziony
Znaleziono szukany element 5

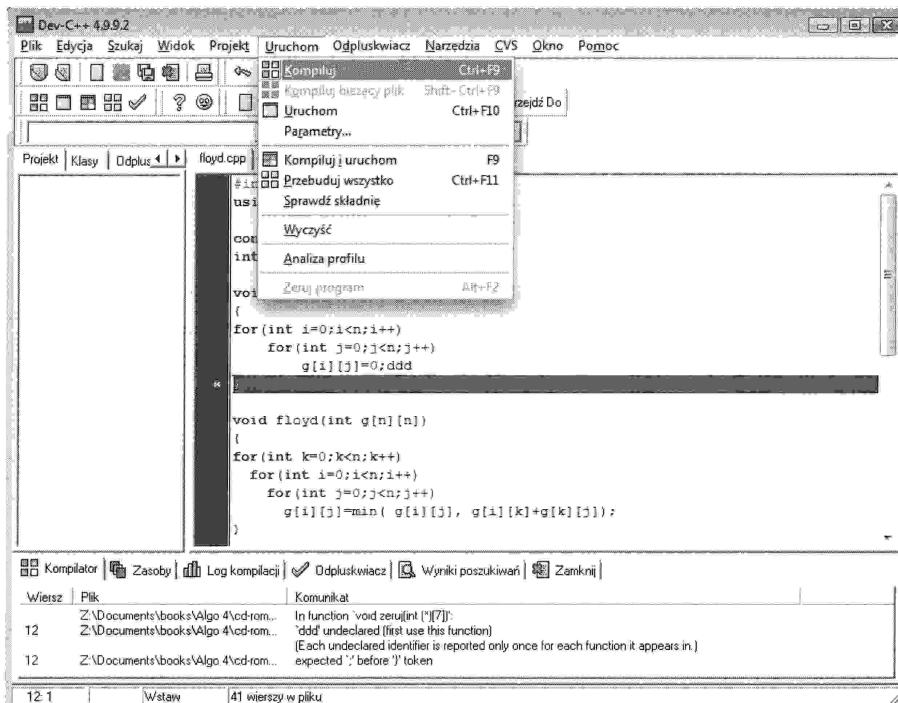
C:\temp\rek1\Debug>

```

**Obrázek C.5:** Spuštění zkompilovaného konzolového programu v systému Windows

## Dev C++

Prostředí Dev C++ (obrázek C.6) je značně oblíbené mezi začínajícími programátory, protože nevyžaduje hodně místa (instalační program s komplikátorem MinGW32 zabírá něco přes 60 MB). Při instalaci si stačí všimmat pouze konfiguračního okna, kde můžeme pro zprávy programu a grafické uživatelské rozhraní zvolit také češtinu.



**Obrázek C.6:** Bezpłatne dostępné prostředí Dev C++

#### **PŘÍLOHA C** Kompilování ukázkových programů

Kvůli komplikaci ukázkového programu dokonce ani není nutné vytvářet soubor projektu. Stačí otevřít soubor v editoru (*Soubor/Otevřít*) a zkompilovat jej příkazem *Spustit/Kompilovat* (*Ctrl+F9*). Jestliže program neobsahoval chyby, objeví se ve stejné složce jako zdrojový soubor `.cpp` spustitelný soubor (`.exe`), který lze spustit např. z příkazového řádku nebo z nabídky *Spustit/Kompilovat* (*Ctrl+F10*). Početstění programu Dev C++ není stoprocentní, ale upozorněním a zprávám o chybách v programech jistě snadno porozumí i programátoři, kteří neovládají angličtinu dokonale. Prostředí totiž uvádí řádky kódu, kde se chyby vyskytly.

# Literatura

- [Ara01] J. Arabas. *Wykłady z algorytmów ewolucyjnych* (Přednášky o evolučních algoritmech). Wydawnictwa Naukowo-Techniczne, 2001.
- [AHU83] A. V. Aho, J. E. Hopcroft a J. D. Ullmann. *Data Structures and Algorithms* (Datové struktury a algoritmy). Addison-Wesley, 1983.
- [BB87] G. Brassard a P. Bratley. *Algorithmique conception et analyse*. Masson Les Presses de l'Université de Montréal, 1987.
- [BC89] L. Bolc a J. Cytowski. *Metody przeszukiwania heurystycznego* (Metody heuristického vyhledávání). PWN, 1989.
- [Ben92] J. Bentley. *Perly programovania*. Alfa, Bratislava, 1992. (Překlad z anglického originálu.)
- [CLR02] Cormen T. H., Leiserson Ch. E., Rivest R.L. *Introduction to algorithms*, 2. ed. (Úvod do algoritmů, 2. vyd.). MIT Press, Cambridge, 2002.
- [CP84] A. Couvert a R. Pendrono. *Techniques de programmation*. IFSIC Cours C45, Octobre 1984.
- [CR90] J. Chojcan a J. Rutkowski. *Zbiór zadań z teorii informacji i kodowania* (Sbírka úloh z teorie informace a kódování). Skriptum č. 1501 Slezské polytechniky v Gliwicích, Gliwice 1990.
- [Del93] C. Delannoy. *Ćwiczenia z języka C++ programowanie obiektowe* (Cvičení v jazyce C++, objektové programování). Wydawnictwa Naukowo-Techniczne, 1993.
- [DF98] E. Dijkstra a W. H. Feijen. *A Method of Programming* (Metoda programování). Addison-Wesley Publishing Company, Wokingham, 1998.
- [Eck00] B. Eckel. *Myslíme v jazyku C++ : Knihovna programátora*. Grada Publishing, Praha, 2000. (Překlad z anglického originálu.)
- [FGS90] C. Froidevaux, M-C Gaudel a M. Soria. *Types de données et algorithmes*. McGraw-Hill (Paris), 1990.
- [Gri87] D. Gries. *The Science of Programming* (Nauka programování). Springer-Verlag, New York, 1987.
- [Har92] D. Harel. *Algorithmics: The Spirit of Computing – Second Edition* (Algoritmika: princip programování, 2. vyd.). Addison-Wesley Publishing Company, 1992.
- [Hel86] J-M. Helary. *Algorithmique des graphes (version partielle)*. IFSIC Cours C66, 1986.

## Literatura

- [HS78] E. Horowitz a S. Sahni. *Fundamentals of Computer Algorithms* (Základy počítačových algoritmů). Computer Science Press, 1978.
- [Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms* (Programování: odvozování algoritmů). Prentice Hall, New York, 1990.
- [Kas03] M. J. Kasperski. *Sztuczna Inteligencja* (Umělá inteligence). Helion, 2003.
- [Kla87] Týmová práce pod vedením Jerzyho Klamky. *Laboratorium metod numerycznych* (Laboratoř numerických metod). Skriptum č. 1305 Slezské polytechniky v Gliwicích, Gliwice 1987.
- [KM03] A. Koenig, B. Moo. *Rozumíme C++*. Computer Press, Praha, 2003. (Překlad z anglického originálu.)
- [Knu10] D. E. Knuth. *Umění programování. 2. díl, Seminumerické algoritmy*. Computer Press, Brno, 2010. (Překlad z anglického originálu.)
- [Knu08] D. E. Knuth. *Umění programování. 1. díl, Základní algoritmy*. Computer Press, Brno, 2008. (Překlad z anglického originálu.)
- [Knu98] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching* (Umění programování: 3. díl, Třídění a vyhledávání). Addison-Wesley Publishing Company, Upper Saddle River, 1998.
- [Kro89] D. Krob. *Algorithmique et structures de données*. Elipses, 1989.
- [Lou99] K. Loudon. *Mastering Algorithms with C* (Ovládнete algoritmy v C). O'Reilly, Beijing, 1999.
- [Nil82] N. J. Nilsson. *Principles of Artificial Intelligence* (Základy umělé inteligence). Springer-Verlag, 1982.
- [Poh99] I. Pohl. *C++ for C Programmers* (C++ pro programátory v jazyce C). Addison-Wesley, Reading, Mass., 1999.
- [Sed92] R. Sedgewick. *Algorithms in C++* (Algoritmy v C++). Addison-Wesley, Reading, 1992.
- [Sed02] R. Sedgewick. *Algorithms in C++. Part 5, Graph algorithms* (Algoritmy v C++. 5. část – grafové algoritmy). Addison-Wesley, Boston, 2002.
- [Say06] K. Sayood. *Introduction to data compression* (Úvod do komprese dat). Morgan Kaufmann, San Francisco, 2006.
- [Str97] B. Stroustrup. *C++ : programovací jazyk*. Softwarové Aplikace a Systémy : BEN - technická literatura, Praha, 2004. (Překlad z anglického originálu.)
- [WF92] K. Weiskamp a B. Flaming. *The Complete C++ Primer – 2nd ed.* (Kompletní úvod do jazyka C++, 2. vyd.). Academic Press, Inc., 1992.

# Rejstřík

## #

#include, 318  
3DES, 288

## A

abeceda, Morseova, 296  
abstraktní  
– datová struktura, 96  
– třída, 334  
Ackermannova, funkce, 69  
algoritmický stroj, 20  
algoritmus, 19, 246  
– analýza složitosti, 51  
– Bellman-Fordův, 245  
– Boyer-Mooreův, 200  
– Dijkstrův, 243  
– Floyd-Warshallův, 240  
– generický, 224  
– hladový, 213  
– K-M-P, 196  
– Karp-Rabinův, 202  
– mini-max, 278  
– Newtonův, 260  
– numerický, 259  
– odstranění rekurze, 163

– optimalizace, 159  
– praktická složitost, 56  
– prefixový, 300  
– Primův, 247  
– Roy-Warshallův, 237–238  
– řezů  $\alpha\beta$ , 279  
– správnost, 26  
– Strassenův, 211  
– třídicí, 75  
– třídy  $O(N \log N)$ , 79  
– třídy  $O(N^2)$ , 76  
– vyhledávací, 179  
analýza  
– bezpečnosti sítě, 228  
– binárního vyhledávání, 68  
– elektronických obvodů, 228  
– rekurzivních programů, 64  
– složitosti algoritmů, 51  
aritmetická  
– operace, 320  
– operace s dvojkovými čísly, 339  
aritmetický  
– operátor, 142  
– výraz, 144  
aritmetika velkých čísel, 287

## Rejstřík

hloubka uzlu, 141  
hodnota, 142  
Hollerith, Herman, 22  
Hornerovo schéma, 261, 291  
hra pro dvě osoby, 277  
Huffmanova metoda, 299  
Huffmanovo kódování, 287

## CH

charakteristická rovnice, 65  
chromatické číslo grafu, 229

## I

IBM 650, 22  
if-else, 173  
IFIP, 23  
implementace  
– množiny znaků, 154  
– seznamů pomocí polí, 122  
integrované vývojové prostředí, 24  
intelligence, umělá, 272  
interpolace  
– funkce, 262  
– funkcí Lagrangeova metoda, 262  
iterace, 325  
iterativní  
– procedura, 162  
– výpočet hodnot funkce, 261

## J

Jacquard, Joseph Marie, 21  
jazyk  
– programovací, 24  
– výběr, 24  
jednosměrný seznam, 97, 110

## K

Karp-Rabinův algoritmus, 202  
klasický postup odstranění rekurze, 171  
klasifikace signálů, 275  
klíč, veřejný, 287, 289  
kód  
– konstrukce, 303  
– podmíněný, 335  
– redundantní, 287  
– zdrojový, 24  
kódování, 183  
– dat, 285  
– Huffmanovo, 287  
– slovníkové, 305  
– znaků, 343  
kódový strom, 301  
kolize, řešení, 185  
kompilace, 347  
kompilátor, 160, 228  
komplexní číslo, 328  
kompozice grafů, 236  
komprese  
– bezeztrátová, 297  
– dat, 285, 296  
– metodou RLE, 299  
– pomocí matematického modelování, 298  
– stupeň, 297  
– ztrátová, 297  
koncepce programů, 24, 205  
koncový uzel, 140, 229  
konečná metoda tříd, 332  
konstruktor, 328  
kování LZW, 305  
krádež klíče, 296  
Kruskalův algoritmus, 246  
křížová rekurze, 44

**L**

ladicí program, 26  
 Lagrangeova metoda, 262  
 LIFO, 127  
 lineární  
   – pokus, 187  
   – rekurzivní posloupnost, 64  
   – vyhledávání, 179  
 linie, otevřená, 280  
 LISP, 25  
 list, 140  
 logaritmický rozklad, 67  
 logická  
   – operace, 321  
   – operace s dvojkovými čísly, 339  
 lokální proměnná, 167  
 luštění šifer, 296  
 LZW, 287, 305, 307

**M**

main, parametry programu, 326  
 MARK 1, 22  
 Markov, Andrey, 22  
 matematická funkce, 56  
 matematické modelování, 298  
 matice  
   – násobení, 209  
   – navrhovaných cest, 239  
   – trojúhelníková, 267  
 metoda  
   – Gaussova, 267  
   – hrubou silou, 296  
   – Huffmanova, 299  
   – Lagrangeova, 262  
   – opačných funkcí, 169

- primitivní, 294
- Simpsonova, 265
- souběžných polí, 124
- Stirlingova, 265
- šplhání, 252
- metodologie programování, 23
- metrika, univerzální, 53
- mini-max, algoritmus, 278
- minimalizace konfliktů, 252
- minimální rozpínavý strom, 245
- množina, 153, 235
  - setříděná, 85
  - uzel, 140
  - znaků, 154
- mocnina grafu, 237
- modelování
  - matematické, 298
  - řešené úlohy, 276
- modul, dialogový, 274
- modulo, 203
- Morseova abeceda, 296
- MYCIN, 273
- myšlení, rekurzivní, 40

**N**

- náročnost, výpočetní, 70
- násobení, 185
  - celých čísel, 212
  - matic, 209
- nejdelší společný podřetězec, 220
- nejznámější funkce H, 183
- neorientovaný graf, 232
- nestabilní algoritmus, 44
- neuronová síť, 274
- Newtonův algoritmus, 260
- nulování části pole, 49

nulový bod funkce, 259  
numerický algoritmus, 259

## O

obecné řešení, 65  
obecný model vnějšího třídění, 90  
objektové programování, 327  
obousměrný seznam, 126  
odkaz, 323  
odstranění rekurze, 159  
okruh, uzavřený, 238  
opačná funkce, 169  
operace  
– aritmetická, 320  
– logická, 321  
– se soubory, 326  
operand, 144  
operátor, 144, 223  
– aritmetický, 142  
– standardní, 330  
optimalizace  
– algoritmu, 159  
– programu, 71  
orientovaný  
– cyklus, 231  
– graf, 229, 231  
osmičková soustava, 341  
otevřená linie, 280  
otevření projektu, 350

## P

paměť počítače, 342  
parametr, předávání, 33  
Pascal, 25  
past, 61  
pivot, 79

počáteční uzel, 229  
počítač, simulování inteligentního chování, 271  
podgraf grafu, 229  
podmíněný kód, 335  
podpis, digitální, 289  
podprogram, 318  
pokročilá programovací technika, 205  
pokus, lineární, 187  
pole, 187, 323  
– centrálních rozdílů, 263  
– dvourozměrné, 233  
– nulování části, 59  
– souběžné, 124, 143  
posloupnost  
– Fibonacciho, 34, 218  
– lineární rekurzivní, 64  
posouvání vzoru, 197  
potomek, 143  
poziční soustava, 337  
poznámka, praktická, 91  
praktická složitost algoritmu, 56  
praktické poznámky, 91  
preference, vyjadřování, 252  
prefixový algoritmus, 300  
primitivní metody, 294  
Primův algoritmus, 247  
prioritní fronta, 134  
problém  
– 8 dam, 224  
– batohu, 214  
– obchodního cestujícího, 231  
– optimálního výběru, 228  
– přenosu klíče, 289  
– reprezentace, 276  
– vhodného výběru, 252

- výběru, 252
  - procedura, 319
    - iterativní, 162
  - proces koncepce programů, 24
  - program
    - koncepce, 24
    - ladící, 26
    - optimalizace, 71
    - rekurzivní, 32, 39
  - programovací
    - jazyk, 24
    - technika, 205, 222
  - programování
    - dynamické, 216
    - metodologie, 23
    - objektové, 327
  - prohledávání
    - grafů, 247
    - sestupné, 248
    - strategie, 251, 278
    - textů, 193
  - projekt, otevření, 350
  - PROLOG, 25
  - proměnná
    - dynamická, 322
    - eliminace, 167
    - globální, 167
    - ukazatel, 322
    - v paměti počítače, 342
  - prvek
    - algoritmiky grafů, 227
    - vložení do seznamu, 103
  - předávání parametrů, 33
  - předek, 143
  - přetížení, 117
  - přímé vkládání, 76
  - případ, elementární, 47
- Q**
- qsort, 82
  - quicksort, 79, 86, 213
- R**
- redukce, zpětná, 267
  - redundantní kód, 287
  - regulérní graf, 229
  - rekurze, 29
    - definice, 29
    - křížová, 44
    - odstranění, 159
    - odstranění s využitím zásobníku, 166
    - rizika, 34
    - ukázka principu, 30
  - rekurzivní
    - myšlení, 40
    - program, 32, 39
    - rovnice, 68
    - struktura, 326
    - volání, 47
    - volání procedury, 162
  - relace, binární, 237
  - reprezentace
    - dvojková, 46
    - grafů, 233
    - pomocí pole, 122, 233
    - problémů, 276
    - stromu, 140
  - reverzní polská notace, 145
  - rizika rekurze, 34
  - RLE, 299
  - role, symetrická, 277
  - rovinný graf, 228

- rovnice
  - charakteristická, 65
  - s více proměnnými, 218
- Roy-Warshallův algoritmus, 237–238
- rozděl a panuj, 206, 213
- rozdíl, centrální, 263
- rozhraní, uživatelské, 333
- rozklad, logaritmický, 67
- rozměr dat, 53
- rozpínavý strom, 245
- rozpoznávání signálů, 275
- R**
- řešení
  - kolizí, 185
  - obecné, 65
  - rekurzivní rovnice, 66
  - soustav lineárních rovnic, 267
- řetězec, znakový, 95
- řídicí struktura, 274
- S**
- sestupné prohledávání, 248
- setříděná množina, 85
- seznam, 97, 186, 235
  - jednosměrný, 97, 110
  - jiných typů, 126
  - obousměrný, 126
- shaker-sort, 78
- shodný typ, 154
- schéma
  - Hornerovo, 261, 291
  - s dvojitým rekurzivním voláním, 175
  - typu if-else, 173
  - typu while, 172
- signál
  - klasifikace, 275
  - rozpoznávání, 275
- Simpsonova metoda, 265
- sít, neuronová, 274
- sled grafu, 229
- slovník uzelů, 235
- slovníková struktura, 148
- slovníkové kódování, 305
- složený typ, 93, 323
- složitost, výpočetní, 62
- slučování, 86
  - setříděných množin, 85
- souběžné pole, 124, 143
- soubor, vstupní, 89
- součet
  - grafu, 236
  - modulo 2, 184
  - modulo Rmax, 184
- soustava
  - číselná, 337
  - dvojková, 337
  - lineárních rovnic, 267
  - osmičková, 341
  - poziční, 337
  - šestnáctková, 341
- souvislý graf, 229
- spirála, 40
- správnost algoritmů, 26
- spřátelená funkce, 107
- spustitelný kód, 24
- stack overflow, 35–37
- standardní operátor, 330
- stanovený vrchol grafu, 243
- statická složka tříd, 331
- stav, 223
- Stirlingova metoda, 265

- Stirlingů, vzorec, 263
- Strassenův algoritmus, 211
- strategie
- do hloubky, 248
  - do šířky, 249
  - prohledávání, 251, 278
  - s návraty, 251
  - výherní, 278
- string, 94, 96
- stroj, algoritmický, 20
- strom
- binární, 140, 144
  - her, 277
  - kódový, 301
  - listy, 140
  - minimální rozpínavý, 245
  - reprezentace, 140
- struktura, 93
- datová, 227
  - rekurzivní, 326
  - řídicí, 274
- sudé čtverce, 42
- switch, 324
- symetrická role, 277
- symetrické šifrování, 287
- symetrický graf, 237
- systém
- dvojkového doplňku, 338
  - expertní, 272, 274
- Š**
- šablona třídy, 129
- šestnáctková soustava, 341
- šifra, luštění, 296
- šifrování
- asymetrické, 289
  - symetrické, 287
- T**
- tah, eulerovský, 232
- technika
- komprese dat, 296
  - optimalizace programů, 71
  - programovací, 205, 222
- terminátor bloku, 310
- terminologie, 64, 328
- text, 95
- prohledávání, 193
- tranzitivní uzávěr grafu, 237
- trojúhelníková matice, 267
- třída
- abstraktní, 334
  - konečná metoda, 332
  - statická složka, 331
  - šablona, 129
- třídění
- bublinkové, 77–78
  - haldou, 82
  - přetřásáním, 78
  - přímým vkládáním, 76
  - slučováním, 86
  - vnější, 75, 87
  - vnitřní, 75
- třídicí algoritmus, 75
- Turing, Alan, 22
- typ
- datový, 93
  - shodný, 154
  - složený, 93, 323
  - základní, 93
- typy výpočetní složitosti, 62

## U

### ukazatel

- na porovnávací funkci, 91
- proměnné, 322

umělá inteligence, 272

union, 94

UNIVAC 1, 22

univerzální

- metrika, 53
- slovníková struktura, 148

UNIX, 305

úplný binární strom, 134

úrovně abstrakce popisu, 24

uzavřený okruh, 238

uzel

- číslo, 229
- hloubka, 141
- koncový, 140, 229
- množina, 140
- počáteční, 229
- slovník, 235
- vnější, 140
- vnitřní, 140
- výška, 141

uživatelské rozhraní, 333

## V

Vandermondův determinant, 262

vector, 94

veřejný klíč, 287, 289

vhodný výběr, 252

virtuální funkce, 334

Visual C++ Express Edition, 349

vlastnost, dědičnost, 332

vložení

- posunem, 77

– prvku do seznamu, 123

vnější

- třídění, 75, 87
- uzel, 140

vnitřní

- třídění, 75
- uzel, 140

volání

- dvojité rekurzivní, 175
- hodnotou, 38
- rekurzivní, 47

vstupní

- soubor, 89
- stupeň vrcholu grafu, 229

výběr jazyka, 24

výherní strategie, 278

vyhledávací algoritmus, 179

vyhledávání

- binární, 45, 180
- hrubou silou, 193, 196
- lineární, 179
- minima a maxima v číselném poli, 207
- nové algoritmy, 195

vyjadřování preferencí, 252

výpočet, zjednodušení, 64

výpočetní

- náročnost, 70
- složitost, 62

výraz, aritmetický, 144

výška uzlu, 141

vytvoření jednosměrného seznamu, 101

využití hešování, 190

vývojové prostředí, 24

vzorec, Stirlingův, 263

**W**

while, 172

**Z**

základní typ, 93

zásobník, 33, 127, 134

– princip fungování, 127

– základní operace, 128

záznam, 324

zdrojový kód, 24

zjednodušení výpočtů, 64

změna definičního oboru rekurzivní rovni-  
ce, 68

znak, kódování, 343

znakový řetězec, 95, 342

znalostní báze, 274

zpětná redukce, 267

ztrátová komprese, 297

Piotr Wróblewski

# Alg or it my

Český překlad čtvrtého vydání bestselleru

*Algorytmy, struktury danych i techniki programowania.*

Hledáte jako programátoři **vhodný algoritmus** pro zamýšlený program? Chcete stávající koncepci programu optimalizovat? Jste při vývoji omezeni dostupnými technickými prostředky a potřebujete je využívat co nejefektivněji? Neobjevujte Ameriku a použijte ve svých projektech ten správný algoritmus.

V aktualizovaném **vydání bestselleru** se naučíte nejznámější a nejčastěji používané algoritmy. Výklad se neomezuje pouze na suché popisy jednotlivých postupů, seznámíte se i s jejich principy a možnými příklady využití. Při nasazení v praxi pak zvolíte vhodné řešení odpovídající problému a podmínek. Každá z kapitol obsahuje i praktické příklady, otázky a cvičení, na kterých si můžete čerstvě nabyté znalosti vyzkoušet.

Publikace vás mimo jiné naučí:

- Analyzovat složitost algoritmů
- Optimalizovat vytvořené algoritmy
- Využívat vhodné struktury při práci s daty
- Zpracovat numerické výpočty
- Třídit, řadit a vyhledávat údaje
- Využívat nejrůznější metody kódování a komprese
- Pracovat s grafy, prohledávat je



Čtenáři si mohou na adrese <http://knihy.cpress.cz/K2114> pod odkazem Soubory ke stažení stáhnout soubory se zdrojovými kódy použitými v knize.

computer  
press

v kompletní nabídce najeznete  
již více než 4400 knižních titulů  
z mnoha zajímavých oborů

Navštivte  
náš  
web!

#### O autorovi:

Piotr Wróblewski působil řadu let ve společnostech orientovaných na telekomunikace, psal software na zakázku a vedl počítačová školení. Je autorem několika knih zaměřených na počítače, jeho bestseller *Algorytmy* vyšel v několika vydáních, a to nejen v Polsku.

#### Zařazení publikace:

|             | začátečník | pokročilý | oborník |
|-------------|------------|-----------|---------|
| uživatel    |            |           |         |
| student     |            |           |         |
| programátor |            |           |         |
| analytik    |            |           |         |

#### Objednávky knih:

[knihy.cpress.cz](http://knihy.cpress.cz)  
[www.albatrosmedia.cz](http://www.albatrosmedia.cz)  
[eshop@albatrosmedia.cz](mailto:eshop@albatrosmedia.cz)

Bezplatná linka

800 555 513

ISBN 978-80-251-4126-7

